- cтарейший интернет-   
 магазин России  
- 14 лет на рынке  
- cкидки постоянным   
 покупателям  
- безопасность покупок  
ÚËÏÂÌÚ, ÓÔÂ‡ÚË‚Ì‡ˇ ‰ÓÒÚ‡‚Í‡   
Ë Û‰Ó·Ì˚Â ÒÔÓÒÓ·˚ ÓÔÎ‡Ú˚   
‰Â Î‡ ˛Ú ÔÓÍÛÔÍË ‚ Ì‡¯ÂÏ Ï‡„‡­  
ÁË ÌÂ ÔËˇÚÌ˚Ï ‚ÂÏˇÔÂÔÓ­  
‚ÓÊ ‰ÂÌËÂÏ.  
œÓÒÚÓˇÌÒÚ‚Ó Ë Ì‡ ‰ÂÊ ÌÓÒÚ¸   
Ì‡ ¯ÂÈ ‡·ÓÚ˚ „‡‡Ì ÚËÛ˛Ú   
·Â Á Ó Ô‡ÒÌÓÒÚ¸ ÔÓÍÛÔÓÍ. ∆‰ÂÏ   
‚‡Ò ‚ Ì‡¯ÂÏ Ï‡„‡ ÁËÌÂ!  
»ÌÚÂÌÂÚ­Ï‡„‡ÁËÌ Books.Ru   
ÓÒ ÌÓ ‚‡Ì ‚ 1996 „Ó‰Û. Ã˚ ÚÓ­  
„Û ÂÏ ÍÌË „‡ ÏË ÔÓ ‚ÒÂ ÏÛ ÏËÛ,   
‰Ó ÒÚ‡‚ Îˇˇ Ëı ‚ Ò‡Ï˚Â Û‰‡ ÎÂÌ­  
Ì˚Â „Ó Ó‰‡ Ë ÒÚ‡ Ì˚.  
—‡Ï˚Â ÌÓ‚˚Â Ë ËÌÚÂÂÒÌ˚Â   
ÍÌË „Ë, ‰ËÒÍË Ë ‚Ë‰ÂÓÍ‡ÒÒÂÚ˚   
‚˚ ‚ÒÂ „‰‡ ÒÏÓÊÂÚÂ Ì‡ÈÚË Û Ì‡Ò   
ÔÂ ‚˚ÏË. »ÌÚÂÂÒÌ˚È ‡ÒÒÓ­  
Books.Ru – ваш ближайший книжный магазин  
ÚÂÎ/Ù‡ÍÒ ÃÓÒÍ‚‡ (495) 638-5305   
—‡ÌÍÚ-œÂÚÂ·Û„ (812) 380-5006

Learning  
Python  
Forth Edition  
Mark Lutz

Изучаем  
Python  
Четвертое издание  
Марк Лутц  
Санкт-Петербург – Москва  
2011

Марк Лутц  
Изучаем Python, 4-е издание  
Перевод А. Киселева  
Главный редактор   
А. Галунов  
Зав. редакцией   
Н. Макарова  
Выпускающий редактор   
П. Щеголев  
Редактор   
Ю. Бочина  
Корректор   
С. Минин  
Верстка   
К. Чубаров  
Лутц М.  
Изучаем Python, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. –   
1280 с., ил.  
ISBN 978-5-93286-159-2  
Такие известные компании, как Google и Intel, Cisco и Hewlett-Packard, исполь-  
зуют язык Python, выбрав его за гибкость, простоту использования и обеспечи-  
ваемую им высокую скорость разработки. Он позволяет создавать эффективные   
и надежные проекты, которые легко интегрируются с программами и инстру-  
ментами, написанными на других языках.  
Четвертое издание «Изучаем Python» – это учебник, написанный доступным   
языком, рассчитанный на индивидуальную скорость обучения и основанный на   
материалах учебных курсов, которые автор, Марк Лутц, ведет уже на протяже-  
нии десяти лет. Издание значительно расширено и дополнено в соответствии с из-  
менениями, появившимися в новой версии 3.0. В книге представлены основные   
типы объектов в языке Python, порядок их создания и работы с ними, а также   
функции как основной процедурный элемент языка. Рассматриваются методы   
работы с модулями и дополнительными объектно-ориентированными инстру-  
ментами языка Python – классами. Включены описания моделей и инструкций   
обработки исключений, а также обзор инструментов разработки, используемых   
при создании крупных программ.  
Каждая глава завершается контрольными вопросами с ответами на закрепле-  
ние пройденного материала, а каждая часть – упражнениями, решения которых   
приведены в приложении В. Книга была дополнена примечаниями о наиболее   
существенных расширениях языка, по явившихся в версии Python 3.1.  
ISBN 978-5-93286-159-2   
ISBN 978-0-596-15806-4 (англ)  
© Издательство Символ-Плюс, 2010  
Authorized translation of the English edition © 2009 O’Reilly Media Inc.. This trans-  
lation is pub lished and sold by permission of O’Reilly Media Inc., the owner of all rights   
to publish and sell the same.  
Все права на данное издание защищены Законодательством РФ, включая право на полное или час-  
тичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные зна-  
ки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.   
Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,  
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.  
Подписано в печать 22.10.2010. Формат 70×100 1/16. Печать офсетная.   
Объем 80 печ. л. Тираж ХХ00 экз. Заказ №  
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»   
199034, Санкт-Петербург, 9 линия, 12.

Посвящается Вере.   
Ты – жизнь моя.

Об авторе  
Марк Лутц (Mark Lutz) является ведущим специалистом в области   
обучения языку программирования Python и автором самых ранних и   
наиболее популярных публикаций. Он известен в сообществе пользо-  
вателей Python своими новаторскими идеями.  
Лутц – автор книг «Programming Python»1 и «Python Pocket Refe ren-  
ce», выпущенных издательством O’Reilly и претерпевших несколько   
изданий. Он использует Python и занимается его популяризацией на-  
чиная с 1992 года. Книги об этом языке программирования он начал   
писать в 1995 году, а его преподаванием стал заниматься с 1997 года.   
На начало 2009 года Марк провел 225 курсов, обучил примерно 3500   
студентов и написал книги по языку Python, суммарный тираж ко-  
торых составил около четверти миллиона экземпляров. Книги Лутца   
переведены более чем на десять языков.  
Марк обладает степенями бакалавра и магистра в области информати-  
ки, закончил университет штата Висконсин (США). На протяжении   
последних 25 лет занимался разработкой компиляторов, инструмен-  
тальных средств программиста, приложений и разнообразных систем   
в архитектуре клиент-сервер. Связаться с Марком можно через веб-  
сайт http://www.rmi.net/~lutz.  
1   
Лутц М. «Программирование на Python», 2-е изд. – Пер. с англ. – СПб.:   
Символ-Плюс, 2002. Четвертое издание этой книги выйдет в 2011 году.

Оглавление  
Об авторе ................................................................................... 6  
Предисловие .............................................................................17  
Часть I. Введение ......................................................................39  
Глава 1. Python в вопросах и ответах ........................................41  
Почему программисты используют Python? ..............................41  
Является ли Python «языком сценариев»?................................44  
Все хорошо, но есть ли у него недостатки? ................................45  
Кто в наше время использует Python? ......................................46  
Что можно делать с помощью Python? .....................................48  
Как осуществляется поддержка Python? ..................................52  
В чем сильные стороны Python? ..............................................52  
Какими преимуществами обладает Python перед языком X? ......57  
В заключение .......................................................................58  
Закрепление пройденного ......................................................59  
Глава 2. Как Python запускает программы ................................63  
Введение в интерпретатор Python ............................................63  
Выполнение программы ........................................................64  
Разновидности модели выполнения .........................................69  
В заключение .......................................................................75  
Закрепление пройденного ......................................................75  
Глава 3. Как пользователь запускает программы ......................77  
Интерактивный режим ..........................................................77  
Системная командная строка и файлы .....................................84  
Щелчок на ярлыке файла.......................................................90  
Импортирование и перезагрузка модулей .................................94  
Запуск модулей с помощью функции exec .............................. 101  
Пользовательский интерфейс IDLE ....................................... 102  
Другие интегрированные среды разработки ............................ 108  
Другие способы запуска ....................................................... 109  
Какие способы следует использовать? .................................... 112  
В заключение ..................................................................... 114  
Закрепление пройденного .................................................... 114

8   
Оглавление  
Часть II. Типы и операции ....................................................... 119  
Глава 4. Введение в типы объектов языка Python ................... 121  
Зачем нужны встроенные типы? ........................................... 122  
Числа ................................................................................ 125  
Строки .............................................................................. 126  
Списки .............................................................................. 133  
Словари ............................................................................. 137  
Кортежи ............................................................................ 144  
Файлы............................................................................... 145  
Другие базовые типы ........................................................... 147  
В заключение ..................................................................... 151  
Закрепление пройденного .................................................... 151  
Глава 5. Числа ....................................................................... 153  
Базовые числовые типы ....................................................... 153  
Числа в действии ................................................................ 162  
Другие числовые типы......................................................... 177  
Числовые расширения ........................................................ 191  
В заключение ..................................................................... 191  
Закрепление пройденного .................................................... 192  
Глава 6. Интерлюдия о динамической типизации ................... 194  
Отсутствие инструкций объявления ...................................... 194  
Разделяемые ссылки ........................................................... 199  
Динамическая типизация повсюду ........................................ 204  
В заключение ..................................................................... 205  
Закрепление пройденного .................................................... 205  
Ответы .............................................................................. 205  
Глава 7. Строки ...................................................................... 207  
Литералы строк .................................................................. 210  
Строки в действии ............................................................... 217  
Строковые методы .............................................................. 227  
Выражения форматирования строк ...................................... 234  
Метод форматирования строк ............................................... 239  
Общие категории типов ....................................................... 249  
В заключение ..................................................................... 251  
Закрепление пройденного .................................................... 251  
Глава 8. Списки и словари ...................................................... 253  
Списки .............................................................................. 253  
Списки в действии .............................................................. 256  
Словари ............................................................................. 264  
Словари в действии ............................................................. 266  
В заключение ..................................................................... 282  
Закрепление пройденного .................................................... 282

Оглавление   
9  
Глава 9. Кортежи, файлы и все остальное ............................... 284  
Кортежи ............................................................................ 284  
Кортежи в действии ............................................................ 286  
Файлы............................................................................... 289  
Пересмотренный перечень категорий типов ............................ 301  
Гибкость объектов .............................................................. 302  
Ссылки и копии .................................................................. 303  
Сравнивание, равенство и истина .......................................... 306  
Иерархии типов данных в языке Python ................................. 310  
Другие типы в Python .......................................................... 312  
Ловушки встроенных типов ................................................. 313  
В заключение ..................................................................... 315  
Закрепление пройденного .................................................... 316  
Часть III. Инструкции и синтаксис ........................................... 321  
Глава 10. Введение в инструкции языка Python ....................... 323  
Структура программы на языке Python .................................. 323  
История о двух if ................................................................ 326  
Короткий пример: интерактивные циклы .............................. 334  
В заключение ..................................................................... 340  
Закрепление пройденного .................................................... 340  
Глава 11. Присваивание, выражения и print ............................ 342  
Инструкции присваивания................................................... 342  
Инструкции выражений ...................................................... 360  
Операция print ................................................................... 362  
В заключение ..................................................................... 374  
Закрепление пройденного .................................................... 374  
Глава 12. Условная инструкция if и синтаксические правила ... 376  
Условные инструкции if ...................................................... 376  
Синтаксические правила языка Python .................................. 379  
Проверка истинности .......................................................... 385  
Трехместное выражение if/else ............................................. 387  
В заключение ..................................................................... 390  
Закрепление пройденного .................................................... 390  
Глава 13. Циклы while и for .................................................... 392  
Циклы while ...................................................................... 392  
break, continue, pass и else .................................................... 394  
Циклы for .......................................................................... 400  
Приемы программирования циклов ...................................... 407  
В заключение ..................................................................... 415  
Закрепление пройденного .................................................... 415

10   
Оглавление  
Глава 14. Итерации и генераторы, часть 1 ............................... 417  
Итераторы: первое знакомство .............................................. 417  
Генераторы списков: первое знакомство ................................. 425  
Новые итерируемые объекты в Python 3.0 .............................. 433  
Другие темы, связанные с итераторами .................................. 439  
В заключение ..................................................................... 439  
Закрепление пройденного .................................................... 439  
Глава 15. Документация ......................................................... 441  
Источники документации в языке Python .............................. 441  
Типичные ошибки программирования ................................... 453  
В заключение ..................................................................... 456  
Закрепление пройденного .................................................... 456  
Часть IV. Функции ................................................................... 459  
Глава 16. Основы функций ..................................................... 461  
Зачем нужны функции? ....................................................... 462  
Создание функций .............................................................. 463  
Первый пример: определения и вызовы ................................. 466  
Второй пример: пересечение последовательностей ................... 469  
В заключение ..................................................................... 472  
Закрепление пройденного .................................................... 472  
Глава 17. Области видимости ................................................. 474  
Области видимости в языке Python ........................................ 474  
Инструкция global .............................................................. 482  
Области видимости и вложенные функции ............................. 487  
Инструкция nonlocal ........................................................... 494  
В заключение ..................................................................... 502  
Закрепление пройденного .................................................... 502  
Глава 18. Аргументы .............................................................. 505  
Передача аргументов ........................................................... 505  
Специальные режимы сопоставления аргументов .................... 511  
Функция поиска минимума .................................................. 525  
Универсальные функции для работы с множествами ............... 528  
Имитация функции print в Python 3.0 ................................... 530  
В заключение ..................................................................... 533  
Закрепление пройденного .................................................... 534  
Глава 19. Расширенные возможности функций....................... 536  
Концепции проектирования функций .................................... 536  
Рекурсивные функции ........................................................ 538  
Функции – это объекты: атрибуты и аннотации ...................... 542  
Анонимные функции: lambda ............................................... 548

Оглавление   
11  
Отображение функций на последовательности: map ................. 554  
Средства функционального программирования:   
filter и reduce ..................................................................... 556  
В заключение ..................................................................... 557  
Закрепление пройденного .................................................... 558  
Глава 20. Итераторы и генераторы ......................................... 560  
Еще раз о генераторах списков:   
функциональные инструменты ............................................. 560  
Еще раз об итераторах: генераторы ........................................ 567  
Краткая сводка по синтаксису генераторов в 3.0...................... 583  
Хронометраж итерационных альтернатив .............................. 586  
Типичные ошибки при работе с функциями ............................ 596  
В заключение ..................................................................... 600  
Закрепление пройденного .................................................... 601  
Часть V. Модули ...................................................................... 605  
Глава 21. Модули: общая картина .......................................... 607  
Зачем нужны модули? ......................................................... 608  
Архитектура программы на языке Python .............................. 608  
Как работает импорт ........................................................... 612  
Путь поиска модулей ........................................................... 614  
В заключение ..................................................................... 620  
Закрепление пройденного .................................................... 621  
Глава 22. Основы программирования модулей ...................... 623  
Создание модуля ................................................................. 623  
Использование модулей ....................................................... 624  
Пространства имен модулей ................................................. 630  
Повторная загрузка модулей ................................................ 635  
В заключение ..................................................................... 639  
Закрепление пройденного .................................................... 640  
Глава 23. Пакеты модулей ..................................................... 641  
Основы операции импортирования пакетов ............................ 641  
Пример импортирования пакета ........................................... 645  
Когда используется операция импортирования пакетов? ......... 647  
Импортирование относительно пакета ................................... 650  
В заключение ..................................................................... 663  
Закрепление пройденного .................................................... 664  
Глава 24. Дополнительные возможности модулей ................. 665  
Сокрытие данных в модулях................................................. 665  
Включение будущих возможностей языка .............................. 666

12   
Оглавление  
Смешанные режимы использования:   
\_\_name\_\_ и \_\_main\_\_ .......................................................... 667  
Изменение пути поиска модулей ........................................... 672  
Расширение as для инструкций import и from ......................... 673  
Модули – это объекты: метапрограммы .................................. 674  
Импортирование модулей по имени в виде строки ................... 677  
Транзитивная перезагрузка модулей ..................................... 678  
Концепции проектирования модулей ..................................... 681  
Типичные проблемы при работе с модулями ........................... 682  
В заключение ..................................................................... 688  
Закрепление пройденного .................................................... 688  
Часть VI. Классы и ООП ........................................................... 693  
Глава 25. ООП: общая картина ............................................... 695  
Зачем нужны классы? ......................................................... 696  
ООП с высоты 30 000 футов .................................................. 697  
В заключение ..................................................................... 706  
Закрепление пройденного .................................................... 707  
Глава 26. Основы программирования классов ........................ 709  
Классы генерируют множество экземпляров объектов ............. 709  
Классы адаптируются посредством наследования .................... 713  
Классы могут переопределять операторы языка Python ............ 717  
Самый простой в мире класс на языке Python ......................... 721  
Классы и словари ............................................................... 723  
В заключение ..................................................................... 725  
Закрепление пройденного .................................................... 726  
Глава 27. Более реалистичный пример ................................... 728  
Шаг 1: создание экземпляров ............................................... 729  
Шаг 2: добавление методов, определяющих поведение ............. 733  
Шаг 3: перегрузка операторов ............................................... 737  
Шаг 4: адаптация поведения с помощью подклассов ................ 739  
Шаг 5: адаптация конструкторов .......................................... 745  
Шаг 6: использование инструментов интроспекции ................. 750  
Шаг 7 (последний): сохранение объектов в базе данных ............ 757  
Рекомендации на будущее .................................................... 763  
В заключение ..................................................................... 765  
Закрепление пройденного .................................................... 766  
Глава 28. Подробнее о программировании классов ................ 769  
Инструкция class ................................................................ 769  
Методы ............................................................................. 772  
Наследование ..................................................................... 775

Оглавление   
13  
Пространства имен: окончание истории ................................ 781  
Еще раз о строках документирования .................................... 790  
Классы и модули ................................................................ 791  
В заключение ..................................................................... 792  
Закрепление пройденного .................................................... 792  
Глава 29. Перегрузка операторов ........................................... 794  
Доступ к элементам по индексу и извлечение срезов:   
\_\_getitem\_\_ и \_\_setitem\_\_ ................................................... 797  
Итерации по индексам: \_\_getitem\_\_ ...................................... 800  
Итераторы: \_\_iter\_\_ и \_\_next\_\_ ............................................ 802  
Проверка на вхождение:   
\_\_contains\_\_, \_\_iter\_\_ и \_\_getitem\_\_ ..................................... 807  
Обращения к атрибутам: \_\_getattr\_\_ и \_\_setattr\_\_ .................. 809  
Строковое представление объектов: \_\_repr\_\_ и \_\_str\_\_ ............ 812  
Правостороннее сложение и операция приращения:   
 \_\_radd\_\_ и \_\_iadd\_\_ ........................................................... 814  
Операция вызова: \_\_call\_\_ ................................................... 816  
Функциональные интерфейсы   
и программный код обратного вызова .................................... 818  
Сравнивание: \_\_lt\_\_, \_\_gt\_\_ и другие .................................... 820  
Проверка логического значения: \_\_bool\_\_ и \_\_len\_\_ ................ 821  
В заключение ..................................................................... 825  
Закрепление пройденного .................................................... 826  
Глава 30. Шаблоны проектирования с классами ..................... 828  
Python и ООП ..................................................................... 828  
ООП и наследование:   
взаимосвязи типа «является» ............................................... 830  
ООП и композиция: взаимосвязи типа «имеет» ....................... 832  
ООП и делегирование: объекты-обертки ................................. 837  
Псевдочастные атрибуты класса ........................................... 839  
Методы – это объекты:   
связанные и несвязанные методы .......................................... 842  
Множественное наследование: примесные классы ................... 849  
Классы – это объекты:   
универсальные фабрики объектов ......................................... 861  
Прочие темы, связанные с проектированием .......................... 863  
В заключение ..................................................................... 863  
Закрепление пройденного .................................................... 864  
Глава 31. Дополнительные возможности классов .................... 865  
Расширение встроенных типов ............................................. 866  
Классы «нового стиля» ........................................................ 869  
Изменения в классах нового стиля ........................................ 870

14   
Оглавление  
Другие расширения в классах нового стиля ............................ 880  
Статические методы и методы класса ..................................... 887  
Декораторы и метаклассы: часть 1 ........................................ 896  
Типичные проблемы при работе с классами ............................ 901  
В заключение ..................................................................... 907  
Закрепление пройденного .................................................... 908  
Часть VII. Исключения и инструменты ................................... 917  
Глава 32. Основы исключений ............................................... 919  
Зачем нужны исключения? .................................................. 920  
Обработка исключений: краткий обзор .................................. 921  
В заключение ..................................................................... 927  
Закрепление пройденного .................................................... 928  
Глава 33. Особенности использования исключений ................ 929  
Инструкция try/except/else.................................................. 929  
Инструкция try/finally ........................................................ 936  
Объединенная инструкция try/except/finally ......................... 939  
Инструкция raise ................................................................ 943  
Инструкция assert .............................................................. 946  
Контекстные менеджеры with/as .......................................... 948  
В заключение ..................................................................... 952  
Закрепление пройденного .................................................... 952  
Глава 34. Объекты исключений .............................................. 954  
Исключения: назад в будущее............................................... 955  
Исключения на основе классов ............................................. 956  
В чем преимущества иерархий исключений? .......................... 959  
Классы встроенных исключений ........................................... 962  
Определение текста исключения ........................................... 965  
Передача данных в экземплярах и реализация поведения ........ 966  
В заключение ..................................................................... 968  
Закрепление пройденного .................................................... 969  
Глава 35. Использование исключений .................................... 971  
Вложенные обработчики исключений .................................... 971  
Идиомы исключений ........................................................... 975  
Советы по применению   
и типичные проблемы исключений ....................................... 980  
Заключение по основам языка .............................................. 984  
В заключение ..................................................................... 990  
Закрепление пройденного .................................................... 991

Оглавление   
15  
Часть VIII. Расширенные возможности .................................. 993  
Глава 36. Юникод и строки байтов ......................................... 995  
Изменения в Python 3.0, касающиеся строк ............................ 996  
Основы строк ..................................................................... 997  
Примеры использования строк в Python 3.0.......................... 1003  
Кодирование строк Юникода .............................................. 1006  
Использование объектов bytes в Python 3.0 ........................... 1015  
Использование объектов bytearray в 3.0 (и 2.6) ..................... 1018  
Использование текстовых и двоичных файлов ...................... 1021  
Использование файлов Юникода ......................................... 1026  
Другие инструменты для работы со строками в Python 3.0 ...... 1031  
В заключение ................................................................... 1039  
Закрепление пройденного .................................................. 1040  
Глава 37. Управляемые атрибуты ........................................ 1043  
Зачем нужно управлять атрибутами?................................... 1043  
Свойства .......................................................................... 1045  
Дескрипторы .................................................................... 1050  
\_\_getattr\_\_ и \_\_getattribute\_\_ ............................................ 1059  
Пример: проверка атрибутов .............................................. 1078  
В заключение ................................................................... 1084  
Закрепление пройденного .................................................. 1084  
Глава 38. Декораторы .......................................................... 1087  
Что такое декоратор?......................................................... 1087  
Основы ............................................................................ 1090  
Программирование декораторов функций ............................ 1100  
Программирование декораторов классов .............................. 1116  
Непосредственное управление функциями и классами ........... 1127  
Пример: «частные» и «общедоступные» атрибуты ................. 1130  
Пример: проверка аргументов функций ............................... 1142  
В заключение ................................................................... 1155  
Закрепление пройденного .................................................. 1156  
Глава 39. Метаклассы .......................................................... 1160  
Нужны или не нужны метаклассы ...................................... 1161  
Модель метаклассов .......................................................... 1168  
Объявление метаклассов .................................................... 1172  
Программирование метаклассов ......................................... 1173  
Пример: добавление методов в классы ................................. 1179  
Пример: применение декораторов к методам ........................ 1186  
В заключение ................................................................... 1194  
Закрепление пройденного .................................................. 1195

16   
Оглавление  
Часть IX. Приложения ........................................................... 1197  
Приложение A. Установка и настройка ................................. 1199  
Установка интерпретатора Python ....................................... 1199  
Настройка Python ............................................................. 1203  
Параметры командной строки интерпретатора ...................... 1208  
Дополнительная информация ............................................. 1209  
Приложение B. Решения упражнений .................................. 1211  
Часть I. Введение .............................................................. 1211  
Часть II. Типы и операции ................................................. 1214  
Часть III. Инструкции и синтаксис ...................................... 1219  
Часть IV. Функции............................................................ 1221  
Часть V. Модули ............................................................... 1229  
Часть VI. Классы и ООП ..................................................... 1233  
Часть VII. Исключения и инструменты ................................ 1241  
Алфавитный указатель ....................................................... 1249

Предисловие  
Эта книга представляет собой введение в Python – популярный язык програм-  
Python – популярный язык програм-  
 – популярный язык програм-  
мирования, используемый как для разработки самостоятельных программ,   
так и для создания прикладных сценариев в самых разных областях приме-  
нения. Это мощный, переносимый, простой в использовании и свободно рас-  
пространяемый язык. Программисты, работающие в самых разных областях,   
считают, что ориентация Python на эффективность разработки и высокое ка-  
чество программного обеспечения дает ему стратегическое преимущество как   
в маленьких, так и в крупных проектах.  
Цель этой книги – помочь вам быстро овладеть основными принципами Py-  
Py-  
thon независимо от уровня вашей подготовки. Прочитав эту книгу, вы получи-  
 независимо от уровня вашей подготовки. Прочитав эту книгу, вы получи-  
те объем знаний, достаточный для использования этого языка.  
Издание задумывалось как учебник, основное внимание в котором уделяется   
ядру языка программирования Python, а не прикладным аспектам его исполь-  
зования. Вообще, эта книга должна рассматриваться как первая из следующе-  
го цикла:  
 •  
«Изучаем Python» – эта книга служит учебником по языку Python.  
 •  
«Программирование на Python»1, где помимо всего прочего показаны воз-  
можности применения языка Python после того, как он был освоен.  
То есть издания, посвященные прикладным аспектам, такие как «Програм-  
мирование на Python», начинаются с того места, где заканчивается эта книга,   
и исследуют применение Python в различных прикладных областях, таких   
как веб-приложения, графические интерфейсы пользователя (ГИП) и прило-  
жения баз данных. Кроме того, в книге «Python Pocket Reference», которая за-  
Python Pocket Reference», которая за-  
 Pocket Reference», которая за-  
Pocket Reference», которая за-  
 Reference», которая за-  
Reference», которая за-  
», которая за-  
думывалась как дополнение к этой книге, вы найдете дополнительные спра-  
вочные материалы, не вошедшие в эту книгу.  
Благодаря такой направленности в этой книге стало возможным представить   
основы языка Python более глубоко, чем во многих других пособиях для начи-  
Python более глубоко, чем во многих других пособиях для начи-  
 более глубоко, чем во многих других пособиях для начи-  
нающих. Книга основана на материалах практических курсов, включает в себя   
контрольные вопросы и самостоятельные упражнения и поэтому может слу-  
жить введением в язык, рассчитанным на индивидуальную скорость освоения.  
О четвертом издании  
Четвертое издание книги претерпело три основных изменения:  
1   
Лутц М. «Программирование на Python», 2-е изд. – Пер. с англ. – СПб.: Символ-  
Плюс, 2002. Четвертое издание этой книги выйдет в 2011 году.

18   
Предисловие  
 •  
Охватываются обе версии, Python 3.0 и Python 2.6, с особым вниманием   
к версии 3.0, но при этом отмечаются отличия, имеющиеся в версии 2.6.  
 •  
Добавлено несколько новых глав, в основном посвященных развитию базо-  
вых возможностей языка.  
 •  
Проведена реорганизация части имевшегося материала и добавлены новые   
примеры для большей ясности.  
Когда я работал над этим изданием в 2009 году, существовало две основных   
версии интерпретатора Python – версия 3.0, представляющая новую ступень   
развития языка и несовместимая с прежним программным кодом, и вер-  
сия 2.6, сохранившая обратную совместимость с огромным количеством су-  
ществующих программ на языке Python. Хотя Python 3 рассматривается как   
будущее языка Python, тем не менее Python 2 по-прежнему широко использу-  
Python, тем не менее Python 2 по-прежнему широко использу-  
, тем не менее Python 2 по-прежнему широко использу-  
Python 2 по-прежнему широко использу-  
 2 по-прежнему широко использу-  
ется и будет поддерживаться параллельно с Python 3 еще на протяжении не-  
Python 3 еще на протяжении не-  
 3 еще на протяжении не-  
скольких лет. Версия 3.0 – это в значительной степени тот же язык программи-  
рования, однако она практически несовместима с программным кодом, напи-  
санным для прежних версий интерпретатора (тот факт, что инструкция print   
превратилась в функцию, только на первый взгляд кажется косметическим   
изменением, однако именно это обстоятельство сделало неработоспособными   
почти все программы на языке Python, написанные ранее).  
Наличие двух параллельно существующих версий представляет дилемму –   
как для программистов, так и для авторов книг. Проще всего было бы сделать   
вид, что версия Python 2 никогда не существовала, и сосредоточить все внима-  
Python 2 никогда не существовала, и сосредоточить все внима-  
 2 никогда не существовала, и сосредоточить все внима-  
ние только на версии 3, но это не будет отвечать потребностям большого коли-  
чества пользователей языка Python. В настоящее время существует огромное   
количество программ, написанных для версии Python 2, и в ближайшее время   
они никуда не денутся. Начинающие программисты могут сосредоточиться на   
версии Python 3, но все, кому приходится сопровождать программы, написан-  
Python 3, но все, кому приходится сопровождать программы, написан-  
 3, но все, кому приходится сопровождать программы, написан-  
ные ранее, вынуждены одной ногой оставаться в Python 2. Для переноса всех   
сторонних библиотек и расширений на версию Python 3 могут потребоваться   
годы, поэтому такое раздвоение не будет преодолено в ближайшее время.  
Охват обеих версий, 3.0 и 2.6  
Чтобы учесть это раздвоение и удовлетворить нужды всех потенциальных чи-  
тателей, в этом издании рассматриваются обе версии, Python 3.0 и Python 2.6   
(а также более поздние версии веток 3.X и 2.X). Эта книга предназначена для   
программистов, использующих Python 2, Python 3, а также для тех, кто поль-  
Python 2, Python 3, а также для тех, кто поль-  
 2, Python 3, а также для тех, кто поль-  
Python 3, а также для тех, кто поль-  
 3, а также для тех, кто поль-  
зуется обеими версиями, т. е. с помощью этой книги вы сможете изучить лю-  
бую из основных версий языка Python.   
Основное внимание уделяется версии 3.0, тем не менее по ходу повествования   
будут отмечаться отличия, существующие в версии 2.6, и инструменты для   
тех, кому приходится сопровождать старые программы. Синтаксис языка про-  
граммирования почти не изменился, однако имеется несколько важных отли-  
чий, о которых я также буду рассказывать по ходу изложения материала.  
Например, в большинстве примеров используется вызов функции print в стиле   
версии 3.0, но при этом будет описываться использование инструкции print,   
присутствующей в версии 2.6, чтобы вы могли понимать программный код,   
написанный ранее. Кроме того, я представлю новые особенности языка, такие   
как инструкция nonlocal, появившаяся в версии 3.0, и метод форматирования

Предисловие   
19  
строк format в версиях 2.6 и 3.0, а также укажу, какие расширения отсутству-  
ют в более старых версиях Python.  
Если вы только приступаете к изучению языка Python и не стоите перед необ-  
Python и не стоите перед необ-  
 и не стоите перед необ-  
ходимостью использовать устаревший программный код, я рекомендую начи-  
нать сразу с версии Python 3.0� в этой версии были устранены многие давниш-  
Python 3.0� в этой версии были устранены многие давниш-  
 3.0� в этой версии были устранены многие давниш-  
ние недостатки языка, при этом было сохранено все самое лучшее и добавлены   
некоторые новые возможности.  
К тому времени, когда вы будете читать эти строки, многие популярные биб-  
лиотеки и инструменты наверняка станут доступны и в версии Python 3.0,   
особенно если учесть улучшение производительности операций ввода-вывода,   
которое заявлено в ожидающейся версии 3.1. Если вы используете програм-  
мы, основанные на версии Python 2.X, вы обнаружите, что эта книга отвечает   
вашим потребностям в настоящее время, а в дальнейшем поможет вам выпол-  
нить переход на версию 3.0.  
По большей части это издание книги может использоваться для изучения язы-  
ка Python при использовании разных выпусков версий Python 2 и 3, однако   
некоторые устаревшие версии интерпретатора из ветки 2.X могут оказаться не   
в состоянии выполнить программный код всех примеров, которые приводятся   
здесь. Например, декораторы классов могут использоваться в обеих версиях   
Python 2.6 и 3.0, но вы не сможете применять их в более старых версиях Py-  
 2.6 и 3.0, но вы не сможете применять их в более старых версиях Py-  
Py-  
thon 2.X, где эта особенность языка отсутствует. Ниже, в табл. П.1 и П.2, при-  
 2.X, где эта особенность языка отсутствует. Ниже, в табл. П.1 и П.2, при-  
X, где эта особенность языка отсутствует. Ниже, в табл. П.1 и П.2, при-  
, где эта особенность языка отсутствует. Ниже, в табл. П.1 и П.2, при-  
водятся основные отличия между версиями 2.6 и 3.0.  
Незадолго до передачи книги в печать она была дополнена при-  
мечаниями о наиболее существенных расширениях языка, по-  
явившихся в версии Python 3.1: использование запятых в каче-  
стве разделителей и автоматическая нумерация полей в вызовах   
строкового метода format, синтаксис множественных менедже-  
ров контекста в инструкциях with, новые методы чисел и так да-  
лее. Основная цель версии Python 3.1 – оптимизация скорости   
выполнения, поэтому данная книга также охватывает и эту но-  
вую версию. Фактически версия Python 3.1 выпускается как за-  
мена версии 3.0. Кроме того, последняя версия Python обычно   
всегда самая лучшая. Поэтому в этой книге термин «Python 3.0»   
практически везде используется для обозначения не конкретной   
версии, а изменений в языке, введенных в версии Python 3.0   
и присутствующих во всех версиях ветки 3.X.  
Новые главы  
Основная цель этого издания – обновить примеры и сведения, приводившие-  
ся в предыдущем издании и касающиеся версий Python 3.0 и 2.6. Кроме того,   
я добавил пять новых глав, описывающих новые особенности:  
 •  
Глава 27 – новое руководство по классам, где приведен более реалистичный   
пример, демонстрирующий основы объектно-ориентированного програм-  
мирования (ООП) на языке Python.  
 •  
Глава 36 подробно рассказывает о строках Юникода и строках байтов,   
а также описывает отличия строк и файлов в версиях 3.0 и 2.6.

20   
Предисловие  
 •  
Глава 37 описывает средства управления атрибутами, такие как свойства,   
и содержит новые сведения о дескрипторах.  
 •  
Глава 38 представляет декораторы функций и классов и содержит подроб-  
ные и исчерпывающие примеры.  
 •  
Глава 39 охватывает метаклассы и сравнивает их с декораторами.  
Первая из этих глав представляет собой подробное пошаговое руководство по   
использованию классов и приемов ООП в языке Python. Она основана на при-  
Python. Она основана на при-  
. Она основана на при-  
мерах, используемых на учебных курсах и адаптированных для книги. Эта   
глава призвана продемонстрировать применение ООП в более приближенном   
к реальности контексте, чем в предыдущих изданиях, и показать, как может   
использоваться концепция классов для построения крупных программ. Я на-  
деюсь, что эти примеры будут полезны настолько же, насколько они полезны   
на учебных курсах.  
Последние четыре главы в этом списке составляют заключительную часть   
книги «Расширенные возможности». С технической точки зрения эти темы от-  
носятся к основам языка программирования, тем не менее многие программи-  
сты используют простые строки символов ASCII и не нуждаются в детальном   
изучении строк Юникода и строк с двоичными данными. Точно так же деко-  
раторы и метаклассы являются узкоспециализированными темами, представ-  
ляющими интерес скорее для разработчиков программных интерфейсов, чем   
для прикладных программистов. Поэтому эти четыре главы были выделены   
в отдельную часть и не являются обязательными для чтения.  
Если же вы используете эти возможности или сопровождаете программы, в ко-  
торых они применяются, эти новые главы должны помочь вам освоить их. Кро-  
ме того, примеры в этих главах соединяют разнообразные базовые концепции   
языка и имеют большое практическое значение. В конце каждой главы име-  
ются контрольные вопросы, но отсутствуют упражнения, так как эта часть не   
является обязательной для прочтения.  
Изменения в существующем материале  
Помимо всего прочего была проведена реорганизация материала предыдуще-  
го издания, добавлены новые примеры. Так, в главе 30 в описание механизма   
множественного наследования был включен новый пример вывода деревьев   
классов� в главу 20 были добавлены новые примеры использования генерато-  
ров для реализации собственных версий функций map и zip� в главу 31 были   
включены новые примеры, иллюстрирующие статические методы и методы   
классов� в главе 23 демонстрируется операция импортирования относительно   
пакета, а в главу 29 были добавлены примеры, иллюстрирующие методы \_\_  
contains\_\_, \_\_bool\_\_ и \_\_index\_\_ перегрузки операторов и использование нового   
протокола перегрузки для операций извлечения среза и сравнения.  
Дополнительно была проведена реорганизация материала с целью добиться бо-  
лее последовательного изложения. Например, чтобы добавить новые сведения   
и избежать перегруженности, каждая из пяти вышеупомянутых глав была   
разбита на две главы. В результате появились новые самостоятельные главы,   
посвященные перегрузке операторов, областям видимости и аргументам, осо-  
бенностям обработки исключений, а также генераторам и итераторам. Был   
немного изменен порядок следования существующих глав, чтобы обеспечить   
более последовательное освещение тем.

Предисловие   
21  
В этом издании также была предпринята попытка ликвидировать ссылки на   
будущие темы за счет переупорядочения глав, однако в некоторых случаях это   
оказалось невозможным из-за изменений в Python 3.0: для полного понима-  
Python 3.0: для полного понима-  
 3.0: для полного понима-  
ния особенностей вывода строк и метода format вам потребуется знакомство   
с именованными аргументами функций� чтобы понять, как получить список   
ключей словаря и как проверить наличие того или иного ключа, необходимо   
знакомство с итераторами� чтобы научиться использовать функцию exec для   
выполнения программного кода, необходимо уметь применять объекты фай-  
лов, и так далее. Читать книгу по-прежнему предпочтительнее по порядку, од-  
нако иногда может потребоваться забежать вперед.  
В общей сложности в этом издании было выполнено несколько сотен измене-  
ний. Одни только таблицы в следующем разделе описывают 27 дополнений   
и 57 изменений в языке Python. Справедливости ради следует отметить, что   
это издание стало более продвинутым, потому что сам язык Python стал более   
продвинутым.  
Расширения языка, появившиеся   
в версиях 2.6 и 3.0  
Вообще говоря, Python версии 3.0 стал более цельным, но он также стал в не-  
котором роде и более сложным. На первый взгляд некоторые из изменений   
предполагают, что для изучения языка Python вы уже должны быть знакомы   
с ним! В предыдущем разделе уже были упомянуты некоторые темы, которые   
связаны круговой зависимостью в версии 3.0, например объяснение темы пред-  
ставлений словарей практически невозможно без наличия предварительных   
знаний. Помимо обучения основам языка Python эта книга послужит подспо-  
Python эта книга послужит подспо-  
 эта книга послужит подспо-  
рьем в заполнении подобных пробелов в ваших представлениях.  
В табл. П.1 перечислены наиболее заметные новые особенности языка и номера   
глав, в которых они описываются.  
Таблица П.1. Нововведения в версиях Python 2.6 и 3.0  
Нововведение  
См. главы  
Функция print в версии 3.0  
11  
Инструкция nonlocal в версии 3.0  
17  
Метод str.format в версиях 2.6 и 3.0  
7  
Строковые типы в версии 3.0: str – для представления строк   
Юникода, bytes – для представления двоичных данных  
7, 36  
Отличия между текстовыми и двоичными файлами в вер-  
сии 3.0  
9, 36  
Декораторы классов в версиях 2.6 и 3.0: @private(‘age’)  
31, 38  
Новые итераторы в версии 3.0: range, map, zip  
14, 20  
Представления словарей в версии 3.0: D.keys, D.values, D.items  
8, 14  
Операторы деления в версии 3.0: остаток, / и //  
5

22   
Предисловие  
Нововведение  
См. главы  
Литералы множеств в версии 3.0: {a, b, c}  
5  
Генераторы множеств в версии 3.0: {x\*\*2 for x in seq}  
4, 5, 14, 20  
Генераторы словарей в версии 3.0: {x: x\*\*2 for x in seq}  
4, 5, 14, 20  
Поддержка представления двоичных чисел в виде строк в вер-  
сиях 2.6 и 3.0: 0b0101, bin(I)  
5  
Тип представления рациональных чисел в версиях 2.6 и 3.0:   
Fraction(1, 3)  
5  
Аннотации функций в версии 3.0: def f(a:99, b:str)->int  
19  
Аргументы, которые могут быть только именованными в вер-  
сии 3.0: def f(a, \*b, c, \*\*d)  
18, 20  
Расширенная операция распаковывания последовательностей   
в версии 3.0: a, \*b = seq  
11, 13  
Синтаксис импортирования относительно пакета в версии 3.0:   
from .  
23  
Менеджеры контекста в версиях 2.6 и 3.0: with/as  
33, 35  
Синтаксис исключений в версии 3.0: raise, except/as, супер-  
классы  
33, 34  
Объединение исключений в цепочки в версии 3.0:   
raise e2 from e1  
33  
Изменения в наборе зарезервированных слов в версиях 2.6   
и 3.0  
11  
Переход на использование классов «нового стиля» в версии 3.0  
31  
Декораторы свойств в версиях 2.6 и 3.0: @property  
37  
Дескрипторы в версиях 2.6 и 3.0  
31, 38  
Метаклассы в версиях 2.6 и 3.0  
31, 39  
Поддержка абстрактных базовых классов в версиях 2.6 и 3.0  
28  
Особенности языка, удаленные в версии 3.0  
Помимо дополнений в версии 3.0 некоторые особенности языка были удалены   
с целью сделать его синтаксис более стройным. В табл. П.2 перечислены изме-  
нения, которые были учтены в различных главах этого издания. Для многих   
особенностей, перечисленных в табл. П.2, имеются прямые замены, часть из   
которых также доступна в версии 2.6, чтобы обеспечить поддержку перехода   
на версию 3.0.  
Таблица П.1 (продолжение)

Предисловие   
23  
Таблица П.2. Особенности языка, удаленные в версии Python 3.0, которые   
были учтены в этой книге  
Удаленная особенность  
Замена  
См. главы  
reload(M)  
imp.reload(M) (или exec)  
3, 22  
apply(f, ps, ks)  
f(\*ps, \*\*ks)  
18  
`X`  
repr(X)  
5  
X <> Y  
X != Y  
5  
long  
int  
5  
9999L  
9999  
5  
D.has\_key(K)  
K in D (или D.get(key) != None)  
8  
raw\_input  
input  
3, 10  
прежняя версия input  
eval(input)  
3  
xrange  
range  
14  
file  
open (и классы из модуля io)  
9  
X.next  
X.\_\_next\_\_,   
вызывается функцией next(X)  
14, 20, 29  
X.\_\_getslice\_\_  
Методу X.\_\_getitem\_\_ передается объ-  
ект среза  
7, 29  
X.\_\_setslice\_\_  
Методу X.\_\_setitem\_\_ передается объ-  
ект среза  
7, 29  
reduce  
functools.reduce   
(или реализация цикла вручную)  
14, 19  
execfile(filename)  
exec(open(filename).read())  
3  
exec open(filename)  
exec(open(filename).read())  
3  
0777  
0o777  
5  
print x, y  
print(x, y)  
11  
print >> F, x, y  
print(x, y, file=F)  
11  
print x, y,  
print(x, y, end=’ ‘)  
11  
u’ccc’  
‘ccc’  
7, 36  
‘bbb’ для строк байтов  
b’bbb’  
7, 9, 36  
raise E, V  
raise E(V)  
32, 33, 34  
except E, X:  
except E as X:  
32, 33, 34  
def f((a, b)):  
def f(x): (a, b) = x  
11, 18, 20  
file.xreadlines  
for line in file: (или X=iter(file))  
13, 14  
D.keys и др.,   
результат в виде списка  
list(D.keys) (представления словарей) 8, 14

24   
Предисловие  
Удаленная особенность  
Замена  
См. главы  
map, range и др.,   
результат в виде списка  
list(map()), list(range())   
(встроенная функция)  
14  
map(None, ...)  
zip (или дополнение результатов   
вручную)  
13, 20  
X=D.keys(); X.sort()  
sorted(D) (или list(D.keys()))  
4, 8, 14  
cmp(x, y)  
(x > y) - (x < y)  
29  
X.\_\_cmp\_\_(y)  
\_\_lt\_\_, \_\_gt\_\_, \_\_eq\_\_ и т. д.  
29  
X.\_\_nonzero\_\_  
X.\_\_bool\_\_  
29  
X.\_\_hex\_\_, X.\_\_oct\_\_  
X.\_\_index\_\_  
29  
Различные функции   
сортировки  
Используются аргументы   
key=transform и reverse=True  
8  
Операции над словарями   
<, >, <=, >=  
Сравнивание sorted(D.items())   
(или реализация цикла вручную)  
8, 9  
types.ListType  
list (только для невстроенных типов)  
9  
\_\_metaclass\_\_ = M  
class C(metaclass=M):  
28, 31, 39  
\_\_builtin\_\_  
builtins (переименован)  
17  
Tkinter  
tkinter (переименован)  
18, 19, 24,   
29, 30  
sys.exc\_type, exc\_value  
sys.exc\_info()[0], [1]  
34, 35  
function.func\_code  
function.\_\_code\_\_  
19, 38  
\_\_getattr\_\_ вызывается   
встроенными функциями  
Переопределение методов \_\_X\_\_   
в классе-обертке  
30, 37, 38  
-t, –tt ключи командной   
строки  
Чередование использования символов   
табуляции и пробелов всегда приво-  
дит к ошибке  
10,12  
from ... \*, внутри функ-  
ции  
Может использоваться только   
на верхнем уровне в модуле  
22  
import mod, из модуля   
в том же пакете  
from . import mod, импортирование от-  
носительно пакета  
23  
class MyException:  
class MyException(Exception):  
34  
Модуль exceptions   
Встроенная область видимости, спра-  
вочное руководство по библиотеке  
34  
Модули thread, Queue  
\_thread, queue (переименованы)  
17  
Модуль anydbm   
dbm (переименован)  
27  
Модуль cPickle   
\_pickle (переименован,   
используется автоматически)  
9  
Таблица П.2 (продолжение)

Предисловие   
25  
Удаленная особенность  
Замена  
См. главы  
os.popen2/3/4  
subprocess.Popen (os.popen   
по-прежнему поддерживается)  
14  
Исключения на базе   
строк  
Исключения на базе классов (обяза-  
тельны в версии 2.6)  
32, 33, 34  
Модуль с функциями для   
работы со строками  
Методы объектов строк  
7  
Несвязанные методы  
Функции (staticmethod – для вызова   
относительно экземпляра)  
30, 31  
Смешивание несовмести-  
мых типов в операциях   
сравнения и сортировки  
Смешивание несовместимых типов   
в операциях сравнения вызывает по-  
явление ошибки  
5, 9  
В Python 3.0 имеются изменения, которые не были включены в эту таблицу,   
т. к. они не относятся к кругу тем этой книги. Например, изменения в стан-  
дартной библиотеке представляют больший интерес для книг, освещающих   
прикладные аспекты применения языка, таких как «Программирование на   
Python», чем для этой книги. Хотя стандартная библиотека в значительной   
степени поддерживает прежнюю функциональность, в Python 3.0 некоторые   
модули были переименованы, сгруппированы в пакеты и так далее. Более   
полный список изменений, внесенных в версию 3.0, можно найти в документе   
«�hat’s New in Python 3.0» (Что нового в Python 3.0), включенном в стандарт-  
�hat’s New in Python 3.0» (Что нового в Python 3.0), включенном в стандарт-  
’s New in Python 3.0» (Что нового в Python 3.0), включенном в стандарт-  
s New in Python 3.0» (Что нового в Python 3.0), включенном в стандарт-  
 New in Python 3.0» (Что нового в Python 3.0), включенном в стандарт-  
New in Python 3.0» (Что нового в Python 3.0), включенном в стандарт-  
 in Python 3.0» (Что нового в Python 3.0), включенном в стандарт-  
in Python 3.0» (Что нового в Python 3.0), включенном в стандарт-  
 Python 3.0» (Что нового в Python 3.0), включенном в стандарт-  
Python 3.0» (Что нового в Python 3.0), включенном в стандарт-  
 3.0» (Что нового в Python 3.0), включенном в стандарт-  
Python 3.0), включенном в стандарт-  
 3.0), включенном в стандарт-  
ный набор справочных руководств.1  
При переходе с версии Python 2.X на версию Python 3.X обязательно ознакомь-  
Python 2.X на версию Python 3.X обязательно ознакомь-  
 2.X на версию Python 3.X обязательно ознакомь-  
X на версию Python 3.X обязательно ознакомь-  
 на версию Python 3.X обязательно ознакомь-  
Python 3.X обязательно ознакомь-  
 3.X обязательно ознакомь-  
X обязательно ознакомь-  
 обязательно ознакомь-  
тесь со сценарием 2to3 автоматического переноса программного кода, который   
входит в состав дистрибутива Python 3.0. Конечно, он не гарантирует выпол-  
Python 3.0. Конечно, он не гарантирует выпол-  
 3.0. Конечно, он не гарантирует выпол-  
нение переноса любой программы, но способен преобразовать большую часть   
программного кода, написанного для Python 2.X, так, что он будет выполнять-  
Python 2.X, так, что он будет выполнять-  
 2.X, так, что он будет выполнять-  
X, так, что он будет выполнять-  
 так, что он будет выполнять-  
ся под управлением Python 3.X. К моменту написания этих строк на стадии   
реа лизации находился проект сценария 3to2 для обратного преобразования   
программного кода, написанного для Python 3.X, так, чтобы он мог выпол-  
Python 3.X, так, чтобы он мог выпол-  
 3.X, так, чтобы он мог выпол-  
X, так, чтобы он мог выпол-  
, так, чтобы он мог выпол-  
няться в среде Python 2.X. Любой из этих инструментов может оказаться по-  
X. Любой из этих инструментов может оказаться по-  
. Любой из этих инструментов может оказаться по-  
лезным для тех, кому приходится сопровождать программы, которые должны   
выполняться под управлением обеих основных версий Python. Подробности об   
этих утилитах ищите в Интернете.  
О третьем издании  
Новая редакция отражает последние новшества, появившиеся в самом языке   
и в методиках его обучения. Помимо этого, была несколько изменена структу-  
ра книги.  
1   
Аналогичную информацию (хотя это и не прямой перевод указанного документа)   
на русском языке можно найти на странице http://www.ibm.com/developerworks/ru/  
library/l-python3-1/. – Примеч. перев.

26   
Предисловие  
Изменения в языке Python (для 3-го издания)  
Если говорить о версии языка, это издание описывает Python 2.5 и отражает   
все изменения, появившиеся в языке с момента выхода второго издания кни-  
ги в конце 2003 года. (Во втором издании описывался язык Python 2.2 и не-  
Python 2.2 и не-  
 2.2 и не-  
которые нововведения версии 2.3.) Кроме того, в этом издании обсуждаются   
изменения, которые ожидаются в версии Python 3.0. Ниже приводится список   
основных тем, касающихся языка программирования, которые вы найдете   
в этом издании (нумерация глав была изменена, чтобы соответствовать четвер-  
тому изданию):  
 •  
Новая условная конструкция B if A else C (глава 19)  
 •  
Оператор менеджера контекста with/as (глава 33)  
 •  
Унификация конструкции try/except/finally (глава 33)  
 •  
Синтаксис относительного импорта (глава 23)  
 •  
Выражения-генераторы (глава 20)  
 •  
Новые особенности функций-генераторов (глава 20)  
 •  
Функции-декораторы (глава 31)  
 •  
Объектный тип множества (глава 5)  
 •  
Новые встроенные функции: sorted, sum, any, all, enumerate (главы 13 и 14)  
 •  
Объектный тип десятичных чисел с фиксированной точностью представле-  
ния (глава 5)  
 •  
Файлы, генераторы списков и итераторы (главы 14 и 20)  
 •  
Новые инструменты разработки: Eclipse, dustutils, unittest и doctest, рас-  
ширения IDLE, Shedskin и так далее (главы 2 и 35)  
Менее значительные изменения в языке (такие как широко используемые зна-  
чения True и False, новая функция sys.exec\_info, которая возвращает подроб-  
ную информацию об исключении, и отказ от строковых исключений, методы   
строк и встроенные функции apply и reduce) обсуждаются на протяжении всей   
книги. Кроме того, здесь приводится расширенное описание некоторых новых   
особенностей, впервые появившихся во втором издании, в том числе третье из-  
мерение при работе со срезами и возможность передачи функциям произволь-  
ного числа аргументов, включая функцию apply.  
Изменения в обучении языку Python (для 3-го издания)  
Кроме уже перечисленных изменений в самом языке, третье издание дополне-  
но новыми темами и примерами, наработанными мною при преподавании на   
курсах обучения языку Python в последние годы. Например, здесь вы найдете   
(нумерация глав была изменена с целью соответствовать четвертому изданию):  
 •  
Новую главу о встроенных типах (глава 4)  
 •  
Новую главу о синтаксических конструкциях (глава 10)  
 •  
Полностью новую главу о динамической типизации с углубленным освеще-  
нием этого вопроса (глава 6)  
 •  
Расширенное введение в ООП (глава 25)  
 •  
Новые примеры работы с файлами, областями видимости, вложенными ин-  
струкциями, классами, исключениями и так далее

Предисловие   
27  
Множество изменений и дополнений было сделано, чтобы облегчить чтение   
книги начинающим программистам� в результате обсуждение некоторых тем   
было перемещено в более соответствующие для этого места с учетом опыта пре-  
подавания Python на курсах. Например, описание генераторов списков и ите-  
Python на курсах. Например, описание генераторов списков и ите-  
 на курсах. Например, описание генераторов списков и ите-  
раторов теперь приводится вместе с описанием оператора цикла for, а не с опи-  
санием функциональных инструментов, как это было ранее.  
Кроме того, в третьем издании было расширено описание основ языка, добав-  
лены новые темы и примеры. Поскольку эта книга фактически превратилась   
в стандартный учебник по языку Python, изложение материала стало более   
полным и расширено новыми примерами его использования.  
Был полностью обновлен комплект советов и рекомендаций, подобранных из   
опыта преподавания в течение 10 лет и практического использования Python   
в течение 15 лет. Также были дополнены и расширены учебные упражнения   
с целью отразить наиболее удачные современные приемы программирования   
на языке Python, его новые особенности и показать наиболее типичные ошиб-  
Python, его новые особенности и показать наиболее типичные ошиб-  
, его новые особенности и показать наиболее типичные ошиб-  
ки, которые совершают начинающие программисты на моих курсах. Вообще,   
основы языка в этом издании обсуждаются более широко, чем в предыдущих   
изданиях.  
Структурные изменения в 3-м издании  
Чтобы облегчить усвоение основ языка, весь материал поделен на несколько   
частей, каждая из которых содержит несколько глав. Например, типы и ин-  
струкции теперь описываются в двух разных частях, в каждой из которых   
основным типам и инструкциям отведены отдельные главы. При переработке   
материала упражнения и описания наиболее распространенных ошибок были   
перемещены из конца каждой главы в конец части.  
Каждая глава завершается кратким обзором и контрольными вопросами, ко-  
торые помогут проверить, насколько хорошо понят изложенный материал.   
В отличие от упражнений в конце каждой части, решения для которых приво-  
дятся в приложении B, ответы на вопросы в конце каждой главы следуют не-  
B, ответы на вопросы в конце каждой главы следуют не-  
, ответы на вопросы в конце каждой главы следуют не-  
посредственно за вопросами. Я рекомендую просматривать ответы, даже если   
вы уверены, что правильно ответили, потому что эти ответы, кроме прочего,   
являются кратким обзором только что пройденной темы.  
Несмотря на наличие новых тем, эта книга по-прежнему ориентирована на тех,   
кто только начинает знакомство с языком Python. Поскольку третье издание   
в значительной степени основано на проверенном временем опыте преподава-  
ния, оно, как и первые два, может служить вводным курсом для самостоятель-  
ного изучения языка Python.  
Ограничение области применения книги  
Третье издание представляет собой учебник по основам языка программирова-  
ния Python и ничего больше. Здесь приводятся всесторонние сведения о язы-  
Python и ничего больше. Здесь приводятся всесторонние сведения о язы-  
 и ничего больше. Здесь приводятся всесторонние сведения о язы-  
ке, которые необходимо знать, прежде чем приступать к практическому его   
использованию. Материал подается в порядке постепенного усложнения и дает   
полное представление о языке программирования, не фокусируясь на областях   
его применения.  
Для некоторых «изучить Python» означает потратить час-другой на изучение   
руководств в Интернете. Такой подход пригоден для опытных программи-

28   
Предисловие  
стов – в конце концов, Python – довольно простой язык по сравнению с другими   
языками программирования. Однако проблема такого ускоренного изучения   
состоит в том, что на практике программисты часто сталкиваются с необыч-  
ными случаями необъяснимого изменения значений переменных, параметров   
по умолчанию и так далее. Цель этой книги – дать твердое понимание основ   
языка Python, чтобы даже самые необычные случаи находили свое объяснение   
и чтобы вы смогли применять его для решения прикладных задач независимо   
от предметной области, в которой работаете.  
Это ограничение было введено намеренно. Ограничившись обсуждением основ   
языка, мы можем заняться более глубоким и полным их исследованием. Об-  
суждение темы прикладного использования Python и справочные материалы,   
не вошедшие в эту книгу, вы найдете в других публикациях, которые начина-  
ются с того места, где заканчивается эта книга.   
О книге «Изучаем Python»  
В этом разделе приводятся некоторые наиболее важные замечания об этой   
книге независимо от номера издания. Никакая книга не способна удовлетво-  
рить все нужды и потребности читателя, поэтому важно понимать основные   
цели книги.  
Предварительные условия  
В действительности книга не предъявляет никаких предварительных усло-  
вий. Она с успехом использовалась как начинающими программистами, так   
и умуд ренными опытом ветеранами. Если у вас есть желание изучать Python,   
эта книга наверняка вам поможет. Наличие у читателя некоторого опыта   
в программировании не является обязательным, но будет совсем не лишним.  
Эта книга задумывалась как введение в Python для программистов.1 Возмож-  
но, она не идеальна для тех, кто раньше никогда не имел дела с компьютерами   
(например, мы не будем объяснять, что такое компьютер), но я не делал ника-  
ких предположений о наличии у читателя опыта программирования или об   
уровне его подготовки.  
С другой стороны, я не считаю нужным обижать читателей, предполагая, что   
они «чайники», что бы это ни означало, – писать полезные программы на языке   
Python просто, и эта книга покажет, как это делается. В книге Python иногда   
противопоставляется другим языкам программирования, таким как C, C++,   
Java и Pascal, но эти сравнения можно просто игнорировать, если ранее вам не   
приходилось работать с этими языками.  
Сравнение с другими книгами  
Эта книга охватывает все основные аспекты языка программирования Python,   
но при этом я старался ограничить круг обсуждаемых тем, чтобы уменьшить   
1   
Под «программистами» я подразумеваю всех, кто написал хотя бы одну строчку кода   
на любом языке программирования. Если вы не входите в эту категорию, вы все рав-  
но сможете извлечь пользу из этой книги, но имейте в виду, что основное внимание   
в ней уделяется не основным принципам программирования, а изучению основ язы-  
ка программирования Python.

Предисловие   
29  
объем книги. Для сохранения простоты в ней рассматриваются самые основ-  
ные понятия, используются небольшие и очевидные примеры и опущены не-  
которые незначительные детали, которые вы найдете в справочных руковод-  
ствах. По этой причине данная книга должна рассматриваться как введение,   
как первый шаг к другим, более специализированным и более полным книгам.  
Например, мы не будем говорить об интеграции Python/C� это слишком слож-  
ная тема, которая, однако, является центральной для многих систем, основан-  
ных на применении Python. Мы также не будем говорить об истории развития   
Python и о процессе его разработки. А таких популярных применений Python,   
как создание графического интерфейса, разработка системных инструментов   
и работа с сетью, мы коснемся лишь очень кратко, если они вообще будут упо-  
минаться. Естественно, при таком подходе из поля зрения выпадает значитель-  
ная часть общей картины.  
Вообще говоря, Python стоит на более высоком качественном уровне относи-  
тельно других языков сценариев. Некоторые из его идей требуют более глубо-  
кого изучения, чем может вместить эта книга, поэтому я надеюсь, что боль-  
шинство читателей продолжит изучение принципов разработки приложений   
на этом языке, обратившись к другим источникам информации.  
Например, книга «Программирование на Python»1 (O’Reilly), содержащая бо-  
лее объемные и полные примеры наряду с описанием приемов прикладного   
программирования, задумывалась как продолжение книги «Изучаем Python».   
Эти две книги представляют собой две части курса обучения, который препо-  
дает автор, – основы языка и прикладное программирование. Кроме того, в ка-  
честве справочника можно использовать «Pyton Pocket Reference» (O’Reilly),   
где приводятся некоторые подробности, опущенные здесь.  
Для дальнейшего изучения можно порекомендовать книги, содержащие допол-  
нительные сведения, примеры или особенности использования языка Python   
в определенных прикладных областях, таких как веб-приложения и создание   
графических интерфейсов. Например, книги «Python in a Nutshell» (O’Relly)   
и «Python Essential Reference»2 (Sams) содержат справочную информацию.   
Книга «Python Cookbook» (O’Reilly) представляет собой сборник решений для   
тех, кто уже знаком с приемами прикладного программирования. Поскольку   
выбор книг является делом достаточно субъективным, я рекомендую вам са-  
мостоятельно поискать такие, которые наиболее полно будут отвечать вашим   
потребностям. Неважно, какие книги вы выберете, главное – помните, что для   
дальнейшего изучения Python вам необходимы более реалистичные примеры,   
чем приводятся здесь.  
На мой взгляд, данная книга будет для вас отличным учебником начального   
уровня, даже несмотря на ее ограниченность (и, скорее всего, именно поэтому).   
Здесь вы найдете все, что необходимо знать, прежде чем приступать к созда-  
нию программ и сценариев на языке Python. К тому моменту, когда вы закон-  
чите чтение этой книги, вы изучите не только сам язык, но и начнете пони-  
мать, как лучше применить его к решению ваших повседневных задач. Кроме   
1   
Лутц М. «Программирование на Python», 2-е изд. – Пер. с англ. – СПб.: Символ-  
Плюс, 2002. Четвертое издание этой книги выйдет в 2011 году.  
2   
Дэвид М. Бизли «Python. Подробный справочник». – Пер. с англ. – СПб.: Символ-  
Плюс, 2010.

30   
Предисловие  
того, у вас будет все необходимое для изучения более сложных тем и примеров,   
которые будут встречаться на вашем пути.  
Стиль и структура книги   
Эта книга основана на материалах практических курсов изучения языка   
Python. В конце каждой главы содержится список контрольных вопросов с от-  
ветами, а в конце последней главы каждой части – упражнения, примеры ре-  
шения которых приведены в приложении B. Контрольные вопросы подобраны   
так, что они представляют собой краткий обзор рассмотренного материала,   
а упражнения спроектированы так, чтобы сразу же научить вас правильно-  
му стилю программирования� кроме того, каждое упражнение соответствует   
одному из ключевых аспектов курса.  
Я настоятельно рекомендую прорабатывать контрольные вопросы и упражне-  
ния в ходе чтения книги не только для приобретения опыта программирования   
на Python, но и потому, что в упражнениях поднимаются проблемы, которые   
не обсуждаются нигде в книге. Ответы на вопросы в главах и примеры реше-  
ния упражнений в приложении B в случае необходимости помогут вам выйти   
из затруднительных положений.  
Общая структура книги также следует структуре учебного курса. Так как эта   
книга задумывалась как быстрое введение в основы языка программирования,   
изложение материала организовано так, чтобы оно отражало основные особен-  
ности языка, а не частности. Мы будем двигаться от простого к сложному: от   
встроенных типов объектов к инструкциям, элементам программ и так далее.   
Каждая глава является полным и самостоятельным описанием одной темы, но   
каждая последующая глава основана на понятиях, введенных в предыдущих   
главах (например, когда речь пойдет о классах, я буду исходить из предполо-  
жения, что вы уже знаете, как создаются функции), поэтому для большинства   
читателей имеет смысл читать книгу последовательно.  
Каждая часть посвящена отдельной крупной характеристике языка – типам,   
функциям и так далее. В большинстве своем примеры являются законченны-  
ми небольшими сценариями (некоторые из них являются достаточно искус-  
ственными, но они иллюстрируют достижение поставленной цели).   
Часть I «Введение»  
Изучение Python мы начнем с общего обзора этого языка и с ответов на оче-  
видно возникающие вопросы: почему кто-то использует этот язык, для ре-  
шения каких задач он может использоваться и так далее. В первой главе   
рассматриваются основные идеи, лежащие в основе технологии, которые   
должны дать вам некоторые начальные представления. Далее начинается   
сугубо технический материал книги. Здесь мы рассмотрим, как выполняют   
программы человек и интерпретатор Python. Цель этой части книги состо-  
ит в том, чтобы дать начальные сведения, которые позволят вам работать   
с последующими примерами и упражнениями.  
Часть II «Типы и операции»  
Далее мы приступим к исследованию языка программирования Python   
и начнем его изучение с основных встроенных типов объектов, таких как   
числа, списки, словари и так далее. Обладая только этими инструментами,   
вы уже сможете писать достаточно сложные программы. Это самая важная   
часть книги, потому что она закладывает основу для последующих глав.

Предисловие   
31  
В этой части мы также рассмотрим динамическую типизацию и ссылки –   
ключевые аспекты языка Python.  
Часть III «Инструкции и синтаксис»  
В этой части вводятся инструкции языка Python – программный код на   
языке Python, который создает и обслуживает объекты. Здесь также будет   
представлена общая синтаксическая модель Python. Хотя эта часть в основ-  
ном сосредоточена на описании синтаксиса, тем не менее здесь приводятся   
сведения о дополнительных инструментальных средствах, таких как систе-  
ма PyDoc, и рассматриваются альтернативные стили написания программ-  
ного кода.  
Часть IV «Функции»  
Здесь мы начнем рассматривать высокоуровневые способы структурирова-  
ния программ на языке Python. Функции предоставляют простой способ   
упаковки программного кода многократного использования и предотвра-  
щения появления избыточного кода. В этой части мы исследуем правила   
видимости программных элементов в языке Python, приемы передачи ар-  
гументов и многое другое.  
Часть V «Модули»  
Модули Python позволяют организовать наборы инструкций и функций   
в виде крупных компонентов, и в этой части будет показано, как создавать   
модули, как их использовать и перезагружать. Мы также рассмотрим не-  
которые более сложные темы, такие как пакеты модулей, перезагрузка мо-  
дулей и переменная \_\_name\_\_.  
Часть VI «Классы и ООП»  
Здесь мы приступим к исследованию объектно-ориентированного програм-  
мирования (ООП). Классы – это необязательный, но очень мощный инстру-  
мент структурирования программного кода многократного использования.   
Вы увидите, что классы по большей части используют идеи, которые будут   
описаны к этому моменту, а ООП в языке Python в основном представля-  
ет собой поиск имен в связанных объектах. Здесь вы также увидите, что   
объектно-ориентированный стиль программирования в языке Python не   
является обязательным, но может существенно сократить время разработ-  
ки, особенно если речь идет о долгосрочных проектах.  
Часть VII «Исключения и инструменты»  
Изучение языка мы закончим рассмотрением модели обработки исключи-  
тельных ситуаций, а также кратким обзором инструментальных средств,   
которые особенно удобны при разработке крупных программ (например,   
инструменты отладки и тестирования). Хотя тема исключений является   
достаточно простой, тем не менее она рассматривается после изучения клас-  
сов, так как теперь все исключения должны быть классами.  
Часть VIII «Расширенные возможности»  
В заключительной части мы исследуем некоторые дополнительные возмож-  
ности. Здесь мы поближе познакомимся со строками Юникода и строками   
байтов, со средствами управления атрибутами, такими как свойства и де-  
скрипторы, с декораторами функций и классов, а также с метаклассами.   
Эти главы не являются обязательными для прочтения, потому что далеко   
не всем программистам требуется близкое знакомство с темами, рассмат-  
риваемыми здесь. С другой стороны, читатели, которым приходится зани-

32   
Предисловие  
маться интернационализацией приложений, обработкой двоичных данных   
или проектированием программных интерфейсов для использования дру-  
гими программистами, найдут в этой части немало интересного.  
Часть IX «Приложения»  
Книга заканчивается двумя приложениями, где приводятся рекомендации   
по использованию языка Python на различных платформах (приложение A)   
и варианты решения упражнений, которые приводятся в конце каждой ча-  
сти (приложение B). Ответы на контрольные вопросы, которые приводятся   
в конце каждой главы, находятся непосредственно в самих главах.  
Обратите внимание: предметный указатель и оглавление могут использоваться   
для поиска информации, но в этой книге нет приложений со справочными ма-  
териалами (эта книга – учебник, а не справочник). Как уже говорилось выше,   
в качестве справочников по синтаксису и встроенным особенностям языка   
Python можно использовать книгу «Python Pocket Reference» (O’Reilly), а так-  
же другие книги и руководства, представленные на сайте http://www.python.org.   
Обновления книги  
Книга продолжает улучшаться (исправляются ошибки и опечатки). Обновле-  
ния, дополнения и исправления к этой книге можно найти в сети Интернет на   
одном из следующих сайтов:  
http://www.oreilly.com/catalog/ 9780596158064/ (веб-страница книги на сай-  
те издательства O’Reilly)  
http://www.rmi.net/~lutz (сайт автора книги)  
http://www.rmi.net/~lutz/about-lp.html (веб-страница книги на сайте автора)  
Последний из этих трех URL указывает на веб-страницу, где я выкладываю   
обновления, однако если эта ссылка окажется ошибочной, вам придется вос-  
пользоваться поисковой системой, чтобы восстановить ее. Если бы я был яс-  
новидящим, я указал бы точную ссылку, но Интернет меняется быстрее, чем   
печатаются книги.  
О программах в этой книге  
Эта книга и все примеры программ в ней основаны на использовании Python 3.0.   
Кроме того, значительная часть примеров может выполняться под управлени-  
ем Python 2.6, о чем постоянно будет упоминаться в тексте и в примечаниях   
специально для тех, кто использует Python 2.6.  
Но поскольку эта книга описывает основы языка, можно быть уверенным, что   
большая часть материала в следующих версиях Python изменится не очень   
сильно. Большая часть информации применима и к более ранним версиям   
Python, за исключением некоторых случаев. Естественно, в случае использо-  
вания расширений, которые появятся после выхода этой книги, ничего гаран-  
тировать нельзя.   
Как правило, лучшей версией Python является последняя его версия. Так как   
эта книга описывает основы языка, большинство сведений также применимо   
к Jython – реализации Python на языке Java, а также к другим реализациям,   
описанным в главе 2.

Предисловие   
33  
Исходные тексты примеров, а также ответы к заданиям можно получить на   
веб-сайте книги по адресу http://www.oreilly.com/catalog/ 9780596158064/. Вас   
волнует вопрос, как запускать примеры? Он во всех подробностях обсуждается   
в главе 3, поэтому потерпите до этой главы.  
Использование программного кода примеров  
Данная книга призвана оказать вам помощь в решении ваших задач. Вы мо-  
жете свободно использовать примеры программного кода из этой книги в своих   
приложениях и в документации. Вам не нужно обращаться в издательство за   
разрешением, если вы не собираетесь воспроизводить существенные части про-  
граммного кода. Например, если вы разрабатываете программу и используете   
в ней несколько отрывков программного кода из книги, вам не нужно обращать-  
ся за разрешением. Однако в случае продажи или распространения компакт-  
дисков с примерами из этой книги вам необходимо получить разрешение от   
издательства O’Reilly. Если вы отвечаете на вопросы, цитируя данную книгу   
или примеры из нее, получение разрешения не требуется. Но при включении   
существенных объемов программного кода примеров из этой книги в вашу до-  
кументацию вам необходимо будет получить разрешение издательства.  
Мы приветствуем, но не требуем добавлять ссылку на первоисточник при ци-  
тировании. Под ссылкой на первоисточник мы подразумеваем указание ав-  
торов, издательства и ISBN. Например: «Learning Python, Fourth Edition, by   
Mark Lutz. Copyright 2009 O’Reilly Media, Inc., 978-0-596-15806-4».  
За получением разрешения на использование значительных объемов про-  
граммного кода примеров из этой книги обращайтесь по адресу permissions@  
oreilly.com.  
Типографские соглашения  
В этой книге приняты следующие соглашения:  
Курсив  
Курсив применяется для выделения адресов электронной почты, URL, имен   
файлов и каталогов, а также терминов, когда они упоминаются впервые.  
Моноширинный шрифт  
Применяется для представления содержимого файлов, вывода команд,   
а также для выделения в тексте имен модулей, методов, инструкций и ко-  
манд.  
Моноширинный жирный  
Используется для выделения команд или текста, который должен быть   
введен пользователем, а также для выделения участков программного кода   
в листингах.  
Моноширинный курсив  
Обозначает замещаемые элементы в программном коде и комментарии.  
<Моноширинный шрифт>  
Таким способом выделяются синтаксические элементы, которые должны   
замещаться действительным программным кодом.

34   
Предисловие  
Так выделяются советы, предложения или примечания общего   
характера, имеющие отношение к расположенному рядом тек-  
сту.  
Так выделяются предупреждения или предостережения, имею-  
щие отношение к расположенному рядом тексту.  
В примерах этой книги символ % в начале системной командной   
строки обозначает приглашение к вводу независимо от того, ка-  
кое приглашение используется на вашей машине (например, C:\  
Python30> в окне DOS). Вам не нужно вводить символ %.   
Точно так же в листингах, отображающих сеанс работы с интер-  
претатором, не нужно вводить символы >>> и ..., которые пока-  
заны в начале строки, – это приглашения к вводу, которые вы-  
водятся интерпретатором Python. Вводите лишь текст, который   
находится сразу же за этими приглашениями. Чтобы помочь   
вам запомнить это правило, все, что должно вводиться пользо-  
вателем, выделено жирным шрифтом.   
Кроме того, обычно не требуется вводить текст в листингах, на-  
чинающийся с символа #, так как это комментарии, а не испол-  
няемый программный код.  
Safari® Books Online  
 Safari Books Online – это виртуальная библиотека, позволяю-  
 Books Online – это виртуальная библиотека, позволяю-  
Books Online – это виртуальная библиотека, позволяю-  
 Online – это виртуальная библиотека, позволяю-  
Online – это виртуальная библиотека, позволяю-  
 – это виртуальная библиотека, позволяю-  
щая легко и быстро находить ответы на вопросы среди более   
чем 7500 технических и справочных изданий и видеороликов.  
Подписавшись на услугу, вы сможете загружать любые страницы из книг   
и просматривать любые видеоролики из нашей библиотеки. Читать книги на   
своих мобильных устройствах и сотовых телефонах. Получать доступ к новин-  
кам еще до того, как они выйдут из печати. Читать рукописи, находящиеся   
в работе, и посылать свои отзывы авторам. Копировать и вставлять отрывки   
программного кода, определять свои предпочтения, загружать отдельные гла-  
вы, устанавливать закладки на ключевые разделы, оставлять примечания, пе-  
чатать страницы и пользоваться массой других преимуществ, позволяющих   
экономить ваше время.  
Благодаря усилиям O’Reilly Media данная книга также доступна через услугу   
Safari Books Online. Чтобы получить полный доступ к электронной версии этой   
книги, а также к книгам с похожей тематикой издательства O’Reilly и других   
издательств, подпишитесь бесплатно по адресу http://my.safaribooksonline.com.

Предисловие   
35  
Как с нами связаться  
С вопросами и предложениями, касающимися этой книги, обращайтесь в из-  
дательство:  
O’Reilly Media   
1005 Gravenstein Highway North   
Sebastopol, CA 95472   
800-998-9938 (в Соединенных Штатах Америки или в Канаде)   
707-829-0515 (международный)   
707-829-0104 (факс)  
Список опечаток, файлы с примерами и другую дополнительную информацию   
вы найдете на сайте книги:  
http://www.oreilly.com/catalog/9780596158064/  
Свои пожелания и вопросы технического характера отправляйте по адресу:  
bookquestions@oreilly.com  
Дополнительную информацию о книгах, обсуждения, Центр ресурсов изда-  
тельства O’Reilly вы найдете на сайте:  
http://www.oreilly.com  
Обновления и дополнения к книге вы также можете найти на сайтах, упоми-  
навшихся выше в этом предисловии.  
Благодарности  
Учитывая, что я написал уже четвертое издание этой книги, я не могу не пре-  
бывать в настроении, что мне удалось «выполнить сложное задание». Я исполь-  
зовал и пропагандировал Python на протяжении 17 лет и обучал этому языку   
12 лет. Несмотря на то, что все течет и все меняется, я по-прежнему поражаюсь   
успеху, который сопутствует языку Python. В 1992 году большинство из нас   
едва ли могло предполагать, до какой степени распространится его примене-  
ние. Но, чтобы не выглядеть безнадежно возгордившимся автором, я хотел бы   
немного вспомнить о прошлом и сказать несколько слов благодарности.  
Это была длинная и извилистая дорога. Сейчас понятно, что, когда в 1992 году   
я открыл для себя Python, я предположить не мог, какое влияние он будет ока-  
зывать на мою жизнь в следующие 17 лет. Через два года после начала работы   
над первым изданием «Programming Python» в 1995 году я начал путешество-  
Programming Python» в 1995 году я начал путешество-  
 Python» в 1995 году я начал путешество-  
вать по стране и миру, обучая начинающих программистов этому языку про-  
граммирования. После выхода первого издания «Learning Python» в 1999 году   
преподавание Python и работа над книгами стали моей основной работой во   
многом благодаря интенсивному росту популярности Python.  
В середине 2009 года, когда я пишу эти слова, мною уже написано 12 книг   
о Python (3 из них претерпели 4 издания), а опыт преподавания Python насчи-  
тывает более 10 лет. Проведя более 225 курсов в США, Европе, Канаде и Мек-  
сике, я обучил более 3000 студентов. Помимо множества мучительных часов,   
проведенных в самолетах, эти курсы дали мне возможность существенно улуч-  
шить эту и другие книги о Python. За эти годы преподавание помогало улуч-

36   
Предисловие  
шать качество книг, а работа над книгами – качество преподавания. Фактиче-  
ски книга, которую вы читаете, была почти полностью получена из программы   
моих курсов.  
В связи с этим я хотел бы поблагодарить всех студентов, которые участвовали   
в моих курсах на протяжении последних 12 лет. Как развитие языка Python,   
так и ваши отзывы сыграли немаловажную роль в становлении этой книги.   
(Нет ничего более поучительного, чем наблюдение за 3000 студентами, кото-  
рые совершают одни и те же ошибки, свойственные начинающим программи-  
стам!) Это издание стало возможным в первую очередь благодаря тем, кто по-  
сещал мои курсы после 2003 года, однако все, кто посещал мои курсы начиная   
с 1997 года, так или иначе тоже помогли улучшить эту книгу. Я особенно хотел   
бы поблагодарить компании, предоставившие помещения для проведения кур-  
сов в Дублине, Мехико, Барселоне, Лондоне, Эдмонте и Пуэрто Рико� лучшие   
условия аренды трудно себе представить.  
Я также хотел бы выразить свою благодарность всем, кто принимал участие   
в производстве этой книги. Редакторам, работавшим над этим проектом: Джу-  
лии Стил (Julie Steele), редактору этого издания Татьяне Апанди (Tatiana   
Apandi), работавшей над предыдущим изданием, и многим другим редак-  
торам, работавшим над более ранними изданиями. Дугу Хеллманну (Doug   
Hellmann) и Джесси Ноллер (Jesse Noller) – за участие в техническом обзоре   
этой книги. И издательству O’Reilly – за то, что я получил шанс работать над   
этими 12 проектами книг� это было здорово (правда, порой я чувствовал себя   
как герой фильма «День сурка»).  
Я хочу поблагодарить своего первого соавтора Дэвида Ашера (David Ascher) за   
его работу над ранними изданиями этой книги. Дэвид написал часть «Outer   
Layers» (Внешние уровни) для предыдущих изданий, которую мы, к сожале-  
нию, убрали в третьем издании книги, чтобы освободить место для новых мате-  
риалов об основах языка Python. Чтобы компенсировать эту потерю, в третьем   
издании я добавил больше усложненных программ для самостоятельного изу-  
чения, а в четвертом издании – новые примеры и новую часть, где обсуждаются   
расширенные возможности Python. Если вам не хватает этого материала, про-  
читайте приведенные ранее в предисловии примечания по поводу книг, описы-  
вающих вопросы прикладного программирования.  
За создание такого замечательного и полезного языка я должен поблагодарить   
Гвидо ван Россума (Guido van Rossum) и все сообщество разработчиков и поль-  
Guido van Rossum) и все сообщество разработчиков и поль-  
 van Rossum) и все сообщество разработчиков и поль-  
van Rossum) и все сообщество разработчиков и поль-  
 Rossum) и все сообщество разработчиков и поль-  
Rossum) и все сообщество разработчиков и поль-  
) и все сообщество разработчиков и поль-  
зователей Python. Подобно большинству программных продуктов с открыты-  
ми исходными текстами, Python развивается благодаря героическим усили-  
ям многих программистов. Обладая 17-летним опытом программирования на   
Python, я по-прежнему нахожу его серьезной забавой. Мне очень повезло, что   
я смог наблюдать, как Python из младенца в семействе языков сценариев вырос   
в популярный инструмент, которым пользуются практически все компании,   
занимающиеся разработкой программного обеспечения. Участвовать в этом   
процессе было очень волнующим занятием, и поэтому я хотел бы поблагода-  
рить и поздравить с достигнутыми успехами все сообщество Python.  
Я также хотел бы поблагодарить своего первого редактора из издательства   
O’Reilly, Фрэнка Уиллисона (Frank �illison). Идея этой книги во многом при-  
Frank �illison). Идея этой книги во многом при-  
 �illison). Идея этой книги во многом при-  
�illison). Идея этой книги во многом при-  
). Идея этой книги во многом при-  
надлежит Фрэнку, и в ней нашли отражение его взгляды. Оглядываясь назад,   
можно заметить, что Фрэнк оказал существенное влияние как на мою карьеру,   
так и на Python. Не будет преувеличением сказать, что успех развития Python

Предисловие   
37  
на начальном этапе в определенной степени обусловлен влиянием Фрэнка. Мы   
по-прежнему скучаем по нему.  
В заключение хотелось бы выразить личную благодарность. Компании OQO   
за самые лучшие игрушки, какие я только видел. Покойному Карлу Сагану   
(Carl Sagan) за то, что вдохновил 18-летнего мальчишку из Висконсина. Моей   
матушке за поддержку. И всем крупным корпорациям, с которыми мне прихо-  
дилось иметь дело, за то, что напоминают мне о том, как это здорово работать   
на самого себя.  
Моим детям, Майку, Сэмми и Рокси, за любую будущность, которую они выбе-  
рут. Вы были детьми, когда я начинал работать с языком Python, и вы каким-  
то образом выросли за это время� я горжусь вами. Жизнь может перекрыть нам   
все пути, но путь домой всегда остается открытым.  
Но больше всего я благодарен Вере, моему лучшему другу, моей подруге и моей   
жене. День, когда я нашел тебя, был лучшим днем в моей жизни. Я не знаю,   
что принесут мне следующие 50 лет, но я знаю, что хотел бы прожить их рядом   
с тобой.  
Марк Лутц   
Сарасота, Флорида   
Июль 2009

Часть I.  
Введение

Глава 1.  
   
Python в вопросах и ответах  
Если вы купили эту книгу, вы, скорее всего, уже знаете, что такое Python и на-  
сколько важно овладеть этим инструментом. Если это не так, вы наверняка не   
сможете зарабатывать программированием, пока не изучите этот язык, про-  
читав оставшуюся часть этой книги, и не напишете пару проектов. Но прежде   
чем мы приступим к изучению деталей, давайте сначала рассмотрим основные   
причины высокой популярности Python. Перед тем как приступить собственно   
к языку, в этой главе мы рассмотрим некоторые вопросы, которые обычно за-  
дают начинающие программисты, и дадим ответы на них.  
Почему программисты используют Python?  
Это самый типичный вопрос, который задают начинающие программисты,   
потому что на сегодняшний день существует масса других языков програм-  
мирования. Учитывая, что число пользователей Python составляет порядка   
миллиона человек, достаточно сложно однозначно ответить на этот вопрос. Вы-  
бор средств разработки иногда зависит от уникальных особенностей и личных   
предпочтений.  
Однако после обучения примерно 225 групп и более 3000 студентов за послед-  
ние 12 лет у меня накопились некоторые общие мысли по этому поводу. Основ-  
ные факторы, которые приводятся пользователями Python, примерно таковы:  
Качество программного обеспечения  
Для многих основное преимущество языка Python заключается в удобочи-  
таемости, ясности и более высоком качестве, отличающими его от других   
инструментов в мире языков сценариев. Программный код на языке Python   
читается легче, а значит, многократное его использование и обслужива-  
ние выполняется гораздо проще, чем использование программного кода на   
других языках сценариев. Единообразие оформления программного кода   
на языке Python облегчает его понимание даже для тех, кто не участвовал   
в его создании. Кроме того, Python поддерживает самые современные меха-  
низмы многократного использования программного кода, каким является   
объектно-ориентированное программирование (ООП).

42   
Глава 1. Python в вопросах и ответах   
Высокая скорость разработки  
По сравнению с компилирующими или строго типизированными языками,   
такими как C, C++ и Java, Python во много раз повышает производитель-  
ность труда разработчика. Объем программного кода на языке Python обыч-  
но составляет треть или даже пятую часть эквивалентного программного   
кода на языке C++ или Java. Это означает меньший объем ввода с клавиату-  
ры, меньшее количество времени на отладку и меньший объем трудозатрат   
на сопровождение. Кроме того, программы на языке Python запускаются   
сразу же, минуя длительные этапы компиляции и связывания, необходи-  
мые в некоторых других языках программирования, что еще больше уве-  
личивает производительность труда программиста.  
Переносимость программ  
Большая часть программ на языке Python выполняется без изменений на   
всех основных платформах. Перенос программного кода из операционной   
системы Linux в �indows обычно заключается в простом копировании фай-  
лов программ с одной машины на другую. Более того, Python предоставляет   
массу возможностей по созданию переносимых графических интерфейсов,   
программ доступа к базам данных, веб-приложений и многих других типов   
программ. Даже интерфейсы операционных систем, включая способ запу-  
ска программ и обработку каталогов, в языке Python реализованы перено-  
симым способом.  
Библиотеки поддержки  
В составе Python поставляется большое число собранных и переносимых   
функциональных возможностей, известных как стандартная  библиоте-  
ка. Эта библиотека предоставляет массу возможностей, востребованных   
в прикладных программах, начиная от поиска текста по шаблону и закан-  
чивая сетевыми функциями. Кроме того, Python допускает расширение как   
за счет ваших собственных библиотек, так и за счет библиотек, созданных   
сторонними разработчиками. Из числа сторонних разработок можно на-  
звать инструменты создания веб-сайтов, программирование математиче-  
ских вычислений, доступ к последовательному порту, разработку игровых   
программ и многое другое. Например, расширение NumPy позиционирует-  
ся как свободный и более мощный эквивалент системы программирования   
математических вычислений Mathlab.  
Интеграция компонентов  
Сценарии Python легко могут взаимодействовать с другими частями при-  
ложения благодаря различным механизмам интеграции. Эта интеграция   
позволяет использовать Python для настройки и расширения функцио-  
нальных возможностей программных продуктов. На сегодняшний день   
программный код на языке Python имеет возможность вызывать функции   
из библиотек на языке C/C++, сам вызываться из программ, написанных   
на языке C/C++, интегрироваться с программными компонентами на языке   
Java, взаимодействовать с такими платформами, как COM и .NET, и произ-  
водить обмен данными через последовательный порт или по сети с помощью   
таких протоколов, как SOAP, XML-RPC и CORBA. Python – не обособлен-  
ython – не обособлен-  
 – не обособлен-  
ный инструмент.

Почему программисты используют Python?   
43  
Удовольствие  
Благодаря непринужденности языка Python и наличию встроенных ин-  
струментальных средств процесс программирования может приносить удо-  
вольствие. На первый взгляд это трудно назвать преимуществом, тем не   
менее, удовольствие, получаемое от работы, напрямую влияет на произво-  
дительность труда.  
Из всех перечисленных факторов наиболее существенными для большинства   
пользователей являются первые два (качество и производительность).  
Качество программного обеспечения  
По своей природе Python имеет простой, удобочитаемый синтаксис и ясную мо-  
дель программирования. Согласно лозунгу, выдвинутому на недавней конфе-  
ренции по языку Python, основное его преимущество состоит в том, что Python   
«каждому по плечу» – характеристики языка взаимодействуют ограниченным   
числом непротиворечивых способов и естественно вытекают из небольшого   
круга базовых концепций. Это делает язык простым в освоении, понимании   
и запоминании. На практике программистам, использующим язык Python,   
почти не приходится прибегать к справочным руководствам – это непротиво-  
речивая система, на выходе которой, к удивлению многих, получается профес-  
сиональный программный код.  
Философия Python по сути диктует использование минималистского подхода.   
Это означает, что даже при наличии нескольких вариантов решения задачи   
в этом языке обычно существует всего один очевидный путь, небольшое чис-  
ло менее очевидных альтернатив и несколько взаимосвязанных вариантов ор-  
ганизации взаимодействий. Более того, Python не принимает решения за вас,   
когда порядок взаимодействий неочевиден – предпочтение отдается явному   
описанию, а не «волшебству». В терминах Python явное лучше неявного, а про-  
стое лучше сложного.1  
Помимо философии Python обладает такими возможностями, как модульное   
и объектно-ориентированное программирование, что естественно упрощает   
возможность многократного использования программного кода. Поскольку   
качество находится в центре внимания самого языка Python, оно также нахо-  
дится в центре внимания программистов.  
Высокая скорость разработки  
Во время бума развития Интернета во второй половине 1990-х годов было слож-  
но найти достаточное число программистов для реализации программных про-  
ектов – от разработчиков требовалось писать программы со скоростью разви-  
тия Интернета. Теперь, в эпоху экономического спада, картина изменилась.   
1   
Чтобы получить более полное представление о философии Python, введите в команд-  
ной строке интерпретатора команду import this (как это сделать, будет рассказано   
в главе 2). В результате будет активизировано «пасхальное яйцо», скрытое в недрах   
Python, – сборник принципов проектирования, лежащих в основе Python. Аббревиа-  
тура EIBTI, происходящая от фразы «explicit is better than implicit» («явное лучше   
неявного»), превратилась в модное жаргонное словечко.

44   
Глава 1. Python в вопросах и ответах   
Сегодня от программистов требуется умение решать те же задачи меньшим   
числом сотрудников.  
В обоих этих случаях Python блистал как инструмент, позволяющий про-  
граммистам получать большую отдачу при меньших усилиях. Он изначально   
оптимизирован для достижения высокой скорости разработки – простой син-  
таксис, динамическая типизация, отсутствие этапа компиляции и встроенные   
инструментальные средства позволяют программистам создавать программы   
за меньшее время, чем при использовании некоторых других инструментов.   
В результате Python увеличивает производительность труда разработчика во   
много раз по сравнению с традиционными языками программирования. Это   
значительное преимущество, которое с успехом может использоваться как во   
время бума, так и во время спада, а также во время любого промежуточного   
этапа развития индустрии программного обеспечения.  
Является ли Python «языком сценариев»?  
Python – это многоцелевой язык программирования, который зачастую ис-  
 – это многоцелевой язык программирования, который зачастую ис-  
пользуется для создания сценариев. Обычно он позиционируется как объ ект-  
но-ориентированный язык сценариев – такое определение смешивает поддерж-  
ку ООП с общей ориентацией на сценарии. Действительно, для обозначения   
файлов с программным кодом на языке Python программисты часто исполь-  
зуют слово «сценарий» вместо слова «программа». В этой книге термины «сце-  
нарий» и «программа» рассматриваются как взаимозаменяемые с некоторым   
предпочтением термина «сценарий» для обозначения простейших программ,   
помещающихся в единственный файл, и термина «программа» для обозна-  
чения более сложных приложений, программный код которых размещается   
в нескольких файлах.  
Термин «язык сценариев» имеет множество различных толкований. Некото-  
рые предпочитают вообще не применять его к языку Python. У большинства   
термин «язык сценариев» вызывает три разных ассоциации, из которых одни   
более применимы к языку Python, чем другие:  
Командные оболочки  
Иногда, когда кто-то слышит, что Python – это язык сценариев, то представ-  
ляет себе Python как инструмент для создания системных сценариев. Такие   
программы часто запускаются из командной строки с консоли и решают   
такие задачи, как обработка текстовых файлов и запуск других программ.  
Программы на языке Python способны решать такие задачи, но это лишь   
одна из десятков прикладных областей, где может применяться Python.   
Это не только язык сценариев командной оболочки.  
Управляющий язык  
Другие пользователи под названием «язык сценариев» понимают «связую-  
щий» слой, который используется для управления другими прикладными   
компонентами (то есть для описания сценария работы). Программы на язы-  
ке Python действительно нередко используются в составе крупных прило-  
жений. Например, при проверке аппаратных устройств программы на язы-  
ке Python могут вызывать компоненты, осуществляющие низкоуровневый   
доступ к устройствам. Точно так же программы могут запускать программ-

Все хорошо, но есть ли у него недостатки?   
45  
ный код на языке Python для поддержки настройки программного продук-  
та у конечного пользователя, что ликвидирует необходимость поставлять   
и пересобирать полный объем исходных текстов.  
Простота языка Python делает его весьма гибким инструментом управле-  
ния. Тем не менее технически – это лишь одна из многих ролей, которые   
может играть Python. Многие программисты пишут на языке Python ав-  
тономные сценарии, которые не используют какие-либо интегрированные   
компоненты. Это не только язык управления.  
Удобство в использовании   
Пожалуй, лучше всего представлять себе термин «язык сценариев» как   
обозначение простого языка, используемого для быстрого решения задач.   
Это особенно верно, когда термин применяется к языку Python, который   
позволяет вести разработку гораздо быстрее, чем компилирующие языки   
программирования, такие как C++. Ускоренный цикл разработки способ-  
ствует применению зондирующего, поэтапного стиля программирования,   
который следует попробовать, чтобы оценить по достоинству.  
Не надо заблуждаться, Python предназначен не только для решения про-  
стых задач. Скорее, он упрощает решение задач благодаря своей простоте   
и гибкости. Язык Python имеет небольшой набор возможностей, но он по-  
зволяет создавать программы неограниченной сложности. По этой причине   
Python обычно используется как для быстрого решения тактических, так   
и для решения долговременных, стратегических задач.  
Итак, является ли Python языком сценариев? Ответ зависит от того, к кому   
обращен вопрос. Вообще термин «создание сценариев», вероятно, лучше ис-  
пользовать для описания быстрого и гибкого способа разработки, который под-  
держивается языком Python, а не для описания прикладной области програм-  
мирования.  
Все хорошо, но есть ли у него недостатки?  
После 17 лет работы с языком Python и 12 лет преподавания единственный не-  
достаток, который мне удалось обнаружить, – это скорость выполнения про-  
грамм, которая не всегда может быть такой же высокой, как у программ, напи-  
санных на компилирующих языках программирования, таких как C или C++.  
Подробнее о концепциях реализации мы поговорим ниже в этой книге. В двух   
словах замечу, что в современной реализации Python компилирует (то есть   
транслирует) инструкции исходного программного кода в промежуточное   
представление, известное как байт-код, и затем интерпретирует этот байт-код.   
Байт-код обеспечивает переносимость программ, поскольку это платформоне-  
зависимый формат. Однако из-за того что Python не создает двоичный машин-  
ный код (например, машинные инструкции для микропроцессора Intel), неко-  
торые программы на языке Python могут работать медленнее своих аналогов,   
написанных на компилирующих языках, таких как C.  
Будет ли вас когда-нибудь волновать разница в скорости выполнения про-  
грамм, зависит от того, какого рода программы вы пишете. Python многократ-  
Python многократ-  
 многократ-  
но подвергался оптимизации и в отдельных прикладных областях программ-  
ный код на этом языке отличается достаточно высокой скоростью выполнения.

46   
Глава 1. Python в вопросах и ответах   
Кроме того, когда в сценарии Python делается что-нибудь «значительное», на-  
пример обрабатывается файл или конструируется графический интерфейс,   
ваша программа фактически выполняется со скоростью, которую способен   
дать язык C, потому что такого рода задачи решаются компилированным   
с языка С программным кодом, лежащим в недрах интерпретатора Python. Го-  
раздо важнее, что преимущество в скорости разработки порой важнее потери   
скорости выполнения, особенно если учесть быстродействие современных ком-  
пьютеров.  
Тем не менее даже при высоком быстродействии современных процессоров   
остаются такие области, где требуется максимальная скорость выполнения.   
Реализация математических вычислений и анимационных эффектов, напри-  
мер, часто требует наличия базовых вычислительных компонентов, которые   
решают свои задачи со скоростью языка C (или еще быстрее). Если вы работае-  
те как раз в такой области, вы все равно сможете использовать Python, доста-  
точно лишь выделить из приложения компоненты, требующие максимальной   
скорости работы, в виде компилированных расширений и связать их системой   
сценариев на языке Python.  
В этой книге мы не будем обсуждать расширения слишком подробно, но это   
один из примеров, когда Python может играть упоминавшуюся выше роль язы-  
Python может играть упоминавшуюся выше роль язы-  
 может играть упоминавшуюся выше роль язы-  
ка управления. Типичным примером такой двуязычной стратегии может слу-  
жить расширение NumPy, содержащее реализацию математических вычисле-  
ний для Python� благодаря комбинированию компилированных и оптимизи-  
Python� благодаря комбинированию компилированных и оптимизи-  
� благодаря комбинированию компилированных и оптимизи-  
рованных библиотек расширения с языком Python NumPy превращает Python   
в мощный, эффективный и удобный инструмент математических вычислений.   
Возможно, вам никогда не придется создавать подобные расширения, но вы   
должны знать, что в случае необходимости они могут предоставить в ваше рас-  
поряжение мощный механизм оптимизации.  
Кто в наше время использует Python?  
К моменту, когда я пишу эти строки, наиболее правдоподобной оценкой чис-  
ла пользователей Python является число, близкое 1 миллиону человек во всем   
мире (с небольшой погрешностью). Эта оценка основана на различных стати-  
стических показателях, таких как количество загрузок и результаты опросов   
разработчиков. Дать более точную оценку достаточно сложно, потому что Py-  
Py-  
thon является открытым программным обеспечением – для его использования   
не требуется проходить лицензирование. Более того, Python по умолчанию   
включается в состав дистрибутивов Linux, поставляется вместе с компьютера-  
Linux, поставляется вместе с компьютера-  
, поставляется вместе с компьютера-  
ми Macintosh и некоторыми другими программными и аппаратными продук-  
Macintosh и некоторыми другими программными и аппаратными продук-  
 и некоторыми другими программными и аппаратными продук-  
тами, что существенно затрудняет оценку числа пользователей.  
Вообще же количество пользователей Python значительно больше и вокруг   
него сплотилось очень активное сообщество разработчиков. Благодаря тому,   
что Python появился более 19 лет тому назад и получил широкое распростра-  
Python появился более 19 лет тому назад и получил широкое распростра-  
 появился более 19 лет тому назад и получил широкое распростра-  
нение, он отличается высокой стабильностью и надежностью. Python исполь-  
Python исполь-  
 исполь-  
зуется не только отдельными пользователями, он также применяется компа-  
ниями для создания продуктов, приносящих настоящую прибыль. Например:

Кто в наше время использует Python?   
47  
 •  
Компания Google широко использует Python в своей поисковой системе   
и оплачивает труд создателя Python.  
 •  
Служба коллективного использования видеоматериалов �ouTube в значи-  
�ouTube в значи-  
 в значи-  
тельной степени реализована на языке Python.  
 •  
Популярная программа BitTorrent для обмена файлами в пиринговых се-  
BitTorrent для обмена файлами в пиринговых се-  
 для обмена файлами в пиринговых се-  
тях (peer-to-peer) написана на языке Python.  
 •  
Популярный веб-фреймворк App Engine от компании Google использует   
Python в качестве прикладного языка программирования.  
 •  
Такие компании, как EVE Online и Massively Multiplayer Online Game   
(MMOG), широко используют Python в своих разработках.  
 •  
Мощная система трехмерного моделирования и создания мультиплика-  
ции Maya поддерживает интерфейс для управления из сценариев на языке   
Python.  
 •  
Такие компании, как Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm   
и IBM, используют Python для тестирования аппаратного обеспечения.  
 •  
Такие компании, как Industrial Light & Magic, Pixar и другие, используют   
Python в производстве анимационных фильмов.  
 •  
Компании JPMorgan Chase, UBS, Getco и Citadel применяют Python для   
прогнозирования финансового рынка.  
 •  
NASA, Los Alamos, Fermilab, JPL и другие используют Python для научных   
вычислений.  
 •  
iRobot использует Python в разработке коммерческих роботизированных   
устройств.  
 •  
ESRI использует Python в качестве инструмента настройки своих популяр-  
Python в качестве инструмента настройки своих популяр-  
 в качестве инструмента настройки своих популяр-  
ных геоинформационных программных продуктов под нужды конечного   
пользователя.  
 •  
NSA использует Python для шифрования и анализа разведданных.  
 •  
В реализации почтового сервера IronProt используется более 1 миллиона   
строк программного кода на языке Python.  
 •  
Проект «ноутбук каждому ребенку» (One Laptop Per Child, OLPC) строит   
свой пользовательский интерфейс и модель функционирования на языке   
Python.  
И так далее. Пожалуй, единственное, что объединяет все эти компании, – это   
то, что для решения широкого спектра задач прикладного программирования   
используется язык программирования Python. Универсальная природа языка   
обеспечивает возможность его применения в самых разных областях. Факти-  
чески с определенной долей уверенности можно утверждать, что Python так   
или иначе используется практически каждой достаточно крупной организаци-  
ей, занимающейся разработкой программного обеспечения, – как для реше-  
ния краткосрочных тактических задач, так и для разработки долгосрочных   
стратегических проектов. Как оказалось, Python прекрасно зарекомендовал   
себя в обоих случаях.  
За дополнительными сведениями о компаниях, использующих Python, обра-  
Python, обра-  
, обра-  
щайтесь на веб-сайт http://www.python.org.

48   
Глава 1. Python в вопросах и ответах   
Что можно делать с помощью Python?  
Будучи удачно спроектированным языком программирования Python пре-  
Python пре-  
 пре-  
красно подходит для решения реальных задач из разряда тех, которые разра-  
ботчикам приходится решать ежедневно. Он используется в самом широком   
спектре применений – и как инструмент управления другими программными   
компонентами, и для реализации самостоятельных программ. Фактически   
круг ролей, которые может играть Python как многоцелевой язык программи-  
Python как многоцелевой язык программи-  
 как многоцелевой язык программи-  
рования, практически не ограничен: он может использоваться для реализации   
всего, что угодно, – от веб-сайтов и игровых программ до управления роботами   
и космическими кораблями.  
Однако сферу использования Python в настоящее время можно разбить на не-  
Python в настоящее время можно разбить на не-  
 в настоящее время можно разбить на не-  
сколько широких категорий. Следующие несколько разделов описывают наи-  
более типичные области применения Python в наши дни, а также инструмен-  
Python в наши дни, а также инструмен-  
 в наши дни, а также инструмен-  
тальные средства, используемые в каждой из областей. У нас не будет возмож-  
ности заняться исследованием инструментов, упоминаемых здесь. Если какие-  
то из них заинтересуют вас, обращайтесь на веб-сайт проекта Python за более   
подробной информацией.  
Системное программирование  
Встроенные в Python интерфейсы доступа к службам операционных систем   
делают его идеальным инструментом для создания переносимых программ   
и утилит системного администрирования (иногда они называются инструмен-  
тами  командной  оболочки). Программы на языке Python могут отыскивать   
файлы и каталоги, запускать другие программы, производить параллельные   
вычисления с использованием нескольких процессов и потоков и делать мно-  
гое другое.  
Стандартная библиотека Python полностью отвечает требованиям стандар-  
Python полностью отвечает требованиям стандар-  
 полностью отвечает требованиям стандар-  
тов POSIX и поддерживает все типичные инструменты операционных систем:   
переменные окружения, файлы, сокеты, каналы, процессы, многопоточную   
модель выполнения, поиск по шаблону с использованием регулярных выра-  
жений, аргументы командной строки, стандартные интерфейсы доступа к по-  
токам данных, запуск команд оболочки, дополнение имен файлов и многое   
другое. Кроме того, системные интерфейсы в языке Python созданы переноси-  
Python созданы переноси-  
 созданы переноси-  
мыми, например сценарий копирования дерева каталогов не требует внесения   
изменений, в какой бы операционной системе он ни использовался. Система   
Stackless  Python, используемая компанией EVE Online, также предлагает   
улучшенные решения, применяемые для параллельной обработки данных.  
Графический интерфейс  
Простота Python и высокая скорость разработки делают его отличным сред-  
Python и высокая скорость разработки делают его отличным сред-  
 и высокая скорость разработки делают его отличным сред-  
ством разработки графического интерфейса. В состав Python входит стандарт-  
Python входит стандарт-  
 входит стандарт-  
ный объектно-ориентированный интерфейс к Tk GUI API, который называется   
tkinter (в Python 2.6 он называется Tkinter), позволяющий программам на язы-  
ке Python реализовать переносимый графический интерфейс с внешним видом,   
присущим операционной системе. Графические интерфейсы на базе Python/  
tkinter без изменений могут использоваться в MS �indows, X �indow (в опе-  
kinter без изменений могут использоваться в MS �indows, X �indow (в опе-  
 без изменений могут использоваться в MS �indows, X �indow (в опе-  
MS �indows, X �indow (в опе-  
 �indows, X �indow (в опе-  
�indows, X �indow (в опе-  
, X �indow (в опе-  
X �indow (в опе-  
 �indow (в опе-  
�indow (в опе-  
 (в опе-

Что можно делать с помощью Python?   
49  
рационных системах UNIX и Linux) и Mac OS (как в классической версии, так   
и в OS X). Свободно распространяемый пакет расширения PMW содержит до-  
полнительные визуальные компоненты для набора tkinter. Кроме того, суще-  
kinter. Кроме того, суще-  
. Кроме того, суще-  
ствует прикладной интерфейс wxPython GUI API, основанный на библиотеке   
C++, который предлагает альтернативный набор инструментальных средств   
построения переносимых графических интерфейсов на языке Python.  
Инструменты высокого уровня, такие как PythonCard и Dabo, построены на   
основе таких API, как wxPython и tkinter. При выборе соответствующей би-  
блиотеки вы также сможете использовать другие инструменты создания   
графического интерфейса, такие как Qt (с помощью PyQt), GTK (с помощью   
PyGtk), MFC (с помощью Py�in32), .NET (с помощью IronPython), Swing (с по-  
мощью Jython – реализации языка Python на Java, которая описывается в гла-  
Jython – реализации языка Python на Java, которая описывается в гла-  
 – реализации языка Python на Java, которая описывается в гла-  
Python на Java, которая описывается в гла-  
 на Java, которая описывается в гла-  
Java, которая описывается в гла-  
, которая описывается в гла-  
ве 2, или JPype). Для разработки приложений с веб-интерфейсом или не предъ-  
являющих высоких требований к интерфейсу можно использовать Jython,   
веб-фреймворки на языке Python и CGI-сценарии, которые описываются в сле-  
CGI-сценарии, которые описываются в сле-  
-сценарии, которые описываются в сле-  
дующем разделе и обеспечивают дополнительные возможности по созданию   
пользовательского интерфейса.  
Веб-сценарии  
Интерпретатор Python поставляется вместе со стандартными интернет-моду-  
лями, которые позволяют программам выполнять разнообразные сетевые опе-  
рации как в режиме клиента, так и в режиме сервера. Сценарии могут произ-  
водить взаимодействия через сокеты, извлекать информацию из форм, отправ-  
ленных серверным CGI-сценариям� передавать файлы по протоколу FTP� обра-  
батывать файлы XML� передавать, принимать, создавать и производить разбор   
писем электронной почты� загружать веб-страницы с указанных адресов URL�   
производить разбор разметки HTML и XML полученных веб-страниц� произво-  
XML полученных веб-страниц� произво-  
 полученных веб-страниц� произво-  
дить взаимодействия по протоколам XML-RPC, SOAP и Telnet и многое другое.   
Библиотеки, входящие в состав Python, делают реализацию подобных задач   
удивительно простым делом.  
Кроме того, существует огромная коллекция сторонних инструментов для соз-  
дания сетевых программ на языке Python, которые можно найти в Интернете.   
Например, система HTMLGen позволяет создавать HTML-страницы на основе   
описаний классов Python. Пакет mod\_python предназначен для запуска сце-  
нариев на языке Python под управлением веб-сервера Apache и поддерживает   
шаблоны механизма Python Server Pages. Система Jython обеспечивает бесшо-  
Jython обеспечивает бесшо-  
 обеспечивает бесшо-  
вную интеграцию Python/Java и поддерживает серверные апплеты, которые   
выполняются на стороне клиента.   
Помимо этого для Python существуют полноценные пакеты веб-разработки, та-  
кие как Django, TurboGears, web2py, Pylons, Zope и WebWare, поддерживающие   
возможность быстрого создания полнофункциональных высококачественных   
веб-сайтов на языке Python. Многие из них включают такие возможности, как   
объектно-реляционные отображения, архитектура Модель/Представление/  
Кон троллер (Model/View/Controller), создание сценариев, выполняющихся на   
стороне сервера, поддержка шаблонов и технологии AJAX, предоставляя за-  
AJAX, предоставляя за-  
, предоставляя за-  
конченные и надежные решения для разработки веб-приложений.

50   
Глава 1. Python в вопросах и ответах   
Интеграция компонентов  
Возможность интеграции программных компонентов в единое приложение   
с помощью Python уже обсуждалась выше, когда мы говорили о Python как   
о языке управления. Возможность Python расширяться и встраиваться в си-  
стемы на языке C и C++ делает его удобным и гибким языком для описания по-  
C и C++ делает его удобным и гибким языком для описания по-  
 и C++ делает его удобным и гибким языком для описания по-  
ведения других систем и компонентов. Например, интеграция с библиотекой   
на языке C позволяет Python проверять наличие и запускать библиотечные   
компоненты, а встраивание Python в программные продукты позволяет про-  
изводить настройку программных продуктов без необходимости пересобирать   
эти продукты или поставлять их с исходными текстами.  
Такие инструменты, как Swing и SIP, автоматически генерирующие про-  
граммный код, могут автоматизировать действия по связыванию скомпили-  
рованных компонентов в Python для последующего их использования в сце-  
нариях, а система Cython позволяет программистам смешивать программный   
код на Python и C. Такие огромные платформы на Python, как поддержка COM   
в MS �indows, Jython – реализация на языке Java, IronPython – реализация   
на базе .NET и разнообразные реализации CORBA, предоставляют альтерна-  
тивные способы организации взаимодействий с программными компонента-  
ми. Например, в операционной системе �indows сценарии на языке Python   
могут использовать платформы управления такими приложениями, как MS   
�ord и Excel.  
Приложения баз данных  
В языке Python имеются интерфейсы доступа ко всем основным реляционным   
базам данных – Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL, SQLite   
и многим другим. В мире Python существует также переносимый прикладной   
программный интерфейс баз данных, предназначенный для доступа к базам   
данных SQL из сценариев на языке Python, который унифицирует доступ   
к различным базам данных. Например, при использовании переносимого API   
сценарий, предназначенный для работы со свободной базой данных MySQL,   
практически без изменений сможет работать с другими системами баз данных   
(такими как Oracle). Все, что потребуется сделать для этого, – заменить исполь-  
зуемый низкоуровневый интерфейс.  
Стандартный модуль pickle реализует простую систему хранения объектов,   
что позволяет программам сохранять и восстанавливать объекты Python   
в файлах или в специализированных объектах. В Сети можно также найти   
систему, созданную сторонними разработчиками, которая называется ZODB.   
Она представляет собой полностью объектно-ориентированную базу данных   
для использования в сценариях на языке Python. Существуют также инстру-  
менты, такие как SQLObject и SQLAlchemy, которые отображают реляцион-  
ные таблицы в модель классов языка Python. Начиная с версии Python 2.5,   
стандартной частью Python стала база данных SQLite.   
Быстрое создание прототипов  
В программах на языке Python компоненты, написанные на Python и на C, вы-  
глядят одинаково. Благодаря этому можно сначала создавать прототипы си-  
стем на языке Python, а затем переносить выбранные компоненты на компили-

Что можно делать с помощью Python?   
51  
рующие языки, такие как C и C++. В отличие от ряда других инструментов раз-  
работки прототипов, язык Python не требует, чтобы система была полностью   
переписана, как только прототип будет отлажен. Части системы, которые не   
требуют такой эффективности выполнения, какую обеспечивает C++, можно   
оставить на языке Python, что существенно упростит сопровождение и исполь-  
зование такой системы.  
Программирование математических   
и научных вычислений  
Расширение NumPy для математических вычислений, упоминавшееся выше,   
включает такие мощные элементы, как объекты массивов, интерфейсы к стан-  
дартным математическим библиотекам, и многое другое. Расширение NumPy –   
за счет интеграции с математическими библиотеками, написанными на ком-  
пилирующих языках программирования – превращает Python в сложный,   
но удобный инструмент программирования математических вычислений, ко-  
торый зачастую может заменить существующий программный код, написан-  
ный на традиционных компилирующих языках, таких как FORTRAN и C++.   
Дополнительные инструменты математических вычислений для Python под-  
держивают возможность создания анимационных эффектов и трехмерных   
объектов, позволяют организовать параллельные вычисления и так далее.   
Например, популярные расширения SciPy и ScientificPython предоставляют   
дополнительные библиотеки для научных вычислений и используют возмож-  
ности расширения NumPy.  
Игры, изображения, искусственный интеллект,   
XML роботы и многое другое  
Язык программирования Python можно использовать для решения более ши-  
рокого круга задач, чем может быть упомянуто здесь. Например:  
 •  
Создавать игровые программы и анимационные ролики с помощью систе-  
мы pygame   
 •  
Обмениваться данными с другими компьютерами через последовательный   
порт с помощью расширения PySerial  
 •  
Обрабатывать изображения с помощью расширений PIL, PyOpenGL,   
Blender, Maya и других  
 •  
Управлять роботом с помощью инструмента PyRo   
 •  
Производить разбор XML-документов с помощью пакета xml, модуля xmlrp-  
clib и расширений сторонних разработчиков  
 •  
Программировать искусственный интеллект с помощью эмулятора нейро-  
сетей и оболочек экспертных систем  
 •  
Анализировать фразы на естественном языке с помощью пакета NLTK.   
Можно даже разложить пасьянс с помощью программы PySol. Поддержку   
многих других прикладных областей можно найти на веб-сайте PyPI или с по-  
мощью поисковых систем (ищите ссылки с помощью Google или на сайте http://  
www.python.org).

52   
Глава 1. Python в вопросах и ответах   
Вообще говоря, многие из этих областей применения Python – всего лишь раз-  
новидности одной и той же роли под названием «интеграция компонентов».   
Использование Python в качестве интерфейса к библиотекам компонентов, на-  
писанных на языке C, делает возможным создание сценариев на языке Python   
для решения задач в самых разных прикладных областях. Как универсаль-  
ный, многоцелевой язык программирования, поддерживающий возможность   
интеграции, Python может применяться очень широко.  
Как осуществляется поддержка Python?  
Будучи популярным и открытым проектом, Python имеет многочисленное   
и активное сообщество разработчиков, которые решают проблемы и вносят   
улучшения со скоростью, которую многие коммерческие разработчики сочли   
бы поразительной (если не шокирующей). Деятельность разработчиков Python   
координируется с помощью системы управления исходными текстами. Из-  
менения в языке принимаются только после прохождения формальной про-  
цедуры (известной как «программа совершенствования продукта», или PEP)   
и должны сопровождаться обширными наборами тестов для системы регрес-  
сивного тестирования Python. Фактически в настоящее время работа над Py-  
Python. Фактически в настоящее время работа над Py-  
. Фактически в настоящее время работа над Py-  
Py-  
thon мало чем отличается от работы над коммерческими программными про-  
 мало чем отличается от работы над коммерческими программными про-  
дуктами и очень сильно отличается от того, как велась разработка на первых   
порах, когда достаточно было отправить создателю языка письмо по электрон-  
ной почте. Но самое главное преимущество проекта – огромное количество до-  
 самое главное преимущество проекта – огромное количество до-  
самое главное преимущество проекта – огромное количество до-  
 главное преимущество проекта – огромное количество до-  
главное преимущество проекта – огромное количество до-  
 преимущество проекта – огромное количество до-  
преимущество проекта – огромное количество до-  
 проекта – огромное количество до-  
проекта – огромное количество до-  
 – огромное количество до-  
огромное количество до-  
 количество до-  
количество до-  
 до-  
до-  
бровольных помощников.  
Существует официальная некоммерческая организация PSF (Python Software   
Foundation), которая занимается организацией конференций и решением про-  
), которая занимается организацией конференций и решением про-  
блем, связанных с интеллектуальной собственностью. По всему миру прово-  
дится огромное количество конференций, самыми крупными из которых яв-  
ляются OSCON (организатор – издательство O’Reilly) и PyCon (организатор –   
PSF). Первая из них рассматривает различные открытые проекты, а вторая   
посвящена исключительно событиям, связанным с языком Python, который   
переживает бурный рост в последние несколько лет. Количество посетителей   
PyCon в 2008 году практически удвоилось по сравнению с предыдущим го-  
дом, увеличившись с 586 посетителей в 2007 году до более 1000 посетителей   
в 2008. Этому удвоению предшествовало 40% увеличение числа посетителей   
в 2007 году, с 410 человек в 2006. В 2009 году конференцию PyCon посетили   
943 человека, немного меньше, чем в 2008, но все равно достаточно много для   
периода глобального кризиса.  
В чем сильные стороны Python?  
Естественно – это вопрос разработчика. Если у вас еще нет опыта программи-  
рования, язык следующих нескольких разделов может показаться немного не-  
понятным, но не волнуйтесь, мы будем рассматривать все эти термины позд-  
нее, в ходе изложения материала. А для разработчиков ниже приводится крат-  
кое введение в некоторые особенности языка Python.

В чем сильные стороны Python?   
53  
Он объектно-ориентированный  
Python изначально является объектно-ориентированным языком программи-  
 изначально является объектно-ориентированным языком программи-  
рования. Его объектная модель поддерживает такие понятия, как полимор-  
физм, перегрузка операторов и множественное наследование, однако, учиты-  
вая простоту синтаксиса и типизации Python, ООП не вызывает сложностей   
в применении. Если эти термины вам непонятны, позднее вы обнаружите, что   
изучать Python гораздо легче, чем другие объектно-ориентированные языки   
программирования.  
Объектно-ориентированная природа Python, являясь мощным средством   
структурирования программного кода многократного пользования, кроме   
того, делает этот язык идеальным инструментом поддержки сценариев для   
объектно-ориентированных языков, таких как C++ и Java. Например, при на-  
личии соответствующего связующего программного кода, программы на язы-  
ке Python могут использовать механизм наследования от классов, реализован-  
ных на C++, Java и C#.  
Как бы то ни было, но ООП не является обязательным в Python� вы сможе-  
те стать опытным программистом и при этом не быть специалистом по ООП.   
Как и C++, Python поддерживает оба стиля программирования – процедурный   
и объектно-ориентированный. Объектно-ориентированные механизмы могут   
использоваться по мере необходимости. Это особенно удобно при решении так-  
тических задач, когда отсутствует фаза проектирования.  
Он свободный  
Python может использоваться и распространяться совершенно бесплатно. Как   
и в случае с другими открытыми программными продуктами, такими как Tcl,   
Perl, Linux и Apache, вы сможете получить в Интернете полные исходные тек-  
сты реализации Python. Нет никаких ограничений на его копирование, встра-  
ивание в свои системы или распространение в составе ваших продуктов. Фак-  
тически вы сможете даже продавать исходные тексты Python, если появится   
такое желание.  
Но «свободный» не означает «не поддерживается». Напротив, сообщество   
сторонников Python в Интернете отвечает на вопросы пользователей со ско-  
ростью, которой могли бы позавидовать большинство разработчиков коммер-  
ческих продуктов. Кроме того, свободное распространение исходных текстов   
Python способствует расширению команды экспертов по реализации. И хотя   
предоставляемая возможность изучать или изменять реализацию языка про-  
граммирования не у всех вызывает восторг, тем не менее, наличие последней   
инстанции в виде исходных текстов придает уверенность. Вы уже не зависите   
от прихотей коммерческого производителя – в вашем распоряжении находит-  
ся полный комплект исчерпывающей документации.  
Как уже упоминалось выше, разработка Python ведется сообществом, усилия   
которого координируются в основном через Интернет. В состав сообщества   
входит создатель Python – Гвидо ван Россум (Guido van Rossum), получивший   
официальное звание Пожизненного Великодушного Диктатора (Benevolent   
Dictator for Life, BDFL) Python, плюс тысячи других разработчиков. Измене-  
ния в языке принимаются только после прохождения формальной процедуры

54   
Глава 1. Python в вопросах и ответах   
(известной как «программа совершенствования продукта», или PEP) и тща-  
тельно проверяются формальной системой тестирования и самим Пожизнен-  
ным Диктатором. Это обеспечивает большую степень консерватизма Python   
в отношении изменений, по сравнению с некоторыми другими языками про-  
граммирования.  
Он переносим  
Стандартная реализация языка Python написана на переносимом ANSI C,   
благодаря чему он компилируется и работает практически на всех основных   
платформах. Например, программы на языке Python могут выполняться на са-  
мом широком спектре устройств, начиная от наладонных компьютеров (PDA)   
и заканчивая суперкомпьютерами. Ниже приводится далеко неполный список   
операционных систем и устройств, где можно использовать Python:  
 •  
Операционные системы Linux и UNIX  
 •  
Microsoft �indows и DOS (все современные версии)  
 •  
Mac OS (обе разновидности: OS X и Classic)  
 •  
BeOS, OS/2, VMS и QNX  
 •  
Системы реального времени, такие как Vx�orks  
 •  
Суперкомпьютеры Cray и ЭВМ производства компании IBM  
 •  
Наладонные компьютеры, работающие под управлением PalmOS, PocketPC   
или Linux  
 •  
Сотовые телефоны, работающие под управлением операционных систем   
Symbian и �indows Mobile  
 •  
Игровые консоли и iPod  
 •  
И многие другие  
Помимо самого интерпретатора языка в составе Python распространяется   
стандартная библиотека модулей, которая также реализована переносимым   
способом. Кроме того, программы на языке Python компилируются в перено-  
симый байт-код, который одинаково хорошо работает на любых платформах,   
где установлена совместимая версия Python (подробнее об этом будет рассказы-  
ваться в следующей главе).  
Все это означает, что программы на языке Python, использующие основные   
возможности языка и стандартные библиотеки, будут работать одинаково и в   
Linux, и в �indows, и в любых других операционных системах, где установлен   
интерпретатор Python. В большинстве реализаций Python под определенные   
операционные системы имеется также поддержка специфических механиз-  
мов этих систем (например, поддержка COM в �indows), но ядро языка Python   
и библиотеки работают совершенно одинаково в любой системе. Как уже го-  
ворилось выше, Python включает в себя средства создания графического ин-  
терфейса Tk GUI под названием tkinter (Tkinter в Python 2.6), что позволяет   
программам на языке Python создавать графический интерфейс, совместимый   
со всеми основными графическими платформами без индивидуальной про-  
граммной настройки.

В чем сильные стороны Python?   
55  
Он мощный  
С точки зрения функциональных возможностей Python можно назвать гибри-  
Python можно назвать гибри-  
 можно назвать гибри-  
дом. Его инструментальные средства укладываются в диапазон между тради-  
ционными языками сценариев (такими как Tcl, Scheme и Perl) и языками раз-  
работки программных систем (такими как C, C++ и Java). Python обеспечивает   
простоту и непринужденность языка сценариев и мощь, которую обычно мож-  
но найти в компилирующих языках. Превышая возможности других языков   
сценариев, такая комбинация делает Python удобным средством разработки   
крупномасштабных проектов. Для предварительного ознакомления ниже при-  
водится список основных возможностей, которые есть в арсенале Python:  
Динамическая типизация  
Python сам следит за типами объектов, используемых в программе, благо-  
 сам следит за типами объектов, используемых в программе, благо-  
даря чему не требуется писать длинные и сложные объявления в программ-  
ном коде. В действительности, как вы увидите в главе 6, в языке Python   
вообще отсутствуют понятие типа и необходимость объявления перемен-  
ных. Так как программный код на языке Python не стеснен рамками типов   
данных, он автоматически может обрабатывать целый диапазон объектов.  
Автоматическое управление памятью  
Python автоматически распределяет память под объекты и освобождает ее   
(«сборка мусора»), когда объекты становятся ненужными. Большинство   
объектов могут увеличивать и уменьшать занимаемый объем памяти по   
мере необходимости. Как вы узнаете, Python сам производит все низкоуров-  
невые операции с памятью, поэтому вам не придется беспокоиться об этом.  
Модульное программирование  
Для создания крупных систем Python предоставляет такие возможности,   
как модули, классы и исключения. Они позволяют разбить систему на со-  
ставляющие, применять ООП для создания программного кода многократно-  
го пользования и элегантно обрабатывать возникающие события и ошибки.  
Встроенные типы объектов  
Python предоставляет наиболее типичные структуры данных, такие как   
списки, словари и строки, в виде особенностей, присущих самому языку   
программирования. Как вы увидите позднее, эти типы отличаются вы-  
сокой гибкостью и удобством. Например, встроенные объекты могут рас-  
ширяться и сжиматься по мере необходимости, могут комбинироваться   
друг с другом для представления данных со сложной структурой и многое   
другое.  
Встроенные инструменты  
Для работы со всеми этими типами объектов в составе Python имеются   
мощные и стандартные средства, включая такие операции, как конкатена-  
ция (объединение коллекций), получение срезов (извлечение части коллек-  
ции), сортировка, отображение и многое другое.  
Библиотеки утилит  
Для выполнения более узких задач в состав Python также входит большая   
коллекция библиотечных инструментов, которые поддерживают практиче-

56   
Глава 1. Python в вопросах и ответах   
ски все, что только может потребоваться, – от поиска с использованием ре-  
гулярных выражений до работы в сети. Библиотечные инструменты языка   
Python – это то место, где выполняется большая часть операций.  
Утилиты сторонних разработчиков   
Python – это открытый программный продукт и поэтому разработчики   
могут создавать свои предварительно скомпилированные инструменты   
поддержки задач, решить которые внутренними средствами невозможно.   
В Сети вы найдете свободную реализацию поддержки COM, средств для ра-  
боты с изображениями, распределенных объектов CORBA, XML, механиз-  
CORBA, XML, механиз-  
, XML, механиз-  
мов доступа к базам данных и многое другое.  
Несмотря на широкие возможности, Python имеет чрезвычайно простой син-  
таксис и архитектуру. В результате мы имеем мощный инструмент программи-  
рования, обладающий простотой и удобством, присущими языкам сценариев.  
Он соединяемый  
Программы на языке Python с легкостью могут «склеиваться» с компонента-  
ми, написанными на других языках программирования. Например, приклад-  
ной интерфейс C API в Python позволяет программам на языке C вызывать   
и быть вызываемыми из программ на языке Python. Из этого следует, что вы   
можете расширять возможности программ на языке Python и использовать   
программный код на языке Python в других языковых средах и системах.  
Возможность смешивать Python с библиотеками, написанными на таких язы-  
ках, как C или C++, например, превращает его в удобный язык для создания   
интерфейсов к этим библиотекам и в средство настройки программных про-  
дуктов. Как уже говорилось выше, все это делает Python прекрасным сред-  
ством разработки прототипов – система может быть сначала реализована на   
языке Python, чтобы повысить скорость разработки, а позднее в зависимости   
от требований к производительности системы по частям перенесена на язык C.  
Он удобен  
Чтобы запустить программу на языке Python, достаточно просто ввести ее   
имя. Не требуется выполнять промежуточную компиляцию и связывание, как   
это делается в языках программирования, подобных C или C++. Интерпре-  
татор Python немедленно выполняет программу, что позволяет производить   
программирование в интерактивном режиме и получать результаты сразу же   
после внесения изменений – в большинстве случаев вы сможете наблюдать эф-  
фект изменения программы с той скоростью, с которой вы вводите изменения   
с клавиатуры.  
Безусловно, скорость разработки – это лишь один из аспектов удобства Python.   
Кроме того, он обеспечивает чрезвычайно простой синтаксис и набор мощных   
встроенных инструментов. Поэтому некоторые даже называют Python «испол-  
няемым псевдокодом». Поскольку большая часть сложностей ликвидируется   
другими инструментами, программы на языке Python проще, меньше и гибче   
эквивалентных им программ, написанных на таких языках, как C, C++ и Java!

Какими преимуществами обладает Python перед языком X?   
57  
Он прост в изучении  
Это самый важный аспект данной книги: по сравнению с другими языками   
программирования базовый язык Python очень легко запоминается. В дей-  
ствительности вы сможете писать на языке Python более или менее значимые   
программы уже через несколько дней (или даже через несколько часов, если вы   
уже опытный программист). Это отличная новость для разработчиков, стремя-  
щихся изучить язык для применения его в своей работе, а также для конечных   
пользователей, которые применяют Python для настройки или управления   
программным продуктом.   
Сегодня многие системы исходят из того, что конечные пользователи могут   
быстро изучить Python в достаточной степени, чтобы самостоятельно создать   
свой собственный программный код настройки системы при незначительной   
поддержке со стороны разработчика. И хотя в Python имеются сложные ин-  
струменты программирования, основа языка по-прежнему остается простой   
для изучения как начинающими, так и опытными программистами.  
Он назван в честь Монти Пайтона  
Это не имеет отношения к технической стороне дела, но похоже, что эта тай-  
на, которую я собираюсь открыть, на удивление хорошо охраняется. Несмотря   
на то, что на эмблеме Python изображена рептилия, правда состоит в том, что   
создатель Python, Гвидо ван Россум, назвал свое детище в честь комедийно-  
го сериала «Летающий цирк Монти Пайтона» (Monty Python’s Flying Circus),   
который транслировался по телеканалу BBC. Он большой поклонник Монти   
Пайтона, как и многие программисты (похоже, что между разработкой про-  
граммного обеспечения и цирком есть что-то общее).  
Это обстоятельство неизбежно добавляет юмора в примеры программного кода   
на языке Python. Например, традиционные имена переменных «foo» и «bar»,   
в языке Python превратились в «spam» и «eggs». Встречающиеся иногда имена   
«Brian», «ni» и «shrubbery», точно также появились благодаря своим тезкам.   
Это даже оказывает влияние на сообщество в целом: дискуссии на конференци-  
ях по языку Python обычно именуются «Испанская инквизиция».  
Все это, конечно, очень забавно, если вы знакомы с сериалом, в противном слу-  
чае это кажется непонятным. Вам не требуется знать сериал, чтобы понимать   
примеры, где используются ссылки на Монти Пайтона (включая многие при-  
меры в этой книге), но, по крайней мере, вы теперь знаете, откуда что берется.  
Какими преимуществами обладает Python   
перед языком X?   
Наконец, чтобы разместить язык Python среди уже, возможно, известных вам   
понятий, сравним Python с другими языками программирования, такими как   
Perl, Tcl и Java. Ранее мы уже говорили о проблеме производительности, поэто-  
, Tcl и Java. Ранее мы уже говорили о проблеме производительности, поэто-  
Java. Ранее мы уже говорили о проблеме производительности, поэто-  
. Ранее мы уже говорили о проблеме производительности, поэто-  
му здесь мы сосредоточим свое внимание на функциональных возможностях.   
Другие языки программирования также являются достаточно полезными ин-

58   
Глава 1. Python в вопросах и ответах   
струментами, чтобы знать и использовать их, но многие программисты нахо-  
дят, что Python:  
 •  
Имеет более широкие возможности, чем Tcl. Язык Python поддерживает   
«программирование в целом», что делает его применимым для разработки   
крупных систем.  
 •  
Имеет более четкий синтаксис и более простую архитектуру, чем Perl, что   
делает программный код более удобочитаемым, простым в сопровождении   
и снижает вероятность появления ошибок.  
 •  
Проще и удобнее, чем Java. Python – это язык сценариев, а Java унаследова-  
Python – это язык сценариев, а Java унаследова-  
 – это язык сценариев, а Java унаследова-  
Java унаследова-  
 унаследова-  
ла сложный синтаксис от таких языков программирования, как C++.  
 •  
Проще и удобнее, чем C++, но нередко он не может конкурировать с C++,   
поскольку, будучи языком сценариев, Python предназначен для решения   
другого круга задач.  
 •  
Более мощный и более переносимый, чем Visual Basic. Открытая природа   
Python также означает, что нет какой-то отдельной компании, которая его   
контролирует.  
 •  
Более удобочитаемый и более универсальный, чем PHP. Иногда Python ис-  
Python ис-  
 ис-  
пользуется для создания веб-сайтов, но он способен решать гораздо более   
широкий круг задач, от управления роботами до создания анимационных   
фильмов.  
 •  
Более зрелый и имеет более ясный синтаксис, чем Ruby. В отличие от Ruby   
и Java, объектно-ориентированный стиль программирования является не-  
обязательным в Python – он не вынуждает использовать ООП в проектах,   
где этот стиль неприменим.  
 •  
Обладает динамическими особенностями таких языков, как SmallTalk   
и Lisp, но имеет более простой и традиционный синтаксис, доступный как   
для разработчиков, так и для конечных пользователей настраиваемых си-  
стем.  
Многие считают, что Python, по сравнению с другими современными языками   
сценариев, гораздо лучше подходит для программ, которые делают нечто боль-  
шее, чем простое сканирование текстовых файлов и код которых, возможно,   
потребуется читать другим людям (и может быть, даже вам!). Кроме того, если   
от вашего приложения не требуется наивысшая производительность, Python   
способен составить конкуренцию таким языкам программирования, как C,   
C++ и Java: программный код на языке Python проще писать, отлаживать и со-  
Java: программный код на языке Python проще писать, отлаживать и со-  
: программный код на языке Python проще писать, отлаживать и со-  
провождать.  
Безусловно, автор является горячим поклонником Python с 1992 года, поэтому   
воспринимайте эти комментарии по своему усмотрению. Однако они в действи-  
тельности отражают опыт многих программистов, которые потратили немало   
времени на исследование возможностей Python.  
В заключение  
Этот раздел завершает рекламную часть книги. В этой главе мы рассмотрели   
некоторые из причин, по которым люди выбирают Python для программирова-  
Python для программирова-  
 для программирова-  
ния своих задач. Здесь также было показано, как он используется, и приведены

Закрепление пройденного   
59  
представительные примеры тех, кем он используется в настоящее время. Моя   
цель состоит в том, чтобы обучить вас языку Python, а не продать его. Лучший   
способ создать собственное мнение о языке – это опробовать его в действии,   
поэтому остальная часть книги целиком и полностью будет сфокусирована на   
описании языка, который здесь был представлен.  
Следующие две главы могут рассматриваться как техническое введение   
в язык. В этих главах мы узнаем, как запускаются программы на языке Py-  
Py-  
thon, коротко рассмотрим модель исполнения байт-кода и получим основные   
сведения об организации файлов модулей, в которых хранится программный   
код. Цель этих глав состоит в том, чтобы дать вам объем информации, доста-  
точный для запуска примеров и выполнения упражнений в остальной части   
книги. Мы фактически не будем касаться вопросов программирования до гла-  
вы 4, но прежде чем перейти к нему, вы определенно получите все необходимые   
начальные сведения.  
Закрепление пройденного  
Контрольные вопросы  
Каждая глава в этом издании книги будет завершаться серией коротких кон-  
трольных вопросов, которые помогут вам закрепить в памяти ключевые кон-  
цепции. Ответы на вопросы следуют ниже, и вы можете прочитать эти ответы   
сразу, как только столкнетесь с затруднениями. Помимо контрольных вопро-  
сов в конце каждой части вы найдете упражнения, предназначенные для того,   
чтобы помочь вам программировать на языке Python. Итак, перед вами пер-  
Python. Итак, перед вами пер-  
. Итак, перед вами пер-  
вый тест. Удачи!  
1. Назовите шесть основных причин, по которым программисты выбирают   
Python?  
2. Назовите четыре известные компании или организации, использующие   
Python.  
3. Почему бы вы не хотели использовать Python в приложениях?  
4. Какие задачи можно решать с помощью Python?  
5. Какой важный результат можно получить с помощью инструкции import   
this?  
6. Почему слово «spam» так часто используется в примерах программного   
кода на языке Python?  
7. Какой ваш любимый цвет?  
Ответы  
Ну, как дела? Ниже приводятся ответы, которые подготовил я, хотя на неко-  
торые вопросы существует несколько правильных ответов. Напомню еще раз,   
даже если вы абсолютно уверены в правильности своих ответов, я советую про-  
читать мои ответы, хотя бы ради того, чтобы получить некоторые дополнитель-  
ные сведения. Если мои ответы кажутся вам бессмысленными, прочитайте   
текст главы еще раз.

60   
Глава 1. Python в вопросах и ответах   
1. Качество программного обеспечения, скорость разработки, переносимость   
программ, библиотеки поддержки, интеграция компонентов и просто удо-  
вольствие. Из этих шести причин качество и скорость разработки являются   
наиболее существенными при выборе Python.  
2. Google, Industrial Light & Magic, EVE Online, Jet Propulsion Labs, Maya,   
ESRI и многие другие. Практически каждая организация, занимающаяся   
разработкой программного обеспечения так или иначе использует Python   
как для решения долговременных, стратегических задач проектирования,   
так и для решения краткосрочных тактических задач, таких как тестиро-  
вание и системное администрирование.  
3. Основным недостатком Python является невысокая производительность,   
программы на языке Python не могут выполняться так же быстро, как про-  
Python не могут выполняться так же быстро, как про-  
 не могут выполняться так же быстро, как про-  
граммы на полностью компилируемых языках, таких как C и С++. С дру-  
C и С++. С дру-  
 и С++. С дру-  
гой стороны, для большинства применений он обладает достаточно высокой   
скоростью выполнения и обычно программный код на языке Python рабо-  
Python рабо-  
 рабо-  
тает со скоростью, близкой к скорости языка C, потому что интерпретатор   
вызывает встроенный в него программный код, написанный на языке C.   
Если скорость выполнения имеет критическое значение, отдельные части   
приложения можно реализовать на языке C, в виде расширений.  
4. Вы можете использовать Python для любых задач, которые можно решить   
с помощью компьютера, – от реализации веб-сайта и игровых программ до   
управления роботами и космическими кораблями.  
5. Инструкция import this активизирует «пасхальное яйцо», скрытое в недрах   
Python, которое отображает некоторые принципы проектирования, лежа-  
, которое отображает некоторые принципы проектирования, лежа-  
щие в основе языка. Как запустить эту инструкцию, вы узнаете в следую-  
щей главе.  
6. Слово «spam» взято из известной пародии Монти Пайтона, где герои сери-  
spam» взято из известной пародии Монти Пайтона, где герои сери-  
» взято из известной пародии Монти Пайтона, где герои сери-  
ала пытаются заказать блюдо в кафетерии, а их заглушает хор викингов,   
поющих о консервах фирмы Spam. Ах да! Это еще и типичное имя перемен-  
Spam. Ах да! Это еще и типичное имя перемен-  
. Ах да! Это еще и типичное имя перемен-  
ной, которое используется в сценариях на языке Python...  
7. Голубой. Нет, желтый!  
Программирование на языке Python –   
это технический процесс, а не искусство  
Когда в начале 1990-х годов Python впервые появился на сцене про-  
граммного обеспечения, это породило что-то вроде конфликта между   
сторонниками языка Python и другого популярного языка сценариев –   
Perl. Лично я считаю такие дебаты пустыми и бессмысленными – разра-  
ботчики достаточно умны, чтобы самостоятельно сделать выводы. Одна-  
ко в моей преподавательской практике мне часто приходится слышать   
вопросы на эту тему, поэтому я считаю необходимым сказать несколько   
слов по этому поводу.

Закрепление пройденного   
61  
В двух словах: все,  что  можно  сделать  на  Perl,  можно  сделать  и  на   
Python, но при использовании Python вы еще сможете прочитать свой   
программный код. Для большинства удобочитаемость программного кода   
на языке Python означает возможность многократного его использования   
и простоту сопровождения, что делает Python отличным выбором для   
написания программ, которые не попадут в разряд написанных и сразу   
после отладки выброшенных. Программный код на языке Perl легко пи-  
сать, но сложно читать. Учитывая, что период жизни большинства про-  
грамм длится много дольше, чем период их создания, многие усматрива-  
ют в Python более эффективный инструмент программирования.  
Если говорить более развернуто, история конфликта отражает опыт   
проектировщиков двух языков программирования и подчеркивает не-  
которые из основных причин, по которым программисты отдают пред-  
почтение языку Python. Создатель языка Python – математик по об-  
разованию, и потому он создал язык, обладающий высокой степенью   
однородности – его синтаксис и набор возможностей отличаются удиви-  
тельной согласованностью. Более того, если говорить математическими   
терминами, язык Python обладает ортогональной архитектурой – боль-  
шая часть выразительных возможностей языка следует из небольшого   
числа базовых концепций. Например, как только программист схваты-  
вает суть полиморфизма в Python, все остальное легко достраивается.  
Создатель языка Perl, напротив – лингвист, поэтому и язык отражает   
его профессиональный опыт. В языке Perl одну и ту же задачу можно   
решить множеством способов, а языковые конструкции взаимодейству-  
ют между собой контекстно-зависимым, порой трудноуловимым спосо-  
бом, во многом напоминая естественный язык общения. Как известно,   
девизом языка Perl является выражение: «Всякую задачу можно решить   
более чем одним способом». Учитывая это, можно сказать, что язык Perl   
и сообщество его пользователей исторически стремились к свободе вы-  
ражения мыслей при создании программного кода. Программный код   
одного программиста может радикально отличаться от программного   
кода другого. И правда, искусство создания уникального хитросплете-  
ния инструкций всегда было предметом гордости программистов на Perl.  
Однако любой, кто когда-либо занимался сопровождением программно-  
го кода, скажет вам, что свобода самовыражения хороша для искусства,   
но не для технологического процесса. В технологии нам требуются ми-  
нимальный набор возможностей и высокая степень предсказуемости.   
Свобода выражения мыслей в технологии может превратить процесс со-  
провождения в непрекращающийся кошмар. По секрету говоря, уже не   
от одного пользователя Perl я слышал, что проще написать свой код, чем   
внести изменения в чужой.  
Когда художник пишет картину или ваяет скульптуру, он выражает этим   
исключительно себя, свои эстетические побуждения. Он не предполагает,   
что картина или скульптура будет изменяться другим художником. Это   
важное различие между искусством и технологическим процессом.

62   
Глава 1. Python в вопросах и ответах   
Когда программист пишет сценарий, он пишет его не для себя самого.   
Более того, сценарий пишется даже не для компьютера. Хороший про-  
граммист знает, что свой программный код он пишет для другого чело-  
века, который будет вынужден читать его в ходе сопровождения и ис-  
пользования. Если этот человек не сможет понять сценарий, сценарий   
станет практически бесполезным в реальной жизни.  
Многие находят в этом самое четкое отличие Python от других языков   
сценариев, подобных языку Perl. Синтаксическая модель Python вы-  
нуждает пользователя писать удобочитаемый программный код, поэ-  
тому программы на языке Python лучше вписываются в полный цикл   
разработки программного обеспечения. А такие свойства Python, как   
ограниченное число способов взаимодействия, единообразие, законо-  
мерность и непротиворечивость, способствуют появлению программно-  
го кода, который будет использоваться после того, как будет написан.  
В конечном счете, в центре внимания языка Python находится качество   
программного кода, что само по себе повышает производительность   
программиста и приводит к появлению у него чувства удовлетворенно-  
сти. Программисты, использующие язык Python, могут быть не менее   
творческими натурами и, как мы увидим позднее, в некоторых случа-  
ях этот язык также способен предложить несколько способов решения   
одной и той же задачи. Тем не менее, в своей основе Python стимулирует   
ведение разработки способами, часто недоступными в других языках   
сценариев.  
По крайней мере, это общее мнение многих из тех, кто принял Python   
на вооружение. Конечно, у вас сложится собственное суждение по мере   
изучения Python. А приступить к изучению мы сможем в следующей   
главе.

Глава 2.  
   
Как Python запускает программы  
В этой и в следующей главе будут коротко рассмотрены вопросы исполнения   
программ – как программы запускаются человеком и как Python выполняет   
их. В этой главе мы рассмотрим интерпретатор Python. После этого в главе 3   
будет показано, как создавать и запускать свои собственные программы.  
Порядок запуска программ в любом случае зависит от типа платформы   
и какие-то сведения из этой главы могут оказаться неприменимы к платформе,   
используемой вами, поэтому вы можете просто пропускать разделы, которые   
не относятся к вам. Точно так же опытные пользователи, которым уже прихо-  
дилось использовать подобные инструменты в прошлом и которые стремятся   
побыстрее добраться до самого языка, могут пропустить эту главу, оставив ее   
«для ознакомления в будущем». А со всеми остальными мы попробуем разо-  
браться, как запускать некоторый программный код.  
Введение в интерпретатор Python  
До сих пор я говорил о Python в основном как о языке программирования. Но   
в текущей реализации это еще и программный пакет , который называется ин-  
терпретатором. Интерпретатор – это такой модуль, который исполняет дру-  
гие программы. Когда вы пишете код на языке Python, интерпретатор Python   
читает вашу программу и выполняет составляющие ее инструкции. По сути   
дела интерпретатор – это слой программной логики между вашим программ-  
ным кодом и аппаратурой вашего компьютера.  
В процессе установки пакета Python на компьютер создается ряд программ-  
ных компонентов – как минимум, интерпретатор и библиотека поддержки.   
В зависимости от особенностей использования интерпретатор Python может   
иметь вид исполняемой программы или набора библиотек, связанных с другой   
программой. В зависимости от версии Python сам интерпретатор может быть   
реализован как программа на языке C, как набор классов Java или в каком-  
либо другом виде. Независимо от используемой разновидности Python ваш   
программный код на этом языке всегда будет выполняться этим интерпрета-  
тором. А чтобы обеспечить такую возможность, вы должны установить интер-  
претатор Python на свой компьютер.

64   
Глава 2. Как Python запускает программы   
Процедура установки Python отличается для разных платформ и подробно   
описывается в приложении A. В двух словах:  
 •  
Пользователи �indows должны получить и запустить инсталляционный   
исполняемый файл, который произведет установку Python на компьютер.   
Для этого нужно просто дважды щелкнуть на инсталляционном файле   
и отвечать «�es» (Да) или «Next» (Далее) на все вопросы.  
 •  
В Linux или в Mac OS вполне возможно, что Python уже установлен и готов   
к использованию, поскольку он является стандартным компонентом этих   
операционных систем.  
 •  
В отдельных версиях Linux и Mac OS (а также в большинстве версий UNIX)   
Python может собираться из исходных текстов.  
 •  
Пользователи Linux могут также отыскать файлы RPM, а пользователи   
Mac OS – установочные пакеты для этой операционной системы.  
 •  
Процедура установки на других платформах зависит от этих платформ. На-  
пример, Python присутствует также в сотовых телефонах, игровых консо-  
лях и в проигрывателе iPod, но процедуры установки Python на эти устрой-  
ства слишком отличаются, чтобы описывать их здесь.  
Дистрибутив Python можно получить на странице загрузок сайта проекта. Его   
можно также получить по другим каналам распространения программного   
обеспечения. Но имейте в виду, прежде чем приступать к установке, вы долж-  
ны убедиться, что Python не был ранее установлен на ваш компьютер. Если   
вы пользуетесь операционной системой �indows, обычно Python можно найти   
в меню «Start» (Пуск), как показано на рис. 2.1 (эти пункты меню будут рассма-  
Start» (Пуск), как показано на рис. 2.1 (эти пункты меню будут рассма-  
» (Пуск), как показано на рис. 2.1 (эти пункты меню будут рассма-  
триваться в следующей главе). В операционных системах Linux и UNIX Python   
обычно находится в дереве каталогов /usr.  
Поскольку процедура установки сильно зависит от используемой платформы,   
мы здесь прервем рассказ о ней. За дополнительной информацией о ней обра-  
щайтесь к приложению A. В целях этой и следующей главы я буду исходить из   
предположения, что Python уже установлен и готов к работе.  
Выполнение программы  
Что стоит за словами «написать и запустить программу на языке Python» за-  
висит от того, как вы смотрите на эту задачу – как программист или как ин-  
терпретатор Python. Обе точки зрения определяют свой взгляд на программи-  
рование.  
С точки зрения программиста  
Программа на языке Python, в самой простой форме, – это обычный текстовый   
файл, содержащий инструкции Python. Например, следующий файл, с име-  
нем script0.py, – это один из простейших сценариев на языке Python, который   
только можно придумать, но его официально можно назвать программой на   
языке Python:  
print(‘hello world’)  
print(2 \*\* 100)

Выполнение программы   
65  
Этот файл содержит две инструкции print, которые просто выводят строку   
(текст в кавычках) и результат числового выражения (2 в степени 100) в вы-  
ходной поток. Не надо сейчас стараться вникнуть в синтаксис языка – в этой   
главе нас интересует лишь сам порядок запуска программ. Позднее я расскажу   
Рис. 2.1. Когда Python установлен в Windows, его можно найти в меню «Start»   
(Пуск). Набор пунктов меню может немного отличаться, в зависимости от   
версии, но пункт IDLE запускает среду разработки с графическим интерфей-  
сом, а пункт Python запускает простой интерактивный сеанс работы с ин-  
терпретатором. Кроме того, здесь же можно увидеть пункты вызова стан-  
дартного справочного руководства и запуска механизма документирования   
PyDoc (пункт Module Docs)

66   
Глава 2. Как Python запускает программы   
об инструкции print и объясню, почему можно возвести число 2 в степень 100,   
не опасаясь получить ошибку переполнения.  
Создать такой файл можно с помощью любого текстового редактора. По об-  
щепринятым соглашениям файлы с программами на языке Python должны   
иметь расширение .py – с технической точки зрения, это требование должно   
выполняться только для «импортируемых» файлов, как будет показано позд-  
нее в этой книге, но большинству файлов с программами на языке Python да-  
ются имена с расширением .py для единообразия.  
После того как инструкции будут введены в текстовый файл, можно потре-  
бовать от Python выполнить его, то есть просто выполнить все инструкции   
в файле одну за другой от начала и до конца. Как будет показано в следующей   
главе, вы можете запускать программы, щелкая на их пиктограммах или дру-  
гими стандартными способами. Если при выполнении файла все пройдет как   
надо, вы увидите результаты работы двух инструкций print где-то на экране   
своего компьютера – обычно это происходит в том же окне, где производился   
запуск программы:  
hello world  
1267650600228229401496703205376  
Например, ниже показано, что произошло, когда я попытался запустить этот   
сценарий в командной строке DOS на ноутбуке, где установлена операционная   
система �indows (обычно эту программу можно найти в меню Accessories (Стан-  
дартные) под названием Command Prompt (Командная строка)), чтобы убедиться, что   
я не допустил никаких опечаток:  
C:\temp> python script0.py  
hello world  
1267650600228229401496703205376  
Мы только что запустили сценарий, который вывел строку и число. Вероятно,   
мы не получим награды на конкурсе по программированию за этот сценарий,   
но его вполне достаточно, чтобы понять основные принципы запуска программ.  
С точки зрения Python  
Краткое описание, приведенное в предыдущем разделе, является довольно   
стандартным для языков сценариев, и это обычно все, что необходимо знать   
программисту. Вы вводите программный код в текстовый файл, а затем запу-  
скаете этот файл с помощью интерпретатора. Однако, когда вы говорите интер-  
претатору «вперед», за кулисами много чего происходит. Хотя знание внутрен-  
него устройства Python и не требуется для овладения навыками программиро-  
вания на этом языке, тем не менее, понимание того, как производится выпол-  
нение программ, поможет вам увидеть всю картину в целом.  
Когда интерпретатор Python получает от вас команду запустить сценарий, он   
выполняет несколько промежуточных действий, прежде чем ваш программ-  
ный код начнет «скрипеть колесами». В частности, сценарий сначала будет   
скомпилирован в нечто под названием «байт-код», а затем передан механизму,   
известному под названием «виртуальная машина».

Выполнение программы   
67  
Компиляция в байт-код  
Когда вы запускаете программу, практически незаметно для вас Python сна-  
чала компилирует ваш исходный текст (инструкции в файле) в формат, из-  
вестный под названием байт-код. Компиляция – это просто этап перевода   
программы, а байт-код – это низкоуровневое, платформонезависимое пред-  
ставление исходного текста программы. Интерпретатор Python транслирует   
каждую исходную инструкцию в группы инструкций байт-кода, разбивая ее   
на отдельные составляющие. Такая трансляция в байт-код производится для   
повышения скорости – байт-код выполняется намного быстрее, чем исходные   
инструкции в текстовом файле.  
В предыдущем абзаце вы могли заметить фразу – практически незаметно для   
вас. Если интерпретатор Python на вашем компьютере обладает правом запи-  
си, он будет сохранять байт-код вашей программы в виде файла с расширением   
.pyc (.pyc – это компилированный исходный файл .py). Вы будете обнаруживать   
эти файлы после запуска программ по соседству с файлами, содержащими ис-  
ходные тексты (то есть в том же каталоге).  
Интерпретатор сохраняет байт-код для ускорения запуска программ. В сле-  
дующий раз, когда вы попробуете запустить свою программу, Python загрузит   
файл .pyc и минует этап компиляции – при условии, что исходный текст про-  
граммы не изменялся с момента последней компиляции. Чтобы определить,   
необходимо ли выполнять перекомпиляцию, Python автоматически сравнит   
время последнего изменения файла с исходным текстом и файла с байт-кодом.   
Если исходный текст сохранялся на диск после компиляции, при следующем   
его запуске интерпретатор автоматически выполнит повторную компиляцию   
программы.  
Если интерпретатор окажется не в состоянии записать файл с байт-кодом на   
диск, программа от этого не пострадает, просто байт-код будет сгенерирован   
в памяти и исчезнет по завершении программы.1 Однако поскольку файлы .pyc   
повышают скорость запуска программы, вам может потребоваться иметь воз-  
можность сохранять их, особенно для больших программ. Кроме того, файлы   
с байт-кодом – это еще один из способов распространения программ на языке   
Python. Интерпретатор запустит файл .pyc, даже если нет оригинальных фай-  
лов с исходными текстами. (В разделе «Фиксированные двоичные файлы» опи-  
сывается еще один способ распространения программ).  
Виртуальная машина Python (PVM)  
Как только программа будет скомпилирована в байт-код (или байт-код будет   
загружен из существующих файлов .pyc), он передается механизму под назва-  
нием виртуальная машина Python (PVM – для любителей аббревиатур). Аббре-  
виатура PVM выглядит более внушительно, чем то, что за ней стоит на самом   
деле, – это не отдельная программа, которую требуется устанавливать. Факти-  
чески PVM – это просто большой цикл, который выполняет перебор инструк-  
1   
Строго говоря, байт-код сохраняется только для импортируемых файлов, но не для   
файла самой программы. Об импорте мы поговорим в главе 3 и снова вернемся к нему   
в части V. Байт-код также никогда не сохраняется для инструкций, введенных в ин-  
терактивном режиме, который описывается в главе 3.

68   
Глава 2. Как Python запускает программы   
ций в байт-коде, одну за одной, и выполняет соответствующие им операции.   
PVM – это механизм времени выполнения, она всегда присутствует в составе   
системы Python и это тот самый программный компонент, который выполняет   
ваши сценарии. Формально – это последняя составляющая того, что называют   
«интерпретатором Python».  
На рис. 2.2 показана последовательность действий, которая описывается здесь.   
Не забывайте, что все эти сложности преднамеренно спрятаны от программи-  
стов. Компиляция в байт-код производится автоматически, а PVM – это всего   
лишь часть системы Python, которую вы установили на свой компьютер. По-  
вторю еще раз, что программисты просто создают программный код на языке   
Python и запускают файлы с инструкциями.  
Исходный текст  
Выполнение  
PVM  
m.pyc  
m.py  
Рис. 2.2. Традиционная модель выполнения программ на языке Python:   
исходный текст, который вводится программистом, транслируется в байт-  
код, который затем исполняется виртуальной машиной Python. Исходный   
текст автоматически компилируется и затем интерпретируется  
Производительность  
Читатели, имеющие опыт работы с компилирующими языками программиро-  
вания, такими как C и C++, могут заметить несколько отличий в модели вы-  
полнения Python. Первое, что бросается в глаза, – это отсутствие этапа сборки,   
или вызова утилиты «make»: программный код может запускаться сразу же,   
как только будет написан. Второе отличие: байт-код не является двоичным ма-  
шинным кодом (например, инструкциями для микропроцессора Intel). Байт-  
код – это внутреннее представление программ на языке Python.  
По этой причине программный код на языке Python не может выполняться   
так же быстро, как программный код на языке C или C++, о чем уже говори-  
лось в главе 1. Обход инструкций выполняет виртуальная машина, а не ми-  
кропроцессор, и чтобы выполнить байт-код, необходима дополнительная ин-  
терпретация, инструкции которого требуют на выполнение больше времени,   
чем машинные инструкции микропроцессора. С другой стороны, в отличие от   
классических интерпретаторов, здесь присутствует дополнительный этап ком-  
пиляции – интерпретатору не требуется всякий раз снова и снова анализиро-  
вать инструкции исходного текста. В результате Python способен обеспечить   
скорость выполнения где-то между традиционными компилирующими и тра-  
диционными интерпретирующими языками программирования. Подробнее   
о проблеме производительности рассказывается в главе 1.

Разновидности модели выполнения   
69  
Скорость разработки  
С другой стороны, в модели выполнения Python отсутствуют различия между   
средой разработки и средой выполнения. То есть системы, которые компили-  
руют и выполняют исходный текст, – это суть одна и та же система. Для чита-  
телей, имеющих опыт работы с традиционными компилирующими языками,   
это обстоятельство может иметь некоторое значение, но в Python компилятор   
всегда присутствует во время выполнения и является частью механизма, вы-  
полняющего программы.  
Это существенно увеличивает скорость разработки. Не нужно всякий раз вы-  
полнять компиляцию и связывание программ прежде чем запустить их, – вы   
просто вводите исходный текст и запускаете его. Это также придает языку не-  
который динамизм – вполне возможно, а нередко и удобно, когда программы   
на языке Python создаются и выполняются другими программами Python во   
время выполнения. Например, встроенные инструкции eval и exec принимают   
и выполняют строки, содержащие программный код на языке Python. Благо-  
hon. Благо-  
on. Благо-  
даря такой возможности Python может использоваться для настройки продук-  
тов – программный код Python может изменяться «на лету», а пользователи   
могут изменять части системы, написанные на языке Python, без необходимо-  
сти перекомпилировать систему целиком.  
Если смотреть с более фундаментальных позиций, то все, что имеется в Python,   
работает на этапе времени  выполнения – здесь полностью отсутствует этап   
предварительной компиляции, все что необходимо, производится во время   
выполнения программы. Сюда относятся даже такие операции, как создание   
функций и классов и связывание модулей. Эти события в более статичных язы-  
ках происходят перед выполнением, но в программах на языке Python проис-  
ходят во время выполнения. В результате процесс программирования приоб-  
ретает больший динамизм, чем тот, к которому привыкли некоторые читатели.  
Разновидности модели выполнения  
Прежде чем двинуться дальше, я должен заметить, что внутренний поток вы-  
полнения, описанный в предыдущем разделе, отражает современную стан-  
дартную реализацию интерпретатора Python, которая в действительности не   
является обязательным требованием самого языка Python. Вследствие этого   
модель выполнения склонна изменяться с течением времени. Фактически уже   
существуют системы, которые несколько меняют картину, представленную на   
рис. 2.2. Давайте потратим несколько минут, чтобы ознакомиться с наиболее   
заметными изменениями.  
Альтернативные реализации Python  
В то время когда я писал эту книгу, существовали три основные альтернатив-  
ные реализации языка Python – CPython, Jython и IronPython, а также не-  
сколько второстепенных реализаций, таких как Stackless Python. В двух сло-  
вах: CPython – это стандартная реализация, а все остальные создавались для   
специфических целей и задач. Все они реализуют один и тот же язык Python,   
но выполняют программы немного по-разному.

70   
Глава 2. Как Python запускает программы   
CPython  
Оригинальная и стандартная реализация языка Python обычно называется   
CPython, особенно когда необходимо подчеркнуть ее отличие от двух других   
альтернатив. Это название происходит из того факта, что реализация написана   
на переносимом языке ANSI C. Это тот самый Python, который вы загружаете   
с сайта http://www.python.org, получаете в составе дистрибутива ActivePython   
и который присутствует в большинстве систем Linux и Mac OS X. Если вы об-  
наружили у себя предварительно установленную версию Python, то более чем   
вероятно это будет CPython, – при условии, что ваша компания не использует   
какую-то специфическую версию.  
Если вы не предполагаете создавать приложения на Java или для платформы   
.NET, возможно, вам следует отдать предпочтение стандартной реализации   
CPython. Поскольку это эталонная реализация языка, она, как правило, рабо-  
тает быстрее, устойчивее и лучше, чем альтернативные системы. Рисунок 2.2   
отражает модель выполнения CPython.  
Jython  
Интерпретатор Jython (первоначальное название – JPython) – это альтерна-  
Jython (первоначальное название – JPython) – это альтерна-  
 (первоначальное название – JPython) – это альтерна-  
JPython) – это альтерна-  
) – это альтерна-  
тивная реализация языка Python, основная цель которой – тесная интеграция   
с языком программирования Java. Реализация Jython состоит из Java-классов,   
которые выполняют компиляцию программного кода на языке Python в байт-  
код Java и затем передают полученный байт-код виртуальной машине Java   
(Java Virtual Machine, JVM). Программист помещает инструкции на языке   
Python в текстовые файлы как обычно, а система Jython подменяет два рас-  
Jython подменяет два рас-  
 подменяет два рас-  
положенных на рис. 2.2 справа этапа на эквиваленты языка Java.  
Цель Jython состоит в том, чтобы позволить программам на языке Python   
управлять Java-приложениями, точно так же как CPython может управлять   
компонентами на языках C и C++. Эта реализация имеет бесшовную инте-  
C и C++. Эта реализация имеет бесшовную инте-  
 и C++. Эта реализация имеет бесшовную инте-  
грацию с Java. Поскольку программный код на языке Python транслируется   
в байт-код Java, во время выполнения он ведет себя точно так же, как насто-  
ящая программа на языке Java. Сценарии на языке Jython могут выступать   
в качестве апплетов и сервлетов, создавать графический интерфейс с использо-  
ванием механизмов Java и так далее. Более того, Jython обеспечивает поддерж-  
ку возможности импортировать и использовать Java-классы в программном   
коде Python. Тем не менее поскольку реализация Jython обеспечивает более   
низкую скорость выполнения и менее устойчива по сравнению с CPython, она   
представляет интерес скорее для разработчиков программ на языке Java, кото-  
рым необходим язык сценариев в качестве интерфейса к Java-коду.  
IronPython  
Третья (и к моменту написания этих строк самая новая) реализация язы-  
ка Python – это IronPython. Она предназначена для обеспечения интеграции   
программ Python с приложениями, созданными для работы в среде Microsoft   
.NET Framework операционной системы �indows, а также в Mono – открытом   
эквиваленте для операционной системы Linux. Платформа .NET и среда вы-  
полнения языка C# предназначены для обеспечения взаимодействий между   
программными объектами – независимо от используемого языка программи-  
рования, в духе более ранней модели COM компании Microsoft. Реализация

Разновидности модели выполнения   
71  
IronPython позволяет программам на языке Python играть роль как клиент-  
ских, так и серверных компонентов, доступных из других языков программи-  
рования .NET.  
Модель реализации IronPython очень напоминает Jython (и фактически разра-  
батывается одним и тем же автором) – она подменяет два этапа на рис. 2.2 спра-  
ва на эквиваленты среды .NET. Кроме того, как и Jython, основной интерес   
IronPython представляет для разработчиков, которым необходима интегра-  
ция Python с компонентами .NET. Поскольку разработка ведется компанией   
Microsoft, от IronPython, кроме всего прочего, можно было бы ожидать суще-  
ственной оптимизации производительности. К моменту написания этих строк   
реализация IronPython еще продолжала разрабатываться. За дополнительной   
информацией обращайтесь к ресурсам Python или попробуйте самостоятельно   
поискать в Сети.1  
Средства оптимизации скорости выполнения  
Все три реализации, CPython, Jython и IronPython, реализуют язык Python   
похожими способами: исходный программный код компилируют в байт-код   
и выполняют полученный байт-код с помощью соответствующей виртуальной   
машины. Но кроме них существуют и другие реализации, включая динамиче-  
ский компилятор Psyco и транслятор Shedskin C++, которые пытаются опти-  
мизировать основную модель выполнения. Знание этих реализаций не являет-  
ся для вас обязательным на этой стадии изучения языка Python, тем не менее,   
краткий обзор их реализации модели выполнения поможет пролить свет на   
модель выполнения в целом.  
Динамический компилятор Psyco  
Система Psyco – это не другая реализация языка Python, а компонент, рас-  
ширяющий модель выполнения байт-кода, что позволяет программам выпол-  
няться быстрее. В терминах схемы на рис. 2.2 Psyco – это расширение PVM,   
которое собирает и использует информацию о типах, чтобы транслировать   
части байт-кода программы в истинный двоичный машинный код, который   
выполняется гораздо быстрее. Для такой трансляции не требуется вносить из-  
менения в исходный программный код или производить дополнительную ком-  
пиляцию в ходе разработки.  
Грубо говоря, во время выполнения программы Psyco собирает информацию   
о типах объектов и затем эта информация используется для генерации вы-  
сокоэффективного машинного кода, оптимизированного для объектов этого   
типа. После этого произведенный машинный код замещает соответствующие   
участки байт-кода и тем самым увеличивает скорость выполнения программы.   
В результате при использовании Psyco существенно уменьшается общее время   
выполнения программы. В идеале некоторые участки программного кода под   
1   
Jython и IronPython – это полностью независимые реализации языка Python, кото-  
рые компилируют исходный программный код для различных архитектур времени   
выполнения. Из программ для CPython также можно получить доступ к программ-  
ным компонентам Java и .NET: например, системы JPype и Python for .NET позволя-  
ют коду, исполняемому интерпретатором CPython, обращаться к компонентам Java   
и .NET.

72   
Глава 2. Как Python запускает программы   
управлением Psyco могут выполняться так же быстро, как скомпилированный   
код языка C.  
Поскольку эта компиляция из байт-кода производится во время выполне-  
ния программы, обычно Psyco называют динамическим (just-in-time, JIT)   
компилятором. Однако в действительности Psyco немного отличается от JIT-  
компиляторов, которые, возможно, некоторые читатели видели в языке Java.   
В действительности Psyco – это специализированный  JIT-компилятор� он   
генерирует машинный код, оптимизированный для типов данных, которые   
фактически используются в программе. Например, если один и тот же участок   
программы использует различные типы данных в разное время, Psyco может   
генерировать различные версии машинного кода для поддержки каждой из   
комбинаций.  
Применение Psyco показывает существенное увеличение скорости выполне-  
ния программного кода Python. Согласно информации, которая приводится   
на домашней странице проекта, Psyco обеспечивает увеличение скорости «от   
2 до 100 раз, обычно в 4 раза, при использовании немодифицированного ин-  
терпретатора Python, неизменного исходного текста, всего лишь за счет ис-  
пользования динамически загружаемого модуля расширения на языке C».   
При прочих равных условиях наибольший прирост скорости наблюдается для   
программного кода, реализующего различные алгоритмы на чистом языке   
Python, – именно такой программный код обычно переносят на язык C с це-  
лью оптимизации. При использовании Psyco необходимость в таком переносе   
теряет свою остроту.  
До сих пор Psyco не является стандартной частью Python – его нужно загру-  
жать и устанавливать отдельно. Кроме того, он до сих пор находится на экс-  
периментальной стадии развития, поэтому вам нужно будет следить за его раз-  
работкой. В действительности, когда я пишу эти строки, Psyco все еще можно   
загрузить и установить, но похоже, что большая его часть будет поглощена бо-  
лее новым проектом «PyPy», который представляет собой попытку переписать   
PVM на языке Python с целью обеспечения высокой степени оптимизации, как   
в Psyco.  
Пожалуй, самым большим недостатком Psyco является то обстоятельство, что   
в настоящее время он способен генерировать машинный код только для архи-  
тектуры Intel x86, впрочем, на этой архитектуре работают такие операционные   
системы, как �indows, Linux и даже Mac. За дополнительной информацией   
о расширении Psyco и других попытках реализации JIT-компилятора обра-  
щайтесь на сайт http://www.python.org. Кроме того, вы можете посетить домаш-  
нюю страницу проекта Psyco, которая в настоящее время размещается по адре-  
су http://psyco.sourceforge.net.  
Транслятор Shedskin C++  
Shedskin – это еще одна система, которая реализует нетрадиционный подход   
к выполнению программ на языке Python. Она преобразует исходный код на   
языке Python в исходный код на языке C++, который затем может быть ском-  
пилирован в машинный код. Кроме того, эта система реализует платформоне-  
зависимый подход к выполнению программного кода Python. К моменту на-  
писания этих строк система Shedskin еще находилась на экспериментальной   
стадии развития и ограничивала программы Python неявным использованием

Разновидности модели выполнения   
73  
статических типов, что является ненормальным явлением для программ на   
языке Python, поэтому мы не будем углубляться в описание этой системы. Тем   
не менее, по предварительным результатам, у нее имеется немалый потенциал,   
чтобы выиграть гонку за скоростью как у стандартной реализации Python, так   
и у расширения Psyco, и это весьма многообещающий проект. Сведения о теку-  
щем состоянии проекта вы можете самостоятельно найти в Сети.  
Фиксированные двоичные файлы  
Иногда, когда пользователи спрашивают про «настоящий» компилятор языка   
Python, в действительности они просто ищут способ создавать из своих про-  
грамм на языке Python самостоятельные исполняемые файлы. Это необходи-  
мо скорее для упаковки и распространения программ, чем для их исполнения,   
но эти две стороны взаимосвязаны между собой. При помощи инструментов   
сторонних разработчиков, которые можно загрузить из Сети, вы можете пре-  
вратить свои программы на языке Python в настоящие исполняемые файлы,   
которые в мире Python известны как фиксированные двоичные файлы (frozen   
binaries).  
Фиксированные двоичные файлы объединяют в единый файл пакета байт-код   
программ, PVM (интерпретатор) и файлы поддержки, необходимые програм-  
мам. Существуют разные реализации такого подхода, но в конечном результа-  
те получается единственный исполняемый файл (например, файл с расшире-  
нием .exe в �indows), который легко можно передать заказчику. Такую модель   
можно представить, если на рис. 2.2 объединить байт-код и PVM в единый ком-  
понент – фиксированный двоичный файл.  
На сегодняшний день существует три основных инструмента создания фикси-  
рованных двоичных файлов: py2exe (для �indows), PyInstaller (напоминает   
py2exe, но также работает в Linux и UNIX и способен производить самоуста-  
навливающиеся исполняемые файлы) и freeze (оригинальная версия). Вам при-  
дется загружать эти инструменты отдельно от Python, но они распространя-  
ются совершенно бесплатно. Кроме того, они постоянно развиваются, поэтому   
свежую информацию об этих инструментах смотрите на сайте проекта Python   
(http://www.python.org) или ищите с помощью поисковых систем. Чтобы дать   
вам общее представление об области применения этих инструментов, замечу,   
что py2exe может создавать автономные программы, использующие библиоте-  
ки tkinter, PM�, wxPython и PyGTK для создания графического интерфейса�   
программы, использующие инструментальные средства создания игр pygame�   
клиентские программы win32com и многие другие.  
Фиксированные двоичные файлы – это не то же самое, что получается в ре-  
зультате работы настоящего компилятора, потому что они выполняют байт-  
код с помощью виртуальной машины. Следовательно, программы в фиксиро-  
ванных двоичных файлах исполняются с той же скоростью, что и обычные   
файлы с исходными текстами программ, разве что улучшен способ их запуска.   
Фиксированные двоичные файлы имеют немалый размер (они содержат в себе   
PVM), но по современным меркам их все же нельзя назвать необычно больши-  
), но по современным меркам их все же нельзя назвать необычно больши-  
ми. Так как интерпретатор Python встроен непосредственно в фиксированные   
двоичные файлы, его установка не является обязательным требованием для   
запуска программ на принимающей стороне. Более того, поскольку программ-

74   
Глава 2. Как Python запускает программы   
ный код упакован в фиксированный двоичный файл, он надежно скрыт от по-  
лучателя.  
Такая схема упаковки программ в единственный файл особенно подходит для   
нужд разработчиков коммерческого программного обеспечения. Например,   
программа с графическим интерфейсом на базе tkinter может быть упакована   
в исполняемый файл и распространяться как самостоятельная программа на   
CD или через Интернет. Конечному пользователю не нужно будет устанавли-  
 или через Интернет. Конечному пользователю не нужно будет устанавли-  
вать Python (и даже знать о том, что это такое), чтобы запустить распростра-  
Python (и даже знать о том, что это такое), чтобы запустить распростра-  
 (и даже знать о том, что это такое), чтобы запустить распростра-  
няемую программу.  
Другие способы выполнения  
Существуют также другие схемы выполнения программ на языке Python, пре-  
следующие узкоспециализированные цели:  
 •  
Система Stackless  Python – это разновидность стандартной реализации   
CPython, которая не использует стек вызовов языка C. Она упрощает пере-  
, которая не использует стек вызовов языка C. Она упрощает пере-  
C. Она упрощает пере-  
. Она упрощает пере-  
нос Python на архитектуры с небольшим объемом стека, обеспечивает до-  
Python на архитектуры с небольшим объемом стека, обеспечивает до-  
 на архитектуры с небольшим объемом стека, обеспечивает до-  
полнительные возможности параллельной обработки данных и поощряет   
использование новейших инструментов языка, таких как сопрограммы.  
 •  
Система Cython (расширенная версия проекта Pyrex) – это гибридный   
язык, дополняющий язык Python возможностью вызывать функции на   
языке C и использовать объявления типов переменных, аргументов и атри-  
C и использовать объявления типов переменных, аргументов и атри-  
 и использовать объявления типов переменных, аргументов и атри-  
бутов классов на языке C. Исходные тексты на языке Cython могут быть   
скомпилированы в программный код на языке C, использующий Python/C   
API, который в свою очередь может быть скомпилирован в машинный код.   
Несмотря на то, что получающийся программный код не полностью со-  
вместим со стандартным языком Python, Cython может оказаться как по-  
Python, Cython может оказаться как по-  
, Cython может оказаться как по-  
Cython может оказаться как по-  
 может оказаться как по-  
лезным инструментом для создания оберток вокруг внешних библиотек на   
языке C, так и эффективным средством разработки расширений на C для   
языка Python.  
Дополнительные подробности об этих системах вы без труда найдете в Интер-  
нете.  
Будущие возможности  
В заключение обратите внимание, что модель выполнения, обсуждавшаяся   
здесь, в действительности является лишь отражением текущей реализации   
интерпретатора Python, но не самого языка программирования. Например,   
вполне возможно, что в течение времени, пока эта книга будет сохранять ак-  
туальность (едва ли она сохранится у кого-нибудь через десять лет), появится   
традиционный компилятор для трансляции исходного текста на языке Python   
в машинный код. Кроме того, в будущем могут появиться новые варианты реа-  
лизации интерпретатора и разновидности байт-кода. Например:  
 •  
Проект Parrot поставил перед собой цель выработать единый формат байт-  
кода, единую виртуальную машину и методики оптимизации для различ-  
ных языков программирования (подробности на сайте http://www.python.  
org). Стандартная виртуальная машина PVM в интерпретаторе пока выпол-  
няет программный код быстрее, чем Parrot, но пока неясно, как будет раз-  
виваться этот проект.

В заключение   
75  
 •  
Проект PyPy – попытка реализовать PVM непосредственно на языке Py-  
PVM непосредственно на языке Py-  
 непосредственно на языке Py-  
Py-  
thon, что позволит использовать новые приемы программирования. Его   
цель – создать быструю и гибкую реализацию Python.  
 •  
Проект Unladen Swallow, поддерживаемый компанией Google, поставил пе-  
Google, поставил пе-  
, поставил пе-  
ред собой задачу повысить производительность стандартного интерпретато-  
ра Python по меньшей мере в 5 раз, что позволит использовать его в качестве   
замены языку C во многих проектах. Предполагается, что в результате этой   
оптимизации будет создана полностью совместимая версия CPython, кото-  
CPython, кото-  
, кото-  
рая выполняется гораздо быстрее. Участники этого проекта также надеют-  
ся удалить глобальную блокировку Python (Global Interpreter Lock, GIL),   
которая препятствует возможности одновременного выполнения несколь-  
ких потоков управления. В настоящее время этот проект разрабатывается   
инженерами из Google как проект с открытыми исходными текстами – из-  
Google как проект с открытыми исходными текстами – из-  
 как проект с открытыми исходными текстами – из-  
начально за основу в нем была принята версия Python 2.6, однако вполне   
возможно, что изменения смогут быть внесены и в версию 3.0. Дополни-  
тельные подробности вы найдете на сайте компании Google.  
Подобные грядущие схемы реализации могут несколько изменить схему вре-  
мени выполнения интерпретатора Python, однако, скорее всего компилятор   
байт-кода останется стандартом еще какое-то время. Переносимость и гиб-  
кость байт-кода во время выполнения – это очень важные качества многих реа-  
лизаций Python. Более того, добавление в язык конструкций объявления ти-  
Python. Более того, добавление в язык конструкций объявления ти-  
. Более того, добавление в язык конструкций объявления ти-  
пов с целью обеспечения статической компиляции только повредит гибкости,   
осмысленности, простоте и общему духу языка Python. Из-за динамической   
природы языка Python любые реализации в будущем, скорее всего, сохранят   
некоторые черты нынешней PVM.  
В заключение  
В этой главе была представлена модель выполнения Python (как Python запу-  
Python (как Python запу-  
 (как Python запу-  
Python запу-  
 запу-  
скает программы) и исследованы некоторые наиболее известные разновидно-  
сти этой модели (динамические компиляторы и тому подобное). Чтобы писать   
сценарии на языке Python, вам необязательно знать внутреннюю организацию   
интерпретатора и, тем не менее, некоторое знакомство с темой этой главы по-  
может вам понять, как выполняются ваши программы. В следующей главе вы   
начнете выполнять свой собственный программный код. А теперь – обычные   
контрольные вопросы.  
Закрепление пройденного  
Контрольные вопросы  
1. Что такое интерпретатор Python?  
2. Что такое исходный программный код?  
3. Что такое байт-код?  
4. Что такое PVM?  
5. Назовите две разновидности стандартной модели выполнения Python.  
6. В чем заключаются различия между CPython, Jython и IronPython?

76   
Глава 2. Как Python запускает программы   
Ответы  
1. Интерпретатор Python – это программа, которая выполняет программы на   
языке Python.  
2. Исходный программный код – это инструкции, составляющие программу.   
Он состоит из текста в текстовых файлах, имена которых обычно имеют   
расширение .py.  
3. Байт-код – это низкоуровневое представление программы после ее компи-  
ляции. Python автоматически сохраняет полученный байт-код в файлах   
с расширением .pyc.  
4. PVM – это Python Virtual Machine (виртуальная машина Python) – меха-  
низм Python, который интерпретирует компилированный программный   
код.  
5. Psyco, Shedskin и фиксированные двоичные файлы – все это разновидности   
модели выполнения.  
6. CPython – это стандартная реализация языка. Jython и IronPython реали-  
зуют поддержку программирования на языке Python в среде Java и .NET   
соответственно� они являются альтернативными компиляторами языка   
Python.

Глава 3.  
   
Как пользователь запускает программы  
Итак, настал момент запустить какой-нибудь программный код. Теперь, когда   
вы получили представление, как выполняются программы, вы готовы присту-  
пать к программированию на языке Python. С этого момента я буду предпола-  
гать, что интерпретатор Python уже установлен у вас на компьютере, в против-  
ном случае вернитесь к предыдущей главе, а также прочитайте приложение A,   
где приводятся советы по установке и настройке интерпретатора.  
Существует несколько способов заставить интерпретатор Python выполнить   
программу, которую вы написали. В этой главе рассматриваются все наиболее   
часто используемые приемы запуска программ. Попутно вы узнаете, как вво-  
дить программный код в интерактивном режиме, как сохранять его в фай-  
лах, которые можно будет запускать из командной строки, щелчком на ярлыке   
и импортировать и загружать в виде модулей, выполнять с помощью инструк-  
ции exec или меню графического интерфейса IDLE.  
Если вам требуется лишь узнать, как запускать программы на языке Python,   
прочитайте раздел, где описывается ваша платформа, и переходите к главе 4.   
Однако не пропускайте материал, где описывается импортирование модулей,   
потому что эти сведения являются основой для понимания архитектуры про-  
грамм на языке Python. Я также рекомендую просмотреть разделы с описани-  
ем IDLE и других интегрированных сред разработки, чтобы вы представляли,   
какие инструментальные средства доступны, когда приступите к разработке   
более сложных программ.  
Интерактивный режим  
Пожалуй, самый простой способ запускать программы на языке Python – это   
вводить инструкции непосредственно в командной строке интерпретатора, ко-  
торая иногда называется интерактивной  оболочкой. Запустить эту команд-  
ную строку можно разными способами – в интегрированной среде разработки,   
в системной консоли и так далее. Предположим, что интерпретатор установлен   
в вашей системе как выполняемая программа, тогда самый универсальный   
способ запустить интерактивный сеанс работы с интерпретатором заключает-

78   
Глава 3. Как пользователь запускает программы   
ся в том, чтобы ввести команду python без аргументов в командной строке ва-  
шей операционной системы. Например:  
% python  
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] ...  
Type “help”, “copyright”, “credits” or “license” for more information.  
>>>  
После ввода слова «python» командная оболочка вашей операционной системы   
запустит интерактивный сеанс работы с интерпретатором Python (символ «%»   
здесь означает строку приглашения к вводу, он не должен вводиться вами). По-  
нятие системной командной строки является универсальным, но как получить   
доступ к ней, зависит от используемой платформы:  
 •  
В операционной системе �indows команду python можно ввести в консоли   
DOS (она же – Командная Строка (Command Prompt), которую обычно можно   
найти в разделе Стандартные (Accessories) меню Все программы (Programs), ко-  
торое появляется после щелчка на кнопке Пуск (Start)), или в диалоге Пуск   
(Start) → Выполнить… (Run…).  
 •  
В операционных системах UNIX, Linux и Mac OS X эту команду можно   
ввести в командной оболочке или в окне терминала (например, в xterm или   
в консоли, где запущена командная оболочка, такая как ksh или csh).  
 •  
В других операционных системах можно использовать похожие или какие-  
то специфичные для платформы устройства. Например, чтобы запустить   
интерактивный сеанс в наладонных устройствах, обычно достаточно щел-  
кнуть на ярлыке Python.  
Если вы не включили путь к каталогу установки Python в переменную окру-  
жения PATH, вместо простого слова «python» вам может потребоваться ввести   
полный путь к выполняемой программе. В операционной системе �indows   
можно попробовать ввести команду C:\Python30\python (для версии 3.0)� в UNIX   
и в Linux: /usr/local/bin/python или /usr/bin/python:  
C:\misc> c:\python30\python  
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] ...  
Type “help”, “copyright”, “credits” or “license” for more information.  
>>>  
Или перед вводом команды «python» можно перейти в каталог, куда был уста-  
python» можно перейти в каталог, куда был уста-  
» можно перейти в каталог, куда был уста-  
новлен интерпретатор Python. Для этого в операционной системе �indows, на-  
пример, можно выполнить команду cd c:\python30, например:  
C:\misc> cd C:\Python30  
C:\Python30> python  
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] ...  
Type “help”, “copyright”, “credits” or “license” for more information.  
>>>  
В �indows, кроме ввода команды python в окне командной оболочки, запустить   
интерактивный сеанс можно также, открыв главное окно IDLE (обсуждается   
ниже) или выбрав пункт Python (command line) (Python (командная строка)) в подме-  
ню Python в меню Пуск (Start), как показано на рис. 2.1 в главе 2. В обоих случаях   
открывается интерактивный сеанс Python с одинаковыми возможностями, то   
есть ввод команды в командной оболочке не является обязательным условием.

Интерактивный режим   
79  
Выполнение инструкций в интерактивном режиме  
Интерактивный сеанс работы с интерпретатором Python начинается с вывода   
двух строк информационного текста (которые я буду опускать в примерах для   
экономии места), затем выводится приглашение к вводу >>>, когда интерпре-  
татор Python переходит в режим ожидания ввода новой инструкции или вы-  
Python переходит в режим ожидания ввода новой инструкции или вы-  
 переходит в режим ожидания ввода новой инструкции или вы-  
ражения. При работе в интерактивном режиме результаты выполнения ваших   
инструкций будут выводиться сразу же после нажатия клавиши Enter вслед за   
строкой с приглашением >>>.   
Например, ниже приводятся результаты выполнения двух инструкций print   
(в действительности инструкция print была инструкцией в Python 2.6, а в   
Python 3.0 она стала функцией, поэтому круглые скобки являются обязатель-  
ным элементом только в версии 3.0):  
% python  
>>> print(‘Hello world!’)  
Hello world!  
>>> print(2 \*\* 8)  
256  
Вам пока также не стоит вникать в детали инструкций print, приведенных   
здесь, – изучение синтаксиса мы начнем в следующей главе. В двух словах,   
эта инструкция вывела текстовую строку и целое число, как видно в строках,   
которые были напечатаны ниже строк с приглашением к вводу >>> (выражение   
2 \*\* 8 на языке Python означает 2 в степени 8).  
При работе в интерактивном режиме, как показано в этом примере, вы можете   
вводить любое число команд Python, и каждая из них будет выполняться сразу   
же после ввода. Более того, поскольку в интерактивном сеансе результаты вы-  
ражений, которые вы вводите, выводятся автоматически, совершенно необяза-  
тельно явно использовать функцию «print»:  
>>> lumberjack = ‘okay’  
>>> lumberjack  
‘okay’  
>>> 2 \*\* 8  
256  
>>> # Для выхода используйте клавиши Ctrl-D (в UNIX) или Ctrl-Z (в Windows)  
%  
В этом примере первая строка сохраняет значение в переменной, а две послед-  
ние введенные строки являются выражениями (lumberjack и 2 \*\* 8), результаты   
вычисления которых отображаются автоматически. Чтобы завершить работу   
интерактивного сеанса, как показано в данном примере, и вернуться в систем-  
ную командную строку, в UNIX-подобной системе нажмите комбинацию кла-  
виш Ctrl-D, а в системах MS-DOS и �indows – комбинацию Ctrl-Z. В графическом   
интерфейсе IDLE, который будет рассматриваться ниже, нужно либо нажать   
комбинацию клавиш Ctrl-D, либо просто закрыть окно.  
В приведенных примерах мы сделали немногое – всего лишь ввели несколько   
инструкций print, одну инструкцию присваивания и несколько выражений,   
о которых подробнее мы поговорим позднее. Главное, на что следует обратить   
внимание, – интерпретатор немедленно выполняет введенный программный   
код сразу же после нажатия клавиши Enter.

80   
Глава 3. Как пользователь запускает программы   
Например, когда в строке приглашения к вводу >>> была введена первая ин-  
струкция print, результат (строка) был немедленно выведен на экран. Нам не   
потребовалось создавать файл с исходным текстом программы и для выпол-  
нения программного кода не понадобилось сначала компилировать и компоно-  
вать его, что является обычным делом при использовании таких языков про-  
граммирования, как C или C++. Как будет показано в последующих главах,   
при работе с интерактивной оболочкой вы можете также вводить многостроч-  
ные инструкции – такие инструкции будут выполняться только после ввода   
всех строк.  
Когда может пригодиться интерактивный режим?  
В интерактивном режиме интерпретатор немедленно выполняет введенные ин-  
струкции и выводит результат, но эти инструкции не сохраняются в файле. Это   
означает, что в интерактивном режиме вы едва ли будете выполнять длинные   
отрывки программного кода, но при этом интерактивный режим предостав-  
ляет отличную возможность для проведения экспериментов с возможностями   
языка и тестирования файлов программ на лету.  
Экспериментирование  
Благодаря тому, что программный код выполняется немедленно, интерактив-  
ный режим превращается в замечательный инструмент для проведения экс-  
периментов с конструкциями языка. Интерактивная оболочка часто будет   
использоваться в этой книге для демонстрации небольших примеров. Самое   
первое, что вы должны запомнить: если вы чувствуете, что не понимаете, как   
работает тот или иной отрывок программного кода на языке Python, запустите   
интерактивный сеанс и попробуйте ввести этот фрагмент, чтобы посмотреть,   
что произойдет.   
Например, предположим, что вы изучаете некоторый фрагмент программы на   
языке Python и наталкиваетесь на выражение ‘Spam!’ \* 8, которое вам кажется   
непонятным. Можно, конечно, потратить с десяток минут, пробираясь через   
руководства и учебники, в попытках выяснить, что же делает этот код, но мож-  
но просто выполнить его в интерактивной оболочке:  
>>> ‘Spam!’ \* 8 <== Изучение методом проб и ошибок  
‘Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!’  
Немедленная обратная связь, которую предоставляет интерактивная оболоч-  
ка, часто позволяет быстрее всего выяснить, что делает тот или иной фрагмент   
программного кода. Эксперимент наглядно показывает, что произошло дубли-  
рование строки: в языке Python оператор \* выполняет операцию умножения   
над числами, но если левый операнд является строкой, он действует как опе-  
ратор многократной конкатенации строки с самой собой (подробнее о строках   
рассказывается в главе 4).  
При проведении подобных экспериментов вы едва ли что-нибудь испортите, по   
крайней мере, пока. Чтобы причинить серьезный ущерб, например удалить   
файл или выполнить команду системной командной оболочки, необходимо   
явно импортировать модули (чтобы стать опасным для системы, вы должны   
хорошо знать системные интерфейсы языка Python!). Простой программный   
код на языке Python практически всегда может быть выполнен без опаски.

Интерактивный режим   
81  
Для примера посмотрите, что произойдет, если допустить ошибку в интерак-  
тивной оболочке:  
>>> X <== Допущена ошибка  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
NameError: name ‘X’ is not defined  
В языке Python считается ошибкой пытаться использовать переменную пре-  
Python считается ошибкой пытаться использовать переменную пре-  
 считается ошибкой пытаться использовать переменную пре-  
жде чем ей будет присвоено значение (в противном случае, если бы переменные   
заполнялись некоторым значением по умолчанию, некоторые ошибки было бы   
очень сложно обнаруживать). С этой особенностью мы познакомимся позднее,   
а пока обратите внимание, что допущенная ошибка не вызвала ни аварийного   
завершения оболочки Python, ни краха системы. Вместо этого было выведено   
информативное сообщение об ошибке с указанием номера строки, где она была   
допущена, и вы получили возможность продолжить сеанс работы с интерпре-  
татором. В действительности, как только вы более или менее освоитесь с язы-  
ком Python, сообщения об ошибках во многих случаях станут оказывать вам   
вполне достаточную помощь при отладке (подробнее об отладке рассказывает-  
ся во врезке «Отладка программ на языке Python»).  
Тестирование  
Помимо функции инструмента, предоставляющего возможность поэкспери-  
ментировать в процессе изучения языка, интерактивная оболочка также яв-  
ляется идеальным средством тестирования программного кода, сохраненного   
в файлах. В интерактивном режиме вы можете импортировать файлы модулей   
и тестировать функциональные возможности, которые они предоставляют,   
вводя вызовы функций в строке с приглашением к вводу.  
Например, ниже приводится пример тестирования функции, находящей   
в одном из модулей, входящих в стандартную библиотеку Python (она возвра-  
Python (она возвра-  
 (она возвра-  
щает имя текущего рабочего каталога). Вы сможете проделывать то же самое со   
своими собственными функциями, как только начнете создавать свои модули:  
>>> import os  
>>> os.getcwd() <== Тестирование “на лету”  
‘c:\\Python30’  
В более широком понимании интерактивная оболочка – это инструмент для   
тестирования программных компонентов независимо от их происхождения –   
вы можете вводить вызовы функций из связанных библиотек на языке C,   
создавать экземпляры классов Java в интерпретаторе Jython и делать многое   
другое. Интерпретатор поддерживает возможность проведения экспериментов   
и исследований при программировании, и вы найдете это удобным, начав ра-  
ботать с ним.  
Использование интерактивного режима  
Несмотря на то, что интерактивный режим прост в использовании, я хочу дать   
несколько советов начинающим, которые следует запомнить. Я привожу спи-  
ски наиболее распространенных ошибок, подобные приведенному в этой главе,   
для справки, потому что знакомство с ними позволит вам избежать лишней   
головной боли:

82   
Глава 3. Как пользователь запускает программы   
 •  
Вводите только инструкции на языке Python. Прежде всего следует запом-  
нить, что в интерактивном режиме допускается вводить только программ-  
ный код на языке Python, никаких системных команд. В программном   
коде Python предусмотрены возможности выполнять системные команды   
(например, с помощью os.system), но они не отличаются простотой по срав-  
нению с непосредственным вводом команд.  
 •  
Инструкция print необходима только в файлах. Поскольку в интерактив-  
ном режиме интерпретатор автоматически выводит результаты вычисле-  
ния выражений, вам не требуется вводить полные инструкции print при   
работе в интерактивном режиме. Это замечательная особенность, но она ча-  
сто приводит пользователей в замешательство, когда они приступают к соз-  
данию программного кода в файлах: чтобы программный код в файлах мог   
что-то выводить, вы должны использовать инструкции print, потому что   
в этом случае результаты выражений уже не выводятся автоматически. За-  
помните, вы должны использовать инструкцию print в файлах, но не в ин-  
терактивном режиме.  
 •  
Не используйте отступы в интерактивном режиме (пока). При вводе про-  
грамм на языке Python, как в интерактивном режиме, так и в текстовых   
файлах, вы обязаны начинать все не вложенные инструкции с позиции 1   
(то есть с самого начала строки). Если вы не будете следовать этому правилу,   
Python может вывести сообщение «SyntaxError» (синтаксическая ошибка),   
потому что пробелы слева от инструкции рассматриваются интерпретато-  
ром как отступ, обозначающий принадлежность инструкции к вложенной   
группе. Пока мы не подойдем к главе 10, все инструкции, которые вам при-  
дется вводить, будут не вложенными, поэтому пока данное правило распро-  
страняется на все, что будет вводиться. То же относится и к классам Python.   
Ведущий пробел в строке вызывает сообщение об ошибке.  
 •  
Будьте внимательны, когда строка приглашения к вводу изменяется на   
строку ввода составной инструкции. Нам не придется сталкиваться с со-  
ставными (многострочными) инструкциями до главы 4 и вводить их до   
главы 10, но вы должны знать, что при вводе второй и каждой последующей   
строки составной инструкции в интерактивном режиме строка приглаше-  
ния к вводу может менять свой вид. В простом окне с командной строкой   
приглашение к вводу >>> во второй и каждой последующей строке изменя-  
ется на ..., в интерфейсе IDLE все строки, кроме первой, автоматически по-  
лучают отступы.   
Почему это так важно, вы узнаете в главе 10. А пока, если вдруг случит-  
ся, что вы получите приглашение к вводу ... или пустую строку при вводе   
программного кода, это, скорее всего, будет означать, что каким-то обра-  
зом вам удалось заставить интерактивную оболочку Python думать, что вы   
начали ввод многострочной инструкции. Попробуйте нажать комбинацию   
Ctrl-C или клавишу Enter, чтобы вернуться к основному приглашению к вво-  
ду. Строки приглашений >>> и ... можно изменить (они доступны во встро-  
енном модуле sys), но в последующих листингах я буду предполагать, что   
они не изменялись.  
 •  
При работе в интерактивном режиме завершайте ввод составных инструк-  
ций вводом пустой строки. Ввод пустой строки в интерактивном режиме   
(нажатие клавиши Enter в начале строки) сообщает интерпретатору, что вы   
завершили ввод многострочной инструкции. То есть, чтобы выполнить со-

Интерактивный режим   
83  
ставную инструкцию, необходимо дважды нажать клавишу Enter. В фай-  
лах, напротив, в конце составных инструкций пустая строка не требуется   
и если она имеется, интерпретатор будет просто игнорировать ее. Если при   
работе в интерактивном режиме не завершить ввод составной инструкции   
двумя нажатиями клавиши Enter, у вас может сложиться впечатление, что   
все повисло, потому что интерпретатор не будет делать ничего, ожидая, ког-  
да вы повторно нажмете клавишу Enter!   
 •  
В интерактивном режиме за один раз выполняется одна инструкция. При   
работе в интерактивном режиме сначала следует ввести и выполнить одну   
инструкцию и только потом вводить другую. Для простых инструкций это   
требование соблюдается само собой, так как нажатие клавиши Enter приво-  
дит к выполнению введенной инструкции. Однако при работе с составными   
инструкциями не забывайте, что они должны завершаться и запускаться   
на выполнение вводом пустой строки, и только потом можно будет вводить   
новую инструкцию.  
Ввод многострочных инструкций  
Рискую повториться и все же я получил письма от двух читателей, которые   
пришли в замешательство от двух последних пунктов списка, после того как   
я дополнил эту главу, поэтому я считаю важным остановиться на них. Я по-  
знакомлю вас с многострочными (или составными) инструкциями в следую-  
щей главе, а более формально мы изучим их синтаксис еще ниже в этой книге.   
Поскольку поведение составных инструкций в файлах и в интерактивной обо-  
лочке немного отличается, я посчитал необходимым добавить два следующих   
предупреждения.  
Во-первых, при работе в интерактивном режиме завершайте ввод составных   
инструкций, таких как циклы for и условные операторы if, вводом пустой   
строки. Вы должны нажать клавишу Enter дважды, чтобы завершить много-  
строчную инструкцию и выполнить ее. Например:  
>>> for x in ‘spam’:  
... print(x) <== Здесь нажать Enter дважды, чтобы выполнить этот цикл  
...  
В файлах сценариев не требуется добавлять пустую строку после составных   
инструкций – это необходимо только при работе в интерактивном режиме.   
В файле пустые строки не являются обязательными, и они просто игнориру-  
ются, а в интерактивном режиме они завершают многострочные инструкции.  
Обратите также внимание на то, что интерактивная оболочка выполняет по   
одной инструкции за один раз: прежде чем вводить следующую инструкцию,   
вы должны дважды нажать клавишу Enter, чтобы выполнить цикл или другую   
многострочную инструкцию:  
>>> for x in ‘spam’:  
... print(x) <== Здесь нажать Enter дважды, прежде чем вводить новую инструкцию  
... print(‘done’)   
 File “<stdin>”, line 3  
 print(‘done’)  
 ^  
SyntaxError: invalid syntax

84   
Глава 3. Как пользователь запускает программы   
Это означает, что вы не можете копировать и вставлять сразу несколько строк   
программного кода в интерактивном режиме, если копируемый фрагмент не   
содержит пустые строки после каждой составной инструкции. Такой код луч-  
ше выполнять в виде файла, о чем рассказывается в следующем разделе.  
Системная командная строка и файлы  
Хотя интерактивная командная оболочка является прекрасным инструмен-  
том для проведения экспериментов, тем не менее, у нее есть один существен-  
ный недостаток: программы, которые вы вводите во время интерактивного се-  
анса, исчезают сразу же после того, как интерпретатор Python выполнит их.   
Программный код, который вводится в интерактивном режиме, нигде не со-  
храняется, поэтому вы не сможете запустить его еще раз, не введя код с само-  
го начала. Операция копирования и вставки, а также возможность повторного   
выполнения команды могут оказать некоторую помощь, но они не будут полез-  
ны, когда вы начнете писать большие программы. Чтобы воспользоваться опе-  
рацией копирования и вставки, вы должны исключить из копирования строку   
приглашения к вводу, результаты, которые программа выводит в процессе вы-  
полнения, и так далее – далеко не самая современная методология разработки   
программного обеспечения!  
Чтобы хранить программы длительное время, необходимо сохранять про-  
граммный код в файлах, которые обычно называются модулями. Модули –   
это простые текстовые файлы, содержащие инструкции на языке Python.   
Как только такой файл будет создан, вы можете предложить интерпретатору   
Python выполнить инструкции в нем столько раз, сколько пожелаете. Такой   
файл можно запустить на выполнение разными способами – из командной   
строки системы, щелчком на ярлыке файла, из пользовательского интерфей-  
са IDLE и другими способами. Независимо от выбранного вами способа интер-  
претатор Python будет выполнять весь программный код в модуле от начала до   
конца всякий раз, когда вы будете его запускать.  
Терминология в этой области может несколько изменяться. Например, файлы   
модулей часто называются программами на языке Python, где под програм-  
мой понимается последовательность заранее написанных инструкций, сохра-  
ненных в файле для обеспечения возможности многократного использования.   
Файлы модулей, которые запускаются на выполнение непосредственно, иногда   
называют сценариями – этим неофициальным термином обозначаются файлы   
программ верхнего уровня. Термин «модуль» зарезервирован для обозначения   
файлов, которые могут импортироваться другими файлами. (Подробнее о про-  
граммах «верхнего уровня» и об импорте будет говориться чуть ниже.)  
Как бы вы ни называли их, в следующих нескольких разделах исследуются   
способы запуска программного кода, который был сохранен в файлах модулей.   
В этом разделе вы узнаете, как запускать файлы наиболее типичным способом:   
перечислением их имен в команде python при запуске из системной командной   
строки. Кому-то это может показаться примитивным, но для большинства про-  
граммистов для разработки программ вполне достаточно окна терминала с ко-  
мандной оболочкой и окна текстового редактора.

Системная командная строка и файлы   
85  
Первый сценарий  
В качестве первого упражнения откройте привычный для вас текстовый ре-  
дактор (например, vi, Notepad или редактор IDLE) и сохраните следующие ин-  
Notepad или редактор IDLE) и сохраните следующие ин-  
 или редактор IDLE) и сохраните следующие ин-  
струкции в файле с именем script1.py:  
# Первый сценарий на языке Python  
import sys # Загружает библиотечный модуль  
print(sys.platform)  
print(2 \*\* 100) # Возводит число 2 в степень 100  
x = ‘Spam!’  
print(x \* 8) # Дублирует строку  
Это наш первый официальный сценарий на языке Python (если не считать   
двухстрочный сценарий из главы 2). Пока не нужно вникать в синтаксис про-  
граммного кода в этом файле, тем не менее, в качестве краткого описания ска-  
жу, что этот файл:  
 •  
Импортирует модуль Python (библиотеку дополнительных инструментов),   
чтобы позднее получить название платформы  
 •  
Трижды вызывает функцию print, чтобы отобразить результаты  
 •  
Использует переменную с именем x, которая создается в момент, когда ей   
присваивается значение в виде строкового объекта  
 •  
Выполняет некоторые операции над объектами, с которыми мы познако-  
мимся в следующей главе  
Имя sys.platform – это просто строковая переменная, содержимое которой   
идентифицирует тип компьютера, на котором выполняется сценарий. Эта пе-  
ременная находится в стандартном модуле с именем sys, который необходимо   
загрузить с помощью инструкции import (подробнее об импортировании мы по-  
говорим позже).  
Для разнообразия я также добавил комментарии – текст, следующий за сим-  
волом #. Комментарии могут занимать отдельную строку или добавляться   
в строку с программным кодом, правее его. Текст, следующий за символом #,   
интерпретатором просто игнорируется, как комментарий, добавленный для   
человека, и не считается частью инструкции. Если вы копируете этот пример,   
чтобы опробовать его, можете смело игнорировать комментарии. В этой книге   
я использовал несколько иной стиль оформления комментариев, чтобы обеспе-  
чить их визуальное отличие, но в ваших программах они будут выглядеть как   
обычный текст.  
Повторюсь еще раз, не старайтесь пока вникнуть в синтаксис программного   
кода в этом файле – с ним мы познакомимся позднее. Главное здесь то, что про-  
граммный код вводится в текстовый файл, а не в интерактивной командной   
оболочке интерпретатора Python. Итак, вы создали полноценный сценарий на   
языке Python.  
Обратите внимание, что файл модуля называется script1.py. Так как он явля-  
ется файлом верхнего уровня, его точно так же можно было бы назвать просто   
script, но имена файлов с программным кодом, которые предполагается им-  
портировать из других файлов, должны оканчиваться расширением .py. Об   
импортировании рассказывается ниже, в этой же главе. Позднее вам может   
потребоваться импортировать тот или иной файл, поэтому всегда желательно

86   
Глава 3. Как пользователь запускает программы   
использовать расширение .py в именах файлов с программным кодом на языке   
Python. Кроме того, некоторые текстовые редакторы определяют принадлеж-  
ность файлов по расширению .py – если расширение отсутствует, вы можете   
лишиться таких функциональных возможностей редактора, как подсветка   
синтаксиса и автоматическое оформление отступов.  
Запуск файлов из командной строки  
Сохранив этот текстовый файл, вы сможете предложить интерпретатору   
Python выполнить его, указав полное имя файла в качестве первого аргумента   
команды python, введя следующую строку в системной командной строке:  
% python script1.py  
win32  
1267650600228229401496703205376  
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!   
И в этом случае также вы должны использовать командную оболочку, кото-  
рая предоставляется вашей операционной системой – в окне Командная строка   
(Command Prompt) в �indows, в xterm или в подобных им программах. Не забы-  
вайте заменять слово «python» на полный путь к исполняемому файлу интер-  
претатора, если переменная окружения PATH у вас не настроена.   
Если все было сделано правильно, эта команда запустит интерпретатор Py-  
Py-  
thon, который в свою очередь последовательно, строку за строкой, выполнит   
инструкции в файле, и вы увидите на экране результаты выполнения трех ин-  
струкций print – название платформы, результат возведения числа 2 в степень   
100 и результат многократного дублирования строки, который мы уже видели   
выше (о двух последних операциях более подробно рассказывается в главе 4).  
Если что-то пошло не так, на экране появится сообщение об ошибке – проверь-  
те еще раз, не было ли допущено ошибок при вводе программного кода в файл   
и повторите попытку. О некоторых способах отладки рассказывается ниже, во   
врезке «Отладка программ на языке Python» на стр. 25, но на данном этапе   
лучше всего будет просто механически скопировать пример.  
Поскольку в данной ситуации для запуска программ на языке Python исполь-  
зуется командная оболочка, можно применять любые синтаксические кон-  
струкции, допускаемые командной оболочкой. Например, можно перенапра-  
вить вывод сценария Python в файл, чтобы детально исследовать полученные   
результаты позднее, как показано ниже:  
% python script1.py > saveit.txt  
В этом случае три строки, которые были показаны в предыдущем примере за-  
пуска сценария, не будут выводиться на экран, а будут записаны в файл saveit.  
txt. Это широко известная возможность перенаправления потоков – она мо-  
жет использоваться как для вывода текста, так и для ввода. Она присутствует   
в �indows и в UNIX-подобных системах. Она мало связана с Python (интерпре-  
�indows и в UNIX-подобных системах. Она мало связана с Python (интерпре-  
 и в UNIX-подобных системах. Она мало связана с Python (интерпре-  
татор Python просто поддерживает ее), поэтому здесь мы не будем углубляться   
в подробности работы механизма перенаправления.  
Если вы пользуетесь операционной системой �indows, этот пример будет ра-  
�indows, этот пример будет ра-  
, этот пример будет ра-  
ботать точно так же, хотя сама командная строка будет выглядеть несколько   
иначе:

Системная командная строка и файлы   
87  
C:\Python30> python script1.py  
win32  
1267650600228229401496703205376  
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!  
Если у вас переменная окружения PATH не настроена и не был выполнен переход   
в каталог интерпретатора, вам необходимо вводить полный путь к исполняемо-  
му файлу интерпретатора Python:  
D:\temp> C:\python30\python script1.py  
win32  
1267650600228229401496703205376  
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!  
В новейших версиях �indows вы можете просто вводить имя файла сценария   
независимо от того, в каком каталоге вы находитесь, потому что новейшие   
версии системы �indows отыскивают программы, необходимые для запуска   
файлов, с помощью реестра �indows, и вам не требуется явно указывать ее   
в командной строке. Например, в современных версиях �indows предыдущую   
команду можно упростить до:  
D:\temp> script1.py  
Наконец, не нужно забывать указывать полный путь к файлу сценария, если   
он находится в каталоге, отличном от того, в котором вы работаете. Например,   
следующая команда будет работать в каталоге D:\other в предположении, что   
путь к команде python включен в переменную окружения PATH, при этом она   
должна запустить сценарий, расположенный в некотором другом каталоге:  
D:\other> python c:\code\otherscript.py  
Если переменная окружения PATH не включает путь к каталогу с исполняемым   
файлом интерпретатора Python и при этом файл сценария не находится в теку-  
Python и при этом файл сценария не находится в теку-  
 и при этом файл сценария не находится в теку-  
щем рабочем каталоге, тогда необходимо будет указать полный путь как к ис-  
полняемому файлу интерпретатора, так и к файлу сценария:  
D:\other> C:\Python30\python c:\code\otherscript.py  
Использование системной командной строки и файлов  
Запуск файлов программ из командной строки системы является достаточно   
простой задачей, особенно если у вас уже есть опыт работы с командной стро-  
кой. Тем не менее ниже описываются несколько ловушек, в которые часто по-  
падают начинающие:  
 •  
Остерегайтесь автоматического присвоения расширения файлам в опера-  
ционной системе Windows. Если для создания файлов программ в �indows   
вы пользуетесь редактором «Блокнот» («Notepad»), перед сохранением вы-  
Notepad»), перед сохранением вы-  
»), перед сохранением вы-  
бирайте тип файла Все файлы (All Files) и явно указывайте расширение .py.   
В противном случае «Блокнот» («Notepad») будет присваивать файлам рас-  
Notepad») будет присваивать файлам рас-  
») будет присваивать файлам рас-  
ширение .txt (например, script1.py.txt), что в некоторых ситуациях ослож-  
нит запуск таких файлов.  
Хуже того, по умолчанию операционная система �indows скрывает расши-  
�indows скрывает расши-  
 скрывает расши-  
рения файлов, поэтому, если вы забыли указать тип файла, вы можете даже   
не заметить, что создали обычный текстовый файл, а не файл, который

88   
Глава 3. Как пользователь запускает программы   
должен запускаться интерпретатором Python. Здесь вам может послужить   
подсказкой ярлык файла – если на нем отсутствует изображение головы   
змеи, у вас могут появиться некоторые проблемы с запуском. Отсутствие   
подсветки синтаксиса в IDLE и открытие файла в редакторе вместо его за-  
пуска в результате щелчка мышью могут служить еще одним признаком   
этой проблемы.  
Текстовый процессор Microsoft �ord похожим образом по умолчанию до-  
Microsoft �ord похожим образом по умолчанию до-  
 �ord похожим образом по умолчанию до-  
�ord похожим образом по умолчанию до-  
 похожим образом по умолчанию до-  
бавляет расширение .doc. Мало этого, он добавляет в файл символы форма-  
тирования, которые являются недопустимыми с точки зрения синтаксиса   
Python. Поэтому возьмите за правило всегда выбирать тип файла Все файлы   
(All Files) при сохранении в операционной системе �indows или используйте   
более удобные для программистов текстовые редакторы, такие как IDLE.   
IDLE не добавляет расширение к имени файла автоматически, даже расши-  
 не добавляет расширение к имени файла автоматически, даже расши-  
рение .py – эта особенность нравится программистам, но не пользователям.  
 •  
Указывайте расширение имени файла и полный путь в командной строке,   
но не в инструкциях импорта. Не забывайте указывать полное имя файла   
в командной строке, то есть используйте, например, команду python script1.  
py, а не python script1. Инструкция import, с которой мы познакомимся   
ниже, в этой же главе, требует, чтобы путь к файлу и его расширение были   
опущены (например, import script1). Это несложно, но данная особенность   
часто является источником ошибок.  
Интерпретацией команды в командной строке занимается система, а не   
интерпретатор Python, и в ней неприменимы правила поиска файлов, ко-  
торые использует Python. Поэтому в командной строке необходимо всегда   
указывать расширение файла .py и при необходимости – путь к файлу. На-  
пример, чтобы запустить файл, находящийся в каталоге, отличном от того,   
в котором вы работаете, обычно необходимо указать полный путь к файлу   
(например, python d:\tests\spam.py). Однако в программном коде на языке   
Python достаточно просто указать инструкцию import spam и доверить ин-  
терпретатору самому отыскать требуемый файл в пути поиска модулей, как   
будет описано ниже.  
 •  
Используйте в файлах инструкции print. Да, мы уже говорили об этом, но   
это настолько распространенная ошибка, что она вполне заслуживает, что-  
бы еще раз напомнить о ней. В отличие от интерактивного режима, чтобы   
вывести результаты работы файлов программ на экран, вы должны исполь-  
зовать инструкции print. Если в процессе работы сценарий ничего не выво-  
дит, проверьте еще раз – добавили ли вы инструкции print. Повторюсь еще   
раз, в интерактивном режиме инструкции print можно не использовать, так   
как интерпретатор автоматически выводит результаты вычисления выра-  
жений – инструкции print не будут здесь помехой, просто мы их не исполь-  
зуем, чтобы избавиться от лишнего ввода с клавиатуры.  
Исполняемые сценарии в UNIX (#!)  
Если вы используете Python в UNIX, Linux или в другой UNIX-подобной опера-  
ционной системе, вы можете превратить файлы с программным кодом на язы-  
ке Python в исполняемые программы, точно так же, как программы на языках   
командной оболочки, таких как ksh или csh. Такие файлы обычно называются   
исполняемыми  сценариями. Проще говоря, исполняемые сценарии в UNIX-

Системная командная строка и файлы   
89  
подобных системах – это обычные текстовые файлы, содержащие инструкции   
на языке Python, но обладающие двумя необходимыми свойствами:  
 •  
Первая строка имеет специальный формат. Первая строка в сценариях,   
как правило, начинается с символов #! (эта комбинация часто называется   
как «hash bang»), за которыми следует путь к интерпретатору Python.  
 •  
Как правило, для файлов сценариев установлено разрешение на выполне-  
ние. Обычно файлы сценариев помечаются как исполняемые файлы, чтобы   
сообщить системе, что они могут быть запущены как самостоятельные про-  
граммы. В UNIX-подобных системах это обычно делается с помощью такой   
команды, как chmod +x file.py.  
Давайте рассмотрим пример для UNIX-подобных систем. Сначала с помощью   
текстового редактора создайте файл модуля Python с именем brian:  
#!/usr/local/bin/python  
Print(‚The Bright Side ‚ + ‚of Life...‘) # + означает конкатенацию строк  
Первая строка в файле сообщает системе, где находится интерпретатор Python.   
С технической точки зрения, для интерпретатора Python первая строка явля-  
ется комментарием. Как уже говорилось ранее, все, что начинается с символа #   
и до конца строки, является комментарием – в них размещается дополнитель-  
ная информация, предназначенная для человека, который будет читать ваш   
программный код. Но когда в файле присутствует комментарий, такой как   
в первой строке, он приобретает особый смысл, потому что система использу-  
ет его для поиска интерпретатора, который будет выполнять остальной про-  
граммный код в файле.  
Кроме того, обратите внимание, что этот файл называется просто brian, в его   
имени отсутствует расширение .py, которое мы использовали ранее для обозна-  
чения модулей. Наличие расширения .py не повредило бы (и даже лишний раз   
напоминало бы, что это файл программы на языке Python), но так как этот   
файл не планируется импортировать из других модулей, такое имя файла яв-  
ляется вполне допустимым. Если дать этому файлу право на выполнение с по-  
мощью команды chmod +x brian, вы сможете запускать его из командной строки   
системы, как если бы это была самая обычная программа:  
% brian  
The Bright Side of Life...  
Для пользователей �indows замечу, что метод, описываемый здесь, характе-  
�indows замечу, что метод, описываемый здесь, характе-  
 замечу, что метод, описываемый здесь, характе-  
рен для UNIX и неприменим на вашей платформе. Однако вам незачем волно-  
ваться, просто используйте метод запуска, который описывался выше. Укажи-  
те имя выполняемого файла, как первый аргумент команды python.1  
1   
Как уже говорилось при обсуждении командной строки, современные версии �in-  
dows позволяют указывать в командной строке лишь имя файла с расширением   
.py – эти версии �indows с помощью реестра определяют, что данный файл должен   
открываться с использованием интерпретатора Python (например, команда brian.py   
в них эквивалентна команде python brian.py). Такой режим работы командной строки   
сродни использованию #! в системах UNIX. Следует заметить, что некоторые про-  
граммы для �indows действительно могут использовать и интерпретировать первую   
строку #!, но командная оболочка DOS в �indows полностью игнорирует ее.

90   
Глава 3. Как пользователь запускает программы   
C:\misc> python brian  
The Bright Side of Life...  
В этом случае не требуется добавлять специальный комментарий #! в начало   
файла (хотя, если он присутствует, Python просто игнорирует его) и файл не   
должен иметь право на выполнение. Фактически если вы хотите обеспечить   
переносимость процедуры запуска между UNIX и Microsoft �indows, ваша   
жизнь наверняка станет проще, если вы всегда будете использовать типичный   
подход к запуску программ из командной строки, а не стиль, используемый   
для запуска сценариев в UNIX.  
Трюк с использованием команды env в UNIX  
В некоторых версиях системы UNIX можно избежать явного указания   
пути к интерпретатору Python, если специальный комментарий в пер-  
вой строке оформить, как показано ниже:  
#!/usr/bin/env python  
...здесь находится программный код сценария...  
При таком подходе программа env отыщет интерпретатор Python в со-  
ответствии с настройками пути поиска (то есть в большинстве команд-  
ных оболочек UNIX поиск будет произведен во всех каталогах, перечис-  
ленных в переменной окружения PATH). Такой способ может оказаться   
более универсальным, так как он не требует жестко указывать во всех   
сценариях путь к каталогу, куда был установлен Python.   
Если у вас имеется доступ к программе env из любого места, ваши сцена-  
рии будут запускаться независимо от того, где находится интерпретатор   
Python, – вам достаточно будет лишь настроить переменную окружения   
PATH в своих системах, не внося исправления в первую строку всех сце-  
нариев. Безусловно, этот способ предполагает, что во всех системах про-  
грамма env находится в одном и том же каталоге (в некоторых системах   
она может располагаться в каталогах /sbin, /bin или где-то еще), в про-  
тивном случае о переносимости не может быть и речи.  
Щелчок на ярлыке файла  
Использование реестра в операционной системе �indows позволяет открывать   
файл просто щелчком мыши. При установке интерпретатор Python автомати-  
чески регистрирует себя в качестве программы, используемой для открытия   
файлов с программами на языке Python щелчком мыши. Это делает возмож-  
ным запуск программ на языке Python простым щелчком (или двойным щелч-  
ком) мыши на ярлыке файла.  
В операционных системах, отличных от �indows, наверняка имеется возмож-  
�indows, наверняка имеется возмож-  
, наверняка имеется возмож-  
ность реализовать аналогичный трюк, но сами ярлыки, программа просмотра   
файловой системы, система навигации и прочее могут несколько отличаться.   
В некоторых системах UNIX, например, может потребоваться зарегистриро-  
вать расширение .py в программе просмотра файловой системы, сделать свой

Щелчок на ярлыке файла   
91  
сценарий исполняемым файлом, использующим специальный комментарий   
#!, как обсуждалось в предыдущем разделе, или связать тип MIME файла   
с приложением или командой редактирования файлов за счет установки про-  
грамм или с помощью других инструментальных средств. Если щелчок мы-  
шью не дает нужного результата, обращайтесь к документации используемой   
программы просмотра файловой системы за дополнительной информацией.  
Щелчок на ярлыке в Windows  
Чтобы продемонстрировать эту возможность, мы продолжим использовать сце-  
нарий script1.py, созданный выше, но повторим его содержимое, чтобы вам не   
пришлось перелистывать страницы:   
# Первый сценарий на языке Python  
import sys # Загружает библиотечный модуль  
print(sys.platform)  
print(2 \*\* 100) # Возводит число 2 в степень 100  
x = ‘Spam!’  
print(x \* 8) # Дублирует строку  
Как мы уже видели, этот файл всегда можно запустить из командной строки   
системы:  
C:\misc> c:\python30\python script1.py  
win32  
1267650600228229401496703205376  
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!  
Однако эту программу можно также запустить щелчком мыши, вообще ничего   
не вводя с клавиатуры. Можно попробовать отыскать ярлык этого файла, на-  
пример выбрав пункт Компьютер (Computer) (Мой компьютер (My Computer) – в �in-  
�in-  
dows XP) в меню Пуск (Start) и выполнив переход вглубь дерева каталогов на   
диске C. В этом случае вы получите изображение в проводнике, как показано   
на рис. 3.1 (этот снимок с экрана был получен в �indows Vista). Ярлыки фай-  
�indows Vista). Ярлыки фай-  
 Vista). Ярлыки фай-  
Vista). Ярлыки фай-  
). Ярлыки фай-  
лов с исходными текстами программ на языке Python содержат изображение   
с текстом на белом фоне, а ярлыки с байт-кодом – изображение с текстом на   
черном фоне. Чаще вам придется щелкать (то есть запускать) на файлах с ис-  
ходными текстами, чтобы увидеть последние изменения. Для запуска файла   
здесь нужно просто щелкнуть на ярлыке файла script1.py.  
Трюк с использованием функции input  
К сожалению, запуск файла в �indows щелчком на ярлыке может не приве-  
�indows щелчком на ярлыке может не приве-  
 щелчком на ярлыке может не приве-  
сти к удовлетворяющему вас результату. В действительности запуск сценария,   
как в данном примере, вызывает появление окна на очень короткое время, чего   
явно недостаточно для обеспечения обратной связи, на которую так надеются   
программисты, использующие язык Python! Это не ошибка, это лишь особен-  
ность обслуживания вывода программы в �indows.  
По умолчанию интерпретатор Python открывает черное окно консоли DOS, ко-  
торое будет служить местом для ввода и вывода программы. Если сценарий вы-  
водит какое-то сообщение и завершает работу, что и происходит в приведенном   
примере, то окно консоли открывается, туда выводится текст сообщения, но по   
завершении программы окно закрывается и исчезает. Вы сможете увидеть этот

92   
Глава 3. Как пользователь запускает программы   
текст, только если вы обладаете мгновенной реакцией или ваш компьютер не   
отличается высокой скоростью работы. Это вполне нормальное поведение, но   
скорее всего это совсем не то, что вы имели в виду.  
К счастью, этот недостаток легко ликвидируется. Если вам требуется, чтобы   
результаты работы сценария оставались на экране после щелчка мышью на яр-  
лыке файла, просто добавьте вызов встроенной функции input в самом конце   
сценария (в Python 2.6 следует использовать функцию raw\_input� смотрите при-  
мечание ниже). Например:  
# Первый сценарий на языке Python  
import sys # Загружает библиотечный модуль  
print(sys.platform)  
print(2 \*\* 100) # Возводит число 2 в степень 100  
x = ‘Spam!’  
print(x \* 8) # Дублирует строку  
input() # <== Добавленная строка  
Вообще, функция input считывает следующую строку с устройства стандарт-  
ного ввода, ожидая ее, если она еще недоступна. В результате в данном случае   
сценарий приостанавливается, благодаря чему окно остается на экране, пока   
не будет нажата клавиша Enter, как показано на рис. 3.2.  
Теперь, когда я продемонстрировал этот трюк, вы должны иметь в виду, что   
прибегать к нему требуется только в операционной системе �indows, и только   
Рис. 3.1. В Windows файлы с программами на языке Python отображаются   
в проводнике в виде ярлыков и могут быть запущены двойным щелчком мыши   
(хотя при этом вы можете не увидеть текст, выводимый программой,    
и сообщения об ошибках)

Щелчок на ярлыке файла   
93  
если сценарий выводит какие-либо сообщения, и только если сценарий запу-  
скается щелчком мыши на ярлыке. Вызов функции следует добавлять в са-  
мый конец файлов верхнего уровня и только при выполнении всех трех пере-  
численных условий. Во всех остальных случаях нет смысла добавлять этот   
вызов (разве что вы просто обожаете нажимать клавишу Enter!).1 Несмотря на   
всю очевидность сказанного, это еще одна ошибка, которую часто допускают   
обучающиеся на моих курсах.  
Прежде чем двинуться дальше, следует отметить, что функция input – это   
средство для ввода информации� она дополняет инструкцию print, которая яв-  
ляется средством вывода. Она представляет собой простейший способ чтения   
данных, вводимых пользователем, и обладает более широкими возможностя-  
ми, чем было продемонстрировано в этом примере. Например, функция input:  
 •  
Может принимать строку в качестве аргумента, которая будет выводиться   
как подсказка (например, input(‘Press Enter to exit’))  
 •  
Возвращает сценарию текстовую строку (например, nextinput = input())  
 •  
Поддерживает возможность перенаправления ввода на уровне командной   
оболочки системы (например, python spam.py < input.txt), точно так же, как   
инструкция print поддерживает возможность перенаправления вывода  
Далее в этой книге мы найдем более интересное применение этой функции: на-  
пример, в главе 10 эта функция будет использована для организации интерак-  
тивного цикла.  
1   
Кроме того, существует возможность полностью подавить появление окна консоли   
DOS при запуске файлов щелчком мыши в �indows. Программы, имена файлов ко-  
торых имеют расширение .pyw, отображают только те окна, которые создаются са-  
мими сценариями, – окно консоли DOS при запуске таких файлов не открывается.   
Файлы с расширением .pyw – это обычные файлы .py с исходными текстами, которые   
в операционной системе �indows обслуживаются специальным образом. Главным   
образом они используются для сценариев Python, которые сами создают окна поль-  
зовательского интерфейса, что часто сочетается с организацией вывода в файлы ре-  
зультатов работы и сообщений об ошибках.  
Рис. 3.2. При запуске программы в Windows щелчком мыши на ярлыке   
можно увидеть результаты ее работы, если добавить вызов функции input()   
в самый конец сценария. Но делать это следует, только если это действи-  
тельно необходимо!

94   
Глава 3. Как пользователь запускает программы   
Примечание,  касающееся  различий  между  версиями: если вы   
пользуетесь версией Python 2.6 или более ранней, используйте   
функцию raw\_input() вместо функции input(). В Python 3.0 пер-  
Python 3.0 пер-  
 3.0 пер-  
вая была переименована в последнюю. С технической точки зре-  
ния, версия 2.6 также имеет функцию input, но она интерпрети-  
рует вводимые строки, как если бы они содержали программ-  
ный код, и потому она не может использоваться в данном кон-  
тексте (ввод пустой строки вызывает ошибку). В Python 3.0   
функция input (и raw\_input в Python 2.6) просто возвращают вве-  
Python 2.6) просто возвращают вве-  
 2.6) просто возвращают вве-  
денный текст в виде строки, никак не интерпретируя их. Сыми-  
тировать поведение функции input из Python 2.6 в Python 3.0   
можно с помощью конструкции eval(input()).  
Другие ограничения на запуск щелчком мыши  
Даже при использовании функции input запуск файлов щелчком мыши имеет   
некоторые недостатки. Вы можете не заметить появление сообщений об ошиб-  
ках. Если в ходе работы сценария возникает ошибка, текст сообщения о ней   
выводится в окно консоли, которое тут же закрывается. Хуже того, даже до-  
бавление вызова input не поможет в такой ситуации, потому что работа сце-  
нария будет прервана еще до того, как будет выполнен этот вызов. Другими   
словами, вам будет сложно определить, что именно пошло не так.  
Из-за этих ограничений способ запуска программы щелчком мыши на ярлыке   
лучше использовать уже после того, как сценарий будет полностью отлажен.   
Старайтесь использовать другие способы запуска программ, особенно в самом   
начале работы над ними, такие как запуск из командной строки системы и из   
IDLE (рассматривается ниже в разделе «Пользовательский интерфейс IDLE»).   
Благодаря этому вы сможете увидеть сообщения об ошибках и обычный вы-  
вод от сценария, не прибегая к разного рода хитростям. Когда позднее в этой   
книге мы будем рассматривать исключения, вы узнаете, что существует воз-  
можность перехватывать и обрабатывать ошибки так, чтобы они не приводи-  
ли к аварийному завершению программы. Обратите внимание на приведенное   
ниже обсуждение инструкции try, которая предоставляет альтернативный   
способ предотвратить преждевременное закрытие окна в случае возникнове-  
ния ошибок.  
Импортирование и перезагрузка модулей  
Мы уже говорили об «импортировании модулей», но до сих пор я не давал ни-  
каких пояснений, что означает этот термин. Подробно о модулях и об архитек-  
туре крупных программ мы будем говорить в пятой части книги, но так как   
операция импорта модулей – это еще один из способов запуска программ, мы   
рассмотрим в этом разделе основы модулей, чтобы дать вам начальное пред-  
ставление о них.  
Проще говоря, каждый файл с исходным текстом на языке Python, имя кото-  
рого оканчивается расширением .py, является модулем. Другие файлы могут   
обращаться к программным компонентам, объявляемым модулем, импорти-

Импортирование и перезагрузка модулей   
95  
руя этот модуль. По сути инструкция import выполняет загрузку другого файла   
и обеспечивает доступ к его содержимому. Содержимое модуля становится до-  
ступным внешнему миру через его атрибуты (определение этого термина я дам   
в следующем разделе).  
Такая модульная модель является центральной идеей, лежащей в основе ар-  
хитектуры программ на языке Python. Крупные программы обычно организо-  
ваны в виде множества файлов модулей, которые импортируют и используют   
функциональные возможности из других модулей. Один из модулей определя-  
ется как основной файл верхнего уровня, который запускает всю программу.  
Проблемы модульной архитектуры мы будем рассматривать подробнее поз-  
же, в этой же книге, а в этой главе основное внимание уделяется тому факту,   
что операция импорта на заключительном этапе приводит к выполнению про-  
граммного кода загружаемого файла. Как следствие, импорт файла является   
еще одним способом запустить его.  
Например, если запустить интерактивный сеанс работы с интерпретатором   
(в IDLE, из командной строки или как-то иначе), можно будет запустить файл   
script1.py, созданный ранее, с помощью простой инструкции import (не забудь-  
те перед этим удалить инструкцию input, добавленную в предыдущем разделе,   
иначе вам придется нажимать клавишу Enter без всякой необходимости):  
C:\misc> c:\python30\python  
>>> import script1  
win32  
1267650600228229401496703205376  
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!  
Такой способ пригоден только для однократного запуска модуля в течение се-  
анса. После первой операции импорта все последующие попытки импортиро-  
вать модуль не приводят ни к каким результатам, даже если изменить и сохра-  
нить исходный текст модуля в другом окне:  
>>> import script1  
>>> import script1  
Так сделано преднамеренно – операция импорта требует слишком больших за-  
трат вычислительных ресурсов, чтобы выполнять ее более одного раза в ходе   
выполнения программы. Как вы узнаете в главе 21, в ходе импорта произво-  
дится поиск файлов, компиляция их в байт-код и выполнение этого байт-кода.  
Если действительно возникает необходимость вынудить интерпретатор мно-  
гократно запускать файл в рамках одного и того же сеанса (без остановки   
и перезапуска сеанса), можно воспользоваться встроенной функцией reload,   
доступной в модуле imp из стандартной библиотеки (в Python 2.6 эта функ-  
Python 2.6 эта функ-  
 2.6 эта функ-  
ция была обычной встроенной функцией, но в Python 3.0 она была перенесена   
в модуль imp):  
>>> from imp import reload # В версии 3.0 требуется загрузить функцию  
>>> reload(script1)  
win32  
65536  
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!  
<module ‘script1’ from ‘script1.py’>  
>>>

96   
Глава 3. Как пользователь запускает программы   
Инструкция from в этом примере просто копирует имя функции из модуля (под-  
робнее об рассказывается ниже). Функция reload загружает и запускает теку-  
щую версию программного кода в файле, если он был изменен в другом окне.   
Это позволяет редактировать и использовать новый программный код в ходе   
одного и того же интерактивного сеанса работы с интерпретатором Python.   
В этом сеансе, например, уже после того как модуль был импортирован, вторая   
инструкция print в файле script1.py была изменена в другом окне так, чтобы   
она выводила результат выражения 2 \*\* 16, после чего была выполнена переза-  
грузка модуля с помощью функции reload.  
Функция reload ожидает получить имя уже загруженного модуля, поэтому,   
прежде чем перезагрузка станет возможной, модуль должен быть импортиро-  
ван. Примечательно также, что имя модуля при вызове функции reload долж-  
но быть заключено в круглые скобки, тогда как инструкция import не требует   
этого. Дело в том, что reload – это функция, которая вызывается, а import – это   
инструкция.   
Именно поэтому имя модуля следует передавать функции reload как аргумент,   
в круглых скобках и именно поэтому после перезагрузки модуля выводится до-  
полнительная строка. Последняя строка в выводе выше – это всего лишь пред-  
ставление результата, возвращаемого функцией reload после перезагрузки мо-  
дуля. Более подробно функции будут обсуждаться в главе 16.  
Примечание,  касающееся  различий  между  версиями: в версии   
Python 3.0 встроенная функция reload была перемещена в мо-  
дуль imp из стандартной библиотеки. Она точно так же переза-  
гружает файлы, как и прежде, но перед использованием ее не-  
обходимо импортировать. В версии 3.0 можно использовать ин-  
струкцию import imp и затем вызывать функцию как imp.reload(M),   
или использовать инструкцию from imp import reload и вызывать   
функцию как reload(M), как показано в примере выше. Инструк-  
ции import и from рассматриваются в следующем разделе, а более   
формально они описываются далее в книге.  
Если вы используете версию Python 2.6 (или другую версию из   
ветки 2.X), функция reload доступна как встроенная функция,   
которую не нужно импортировать. В Python 2.6 функция reload   
доступна в двух видах – как встроенная функция и как функ-  
ция в модуле imp, чтобы упростить переход на версию 3.0. Дру-  
гими словами, возможность перезагрузки модулей сохранилась   
в версии 3.0, но чтобы воспользоваться ею, функцию reload не-  
обходимо импортировать.  
Перемещение функции в версии 3.0, вероятно, отчасти было вы-  
звано известными проблемами, связанными с инструкциями re-  
load и from, с которыми мы познакомимся в следующем разделе.   
В двух словах, имена, загруженные с помощью инструкции from,   
не обновляются вызовом функции reload, а имена, загруженные   
инструкцией import, – обновляются. Если вы обнаружите, что   
какие-то импортированные компоненты не обновляются после   
вызова функции reload, попробуйте использовать инструкцию   
import и обращаться к компоненту в формате module.attribute.

Импортирование и перезагрузка модулей   
97  
Важные сведения о модулях: атрибуты  
Операции импортирования и перезагрузки модулей обеспечивают естествен-  
ный способ запуска программы, так как на заключительном этапе этих опера-  
ций производится исполнение файлов. При этом в более широком понимании   
модули играют роль библиотек инструментов, как вы узнаете в пятой части   
книги. Модуль – это, главным образом, всего лишь пакет имен переменных,   
известный как пространство имен. Имена внутри этого пакета называются   
атрибутами, то есть атрибут – это имя переменной, которая связана с опреде-  
ленным объектом (таким как модуль).  
В самом типичном случае импортирующий программный код получает доступ   
ко всем именам верхнего уровня, определяемым в файле модуля. Эти имена   
обычно связаны с функциональными возможностями, экспортируемыми мо-  
дулем – функциями, классами, переменными и так далее, которые предна-  
значены для использования в других файлах и программах. Снаружи доступ   
к именам в файле модуля можно получить с помощью двух инструкций языка   
Python, import и from, а также с помощью вызова функции reload.  
Для иллюстрации вышесказанного создайте с помощью текстового редактора   
однострочный файл модуля Python с именем myfile.py со следующим содержи-  
мым:  
title = “The Meaning of Life”  
Это, пожалуй, один из самых простых модулей Python (он содержит един-  
ственную операцию присваивания), но его вполне достаточно для иллюстра-  
ции основных положений. При импортировании этого модуля выполняется его   
программный код, который создает атрибут модуля. Инструкция присваива-  
ния создает атрибут с именем title.  
Доступ к атрибуту title можно получить из других программных компонентов   
двумя разными способами. Первый заключается в том, чтобы загрузить мо-  
дуль целиком с помощью инструкции import, а затем обратиться к атрибуту по   
его имени, уточнив его именем модуля:  
% python # Запуск интерпретатора Python  
>>> import myfile # Запуск файла; модуль загружается целиком  
>>> print(myfile.title) # Имя атрибута, уточненное именем модуля через ‘.’   
The Meaning of Life  
Вообще синтаксис точечной нотации в виде object.attribute позволяет получить   
доступ к любому атрибуту в любом объекте, и этот прием широко используется   
в программном коде на языке Python. Здесь мы использовали его для обраще-  
ния к строковой переменной title, определенной внутри модуля myfile – то есть   
myfile.title.  
Кроме того, доступ к именам внутри модулей (Фактически создать копии имен)   
можно получать с помощью инструкции from:  
% python # Запуск интерпретатора Python  
>>> from myfile import title # Запуск файла; выполняется копирование имен  
>>> print(title) # Имя атрибута используется напрямую, уточнение не требуется  
The Meaning of Life

98   
Глава 3. Как пользователь запускает программы   
Как будет говориться позднее, инструкция from во многом подобна инструкции   
import, которая выполняет присваивание имен в импортируемом компоненте.   
С технической точки зрения, инструкция from копирует атрибуты модуля   
так, что они становятся простыми переменными в программном коде, выпол-  
няющем импорт, благодаря чему на этот раз он может обратиться к импорти-  
рованной строке уже не по имени myfile.title (ссылка на атрибут), а просто –   
title (переменная).1  
Неважно, как выполняется импортирование модуля, с помощью инструкции   
import или from, в любом случае это приводит к выполнению инструкций в фай-  
ле myfile.py, а импортирующий компонент (в данном случае – интерактивная   
оболочка интерпретатора) получает доступ к именам, определенным в фай-  
ле на верхнем уровне. В этом простом примере существует только одно такое   
имя – переменная title, которой присвоена строка, но сама концепция приоб-  
ретает более важное значение, когда речь заходит об определении в модулях   
таких объектов, как функции и классы. Такие объекты становятся программ-  
ными компонентами многократного использования, доступ к которым можно   
получить из одного или более клиентских модулей.  
На практике модули обычно определяют более чем одно имя, которые могут   
использоваться и внутри, и за пределами модуля. Ниже приводится пример   
модуля, в котором определяются три имени:  
a = ‘dead’ # Определяются три атрибута,  
b = ‘parrot’ # экспортируемые другим модулям  
c = ‘sketch’  
print a, b, c # Кроме того, они используются и самим этим модулем  
В файле threenames.py создаются три переменные, которые становятся тремя   
атрибутами, доступными внешнему миру. Этот модуль сам также использует   
эти переменные в инструкции print, в чем можно убедиться, если запустить   
этот модуль как файл верхнего уровня:  
% python threenames.py  
dead parrot sketch  
Как обычно, программный код этого модуля выполняется всего один раз, при   
импортировании (с помощью инструкции import или from). Клиенты, исполь-  
зующие инструкцию import, получают модуль со всеми его атрибутами, а кли-  
енты, использующие инструкцию from, получают копии имен из этого модуля:  
% python  
>>> import threenames # Загрузить модуль целиком  
dead parrot sketch  
>>>  
>>> threenames.b, threenames.c  
(‘parrot’, ‘sketch’)  
1   
Обратите внимание: в обеих инструкциях, import и from, имя модуля myfile указыва-  
ется без расширения .py. Как вы узнаете в пятой части книги, когда интерпретатор   
Python выполняет поиск файлов модулей, он знает, что к имени модуля необходимо   
добавить расширение. Не забывайте, что расширение обязательно должно указы-  
ваться при вызове файла в системной командной оболочке и опускаться в инструк-  
циях import.

Импортирование и перезагрузка модулей   
99  
>>>  
>>> from threenames import a, b, c # Скопировать несколько имен  
>>> b, c  
(‘parrot’, ‘sketch’)  
Результаты здесь выводятся в круглых скобках, потому что в действительно-  
сти они являются кортежами (разновидность объектов, которая описывается   
в следующей части книги). Пока вы можете спокойно игнорировать эту особен-  
ность.  
Как только вы начнете создавать модули, содержащие несколько имен, как   
в данном случае, вам наверняка пригодится встроенная функция dir. Она мо-  
жет использоваться для получения списка имен, доступных внутри модуля.   
Следующая инструкция возвращает список строк (мы начнем знакомиться со   
списками в следующей главе):  
>>> dir(threenames)  
[‘\_\_builtins\_\_’, ‘\_\_doc\_\_’, ‘\_\_file\_\_’, ‘\_\_name\_\_’, ‘a’, ‘b’, ‘c’]  
Такой результат получается при вызове функции в Python 3.0 и 2.6� в более   
ранних версиях количество возвращаемых имен может быть меньше. При вы-  
зове функции dir передается имя импортированного модуля в круглых скоб-  
ках, как показано выше, а возвращает она список всех атрибутов, определен-  
ных внутри модуля. Некоторые возвращаемые имена, которые начинаются   
и завершаются двумя символами подчеркивания, присутствуют всегда� эти   
встроенные имена определяются самим интерпретатором Python и имеют для   
него особый смысл. Имена переменных, которые определяются нашими ин-  
струкциями присваивания, – a, b и c – выводятся в конце списка, получаемого   
от функции dir.  
Модули и пространства имен  
Импортирование модулей – это один из способов запуска программного кода   
в файлах, но, помимо этого, и это будет рассмотрено в книге позже, модули яв-  
ляются также самой крупной структурной единицей в программах на языке   
Python.   
Вообще программы на языке Python состоят из множества файлов модулей,   
связанных между собой инструкциями import. Каждый файл модуля – это   
самостоятельный пакет переменных, или пространство имен. Один модуль   
не сможет увидеть переменные, определенные в другом модуле, если явно не   
импортирует его. Модули позволяют уменьшить вероятность конфликтов   
имен в программном коде – так как каждый файл является самостоятельным   
пространством имен, имена в одном файле не вступают в конфликт с именами   
в другом файле, даже если они одинаковые.  
Как можно понять, модули – одно из ухищрений, которые используются в язы-  
ке Python для упаковки переменных в категории, чтобы избежать конфликтов   
имен. Далее мы еще будем обсуждать модули и другие конструкции образова-  
ния пространств имен (включая классы и функции). А пока будем использо-  
вать модули в качестве средства многократного использования программного   
кода, позволяющего не вводить его повторно с клавиатуры.

100   
Глава 3. Как пользователь запускает программы   
import или from: необходимо отметить, что инструкция from в не-  
котором смысле стирает границы пространств имен между мо-  
дулями, потому что она копирует переменные из одного файла   
в другой. Это может вызывать затирание переменных в импор-  
тирующем файле одноименными переменными в импортируе-  
мом файле (при этом никаких предупреждений выводиться не   
будет). По сути, эта инструкция выполняет разрушительное   
объединение пространств имен, по крайней мере, в терминах ко-  
пируемых переменных.  
По этой причине многие рекомендуют использовать инструк-  
цию import вместо from. Я не буду вдаваться в глубокомысленные   
рассуждения, однако отмечу, что инструкция from не только ко-  
роче, но и подразумеваемая проблема редко возникает на прак-  
тике. Обратите внимание, что инструкция from позволяет явно   
указывать перечень импортируемых имен, и пока вы помните,   
что им будут присвоены значения, эта операция не более опас-  
на, чем обычная инструкция присваивания, – еще одна возмож-  
ность, которая наверняка будет использоваться вами!  
import и reload, примечания к использованию  
Зачастую, узнав о возможности запуска файлов с помощью import и reload, на-  
чинающие разработчики концентрируют все свое внимание на этом способе   
и забывают о других возможностях запуска, позволяющих запускать всегда   
самую свежую версию программного кода (например, щелчок мышью на яр-  
лыке, пункты меню в IDLE и системная командная строка). К тому же, такой   
подход может быстро привести к появлению ошибок – вам придется помнить,   
импортировали ли вы тот или иной модуль, чтобы иметь возможность переза-  
грузить его� вам нужно будет помнить о необходимости использовать круглые   
скобки при вызове функции reload (только для нее) и не забывать использовать   
ее, чтобы запустить самую последнюю версию модуля. Более того, операция   
перезагрузки не является транзитивной – перезагружается только модуль,   
указанный в вызове функции reload, но не перезагружаются модули, которые   
он импортирует, поэтому может возникнуть потребность перезагрузить не-  
сколько файлов.  
Из-за этих сложностей (и некоторых других, с которыми мы еще столкнемся   
позднее) пока лучше избегать пользоваться операциями импорта и перезагруз-  
ки. Пункт меню Run (Запустить) → Run Module (Запустить модуль) в IDLE (описы-  
вается в следующем разделе), например, предоставляет более простой способ   
запуска файлов, менее подверженный ошибкам. Системная командная строка   
предлагает похожие удобства. Вам не придется выполнять перезагрузку при   
использовании этих приемов.  
Следует добавить, что в случае использования необычных способов примене-  
ния модулей, отличных от тех, которые к этому моменту описаны в книге, вы   
можете столкнуться с некоторыми неприятностями. Например, если вам необ-  
ходимо импортировать файл модуля, который хранится в каталоге, отличном   
от того, в котором вы работаете, вам придется перейти к главе 21, где вы узнае-  
те о пути поиска модулей.

Запуск модулей с помощью функции exec   
101  
А пока, чтобы избежать осложнений, храните все импортируемые файлы мо-  
дулей в рабочем каталоге.1  
Следует также отметить, что операции импортирования и перезагрузки ста-  
ли популярной методикой тестирования классов, и вполне возможно, что вам   
также понравится этот подход. Но если начнут возникать сложности – остано-  
витесь!  
Запуск модулей с помощью функции exec   
В действительности существует еще несколько способов выполнить программ-  
 действительности существует еще несколько способов выполнить программ-  
действительности существует еще несколько способов выполнить программ-  
 существует еще несколько способов выполнить программ-  
существует еще несколько способов выполнить программ-  
 еще несколько способов выполнить программ-  
еще несколько способов выполнить программ-  
 несколько способов выполнить программ-  
несколько способов выполнить программ-  
 способов выполнить программ-  
способов выполнить программ-  
 выполнить программ-  
выполнить программ-  
 программ-  
программ-  
ный код, хранящийся в файлах модулей. Например, вызов встроенной функ-  
 код, хранящийся в файлах модулей. Например, вызов встроенной функ-  
код, хранящийся в файлах модулей. Например, вызов встроенной функ-  
, хранящийся в файлах модулей. Например, вызов встроенной функ-  
хранящийся в файлах модулей. Например, вызов встроенной функ-  
 в файлах модулей. Например, вызов встроенной функ-  
в файлах модулей. Например, вызов встроенной функ-  
 файлах модулей. Например, вызов встроенной функ-  
файлах модулей. Например, вызов встроенной функ-  
 модулей. Например, вызов встроенной функ-  
модулей. Например, вызов встроенной функ-  
. Например, вызов встроенной функ-  
Например, вызов встроенной функ-  
ции exec(open(‘module.py’).read()) – это еще один способ выполнять файлы из   
интерактивной оболочки, фактически не импортируя модуль. Каждый по-  
следующий вызов exec будет выполнять текущую версию файла и ликвиди-  
рует необходимость позднее выполнять перезагрузку модуля (возьмем опять   
сценарий script1.py в том виде, в каком мы оставили его после перезагрузки   
в предыдущем разделе):  
C:\misc> c:\python30\python  
>>> exec(open(‘script1.py’).read())  
win32  
65536  
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!  
   
...изменим script1.py в текстовом редакторе...  
   
>>> exec(open(‘script1.py’).read())  
win32  
4294967296  
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!  
Вызов функции exec производит эффект, похожий на вызов инструкции import,   
но при этом он не импортирует модуль – по умолчанию всякий раз, когда вызы-  
вается функция call, она выполняет файл заново, как если бы он был вставлен   
в месте вызова функции exec. По этой причине при использовании функции   
exec не требуется перезагружать модуль после внесения в него изменений – она   
не следует обычной логике импортирования.  
Однако, так как вызов exec по своему действию напоминает простую вставку   
программного кода модуля на его место, подобно инструкции from, упоминав-  
шейся выше, он может без предупреждения затереть существующие перемен-  
ные. Например, в нашем сценарии script1.py выполняется присваивание значе-  
1   
Для тех, кто сгорает от любопытства, скажу, что интерпретатор Python выполняет   
поиск импортируемых модулей во всех каталогах, перечисленных в переменной sys.  
path, – в списке имен каталогов, определенном в модуле sys, который инициализи-  
руется значением переменной окружения PYTHONPATH, и в наборе стандартных имен   
каталогов. Если возникает потребность импортировать модули из других каталогов,   
отличных от того, в котором вы работаете, они должны быть перечислены в перемен-  
ной PYTHONPATH. За дополнительной информацией обращайтесь к главе 21.

102   
Глава 3. Как пользователь запускает программы   
ния переменной x. Если это имя уже используется к моменту вызова функции   
exec, значение переменной с этим именем будет затерто:  
>>> x = 999  
>>> exec(open(‘script1.py’).read()) # Код выполняется в этом же   
 # пространстве имен  
...тот же самый вывод...  
   
>>> x # Присваивание в модуле затерло прежнее значение  
‘Spam!’  
Инструкция import, напротив, выполняет файл только один раз за все время   
выполнения программы и создает отдельное пространство имен модуля, поэто-  
му подобные операции присваивания не приводят к затиранию значений пере-  
менных в импортирующем программном коде. Однако за удобства, которые   
несут пространства имен, приходится платить необходимостью перезагружать   
модули после их изменения.  
Примечание,  касающееся  различий  между  версиями: в версии   
Python 2.6 кроме всего прочего имеется встроенная функция   
execfile(‘module.py’), которая автоматически читает содержимое   
файла, как и вызов exec(open(‘module.py’)). Оба эти вызова можно   
имитировать вызовом exec(open(‘module.py’).read()), который   
хоть и более сложный, но может использоваться в обеих версиях   
интерпретатора, 2.6 и 3.0.  
К сожалению, ни одна из простых форм вызова не доступна в вер-  
сии 3.0, поэтому, чтобы полностью понять, как действует этот   
прием, вам необходимо знать, что такое объекты файлов и их ме-  
тоды чтения (увы, похоже, что это один из примеров нарушения   
эстетики в версии 3.0). Форма использования exec в версии 3.0   
выглядит слишком длинной и сложной, поэтому самый лучший   
совет, какой только можно дать, – вообще не использовать ее,   
а запускать файлы с помощью команд системной оболочки или   
с помощью меню в IDLE, как описывается в следующем разделе.   
Дополнительная информация о применении формы запуска на   
основе функции exec в версии 3.0 приводится в главе 9.  
Пользовательский интерфейс IDLE  
До сих пор мы рассматривали запуск программного кода Python с помощью   
интерактивной оболочки интерпретатора, системной командной строки,   
с помощью щелчка мышью на ярлыке, с использованием операции импорта   
и функции exec. Если вам требуется более наглядный подход, программа IDLE   
может предложить вам графический интерфейс пользователя (ГИП) для раз-  
работки программ на языке Python� IDLE является стандартной и свободно   
распространяемой частью системы Python. Обычно она называется интегриро-  
ванной средой разработки (integrated development environment, IDE), потому   
что позволяет решать разнообразные задачи в единой оболочке.1  
1   
Официально название IDLE считается искаженной аббревиатурой IDE, но в действи-  
тельности она была названа так в честь члена труппы цирка Монти Пайтона (Monty   
Python) – Эрика Айдла (Eric Idle).

Пользовательский интерфейс IDLE   
103  
Проще говоря, IDLE – это набор инструментальных средств с графическим ин-  
терфейсом, который способен работать на самых разных платформах, включая   
Microsoft �indows, X �indow (в Linux, UNIX и других UNIX-подобных опе-  
�indows, X �indow (в Linux, UNIX и других UNIX-подобных опе-  
, X �indow (в Linux, UNIX и других UNIX-подобных опе-  
X �indow (в Linux, UNIX и других UNIX-подобных опе-  
 �indow (в Linux, UNIX и других UNIX-подобных опе-  
�indow (в Linux, UNIX и других UNIX-подобных опе-  
 (в Linux, UNIX и других UNIX-подобных опе-  
рационных системах) и Mac OS (включая версии Classic и OS X). Для многих   
IDLE представляет собой удобную альтернативу командной строке, а также   
альтернативу способу запуска щелчком мыши.  
Основы IDLE  
Давайте начнем с примера. Запуск IDLE в операционной системе �indows не   
вызывает проблем – для нее создается отдельный пункт в разделе Python меню   
кнопки Пуск (Start) (см. рис. 2.1), а кроме того, ее можно запустить, выбрав   
пункт контекстного меню, щелкнув правой кнопкой мыши на ярлыке про-  
граммы, написанной на языке Python. В некоторых UNIX-подобных системах   
для запуска начального сценария IDLE может потребоваться использовать ко-  
мандную строку или щелкнуть мышью на ярлыке файла idle.pyw или idle.py,   
размещенного в подкаталоге idlelib в каталоге Lib, где установлен интерпрета-  
тор Python. В �indows IDLE является сценарием Python, который по умолча-  
Python. В �indows IDLE является сценарием Python, который по умолча-  
. В �indows IDLE является сценарием Python, который по умолча-  
IDLE является сценарием Python, который по умолча-  
 является сценарием Python, который по умолча-  
Python, который по умолча-  
, который по умолча-  
нию находится в каталоге C:\Python30\Lib\idlelib (или C:\Python26\Lib\idlelib   
в Python 2.6).1   
На рис. 3.3 показано, как выглядит среда IDLE, запущенная в операционной   
системе �indows. Окно с заголовком Python Shell (Оболочка Python), которое от-  
крывается первоначально, является основным окном среды, в котором запу-  
скается интерактивный сеанс работы с интерпретатором (обратите внимание   
на приглашение к вводу >>>). Это самый обычный интерактивный сеанс, кото-  
рый играет роль инструмента проведения экспериментов – программный код,   
который здесь вводится, исполняется немедленно.  
В IDLE присутствуют привычные пункты меню, а для выполнения наиболее   
распространенных операций можно использовать короткие комбинации кла-  
виш. Чтобы создать (или отредактировать) файл с исходным программным ко-  
дом в среде IDLE, откройте окно текстового редактора: в главном окне откройте   
меню File (Файл) и выберите пункт New Window (Новое окно), чтобы открыть окно   
текстового редактора (или Open… (Открыть) – чтобы отредактировать существу-  
ющий файл).   
В книге это недостаточно четко видно, но IDLE обеспечивает подсветку синтак-  
сиса программного кода, который вводится как в главном окне, так и во всех   
окнах текстового редактора – ключевые слова выделяются одним цветом, ли-  
тералы другим цветом и так далее. Это позволяет визуально выделять элемен-  
ты программного кода. Это поможет вам различать синтаксические элементы   
программного кода (и даже поможет сразу же замечать ошибки – например,   
все строки здесь выделяются одним цветом).  
1   
IDLE – это программа на языке Python, которая создает графический интерфейс с по-  
мощью библиотеки tkinter GUI (Tkinter – в Python 2.6), что обеспечивает ее перено-  
симость, но также означает, что для использования IDLE вам придется обеспечить   
поддержку tkinter в Python. Версия Python для �indows обладает такой поддерж-  
кой по умолчанию, но некоторым пользователям Linux и UNIX может потребоваться   
установить соответствующую поддержку tkinter (для этого в некоторых дистрибу-  
тивах Linux можно использовать команду yum tkinter, более подробные сведения об   
установке вы найдете в приложении A). В системе Mac OS X все необходимое может   
быть уже установлено – поищите на своей машине команду idle.

104   
Глава 3. Как пользователь запускает программы   
Чтобы запустить файл с программным кодом в среде IDLE, выберите окно, где   
редактируется текст, раскройте меню Run (Запустить) и выберите в нем пункт   
Run Module (Запустить модуль) (или воспользуйтесь соответствующей этому пун-  
кту меню горячей комбинацией клавиш). Если с момента открытия или по-  
следнего сохранения файла его содержимое изменялось, Python предложит   
сохранить его.  
Когда сценарий запускается таким способом, весь вывод, который он генери-  
рует, а также все сообщения об ошибках появляются в основном окне интерак-  
тивного сеанса работы с интерпретатором (командная оболочка Python). На-  
пример, на рис. 3.3 последние три строки являются результатом выполнения   
нашего сценария script1.py, открытого в отдельном окне редактирования. Сооб-  
щение «RESTART» говорит о том, что пользовательский процесс был перезапу-  
щен с целью выполнить отредактированный сценарий, и позволяет визуально   
отделить вывод, полученный от сценария (это сообщение не появляется, если   
пользовательский программный код не был запущен в виде дочернего процес-  
са – подробнее об этом режиме рассказывается ниже).  
Рис. 3.3. Основное окно интерактивного сеанса работы с интерпретатором   
Python в интегрированной среде IDLE, запущенной под управлением опера-  
ционной системы Windows. Используйте меню «File», чтобы создать («New   
Window»), или изменить («Open…») файл с исходным программным кодом.   
Для запуска сценария, открытого в окне редактирования, используйте меню   
«Run» этого окна (пункт «Run Module»)

Пользовательский интерфейс IDLE   
105  
Совет дня: если вам потребуется повторно выполнить команду   
в основном окне интерактивного сеанса в среде IDLE, можно вос-  
пользоваться комбинацией Alt-P, выполняющей последователь-  
ный переход к началу истории команд, и Alt-N, выполняющей   
переход к концу истории команд (в некоторых системах Mac вме-  
сто этих комбинаций могут использоваться комбинации Ctrl-P   
и Ctrl-N). При нажатии этих комбинаций клавиш вы сможете вы-  
зывать предыдущие команды, изменять их и запускать. Кроме   
того, имеется возможность повторно вызывать команды, пози-  
ционируя в них курсор, и использовать операцию «скопировать   
и вставить», но часто этот прием оказывается более трудоемким,   
чем ввод вручную. Вне среды разработки IDLE в ходе интерак-  
тивного сеанса работы с интерпретатором в системе �in dows   
можно повторно возвращаться к предыдущим командам с помо-  
щью клавиш управления курсором.  
Использование IDLE  
Программа IDLE проста в использовании, переносима и доступна в большин-  
стве платформ. Я обычно рекомендую ее тем, кто только начинает програм-  
мировать на языке Python, потому что она упрощает некоторые аспекты и не   
предполагает наличие опыта работы с системной командной строкой. Но, по   
сравнению с некоторыми коммерческими интегрированными средами разра-  
ботки, она имеет некоторые ограничения. Ниже приводится список особенно-  
стей, которые должны приниматься во внимание начинающими пользовате-  
лями IDLE:  
 •  
При сохранении файлов необходимо явно добавлять расширение «.py».   
Я уже упоминал об этом, когда мы говорили о файлах вообще, но это самый   
распространенный камень преткновения, особенно для пользователей �in-  
�in-  
dows. Среда IDLE не выполняет автоматическое добавление расширения .py   
к именам сохраняемых файлов. Не забывайте добавлять расширение .py,   
когда сохраняете файл в первый раз. В противном случае, вы хотя и смо-  
жете запустить свой файл из среды IDLE (а также из системной командной   
строки), но не сможете импортировать его в интерактивную командную   
оболочку или в другой модуль.  
 •  
Запускайте сценарии, выбирая пункт меню Run (Запустить) → Run Module (За-  
пустить модуль) в окне редактирования, а не за счет их импортирования или   
перезагрузки в окне интерактивного сеанса. Ранее в этой главе было по-  
казано, что вполне возможно запустить файл, выполнив операцию импор-  
тирования в интерактивной оболочке интерпретатора. Однако такой способ   
несет определенные сложности, потому что он требует вручную выполнять   
перезагрузку файлов после внесения изменений. В противовес ему пункт   
меню Run (Запустить) → Run Module (Запустить модуль) всегда приводит к запу-  
ску текущей версии файла. Кроме того, в случае необходимости будет пред-  
ложено сохранить файл (еще одна распространенная ошибка при работе вне   
среды IDLE).  
 •  
Вам по-прежнему может потребоваться выполнять перезагрузку импор-  
тируемых модулей. Пункт меню Run (Запустить) → Run Module (Запустить мо-

106   
Глава 3. Как пользователь запускает программы   
дуль) в среде IDLE всегда запускает текущую версию только файла верхнего   
уровня. Если изменениям подвергались модули, импортируемые сценари-  
ем, их необходимо будет перезагрузить вручную в интерактивной оболочке.   
Но, несмотря на это, использование пункта меню Run (Запустить) → Run Module   
(Запустить модуль) позволяет избавиться от некоторых ошибок, связанных   
с операцией импортирования. Если вы предпочитаете использовать опера-  
ции импортирования и перезагрузки, не забывайте о комбинациях клавиш   
Alt-P и Alt-N, позволяющих возвращаться к ранее запускавшимся командам.  
 •  
Вы можете настроить IDLE. Чтобы изменить шрифты или цвета в IDLE,   
выберите пункт Configure (Настройка) в меню Options (Параметры) в любом окне   
IDLE. Кроме того, вы сможете настроить комбинации клавиш, настройки   
отступов и многое другое. Более подробные сведения вы сможете получить   
в меню Help (Справка) среды IDLE.  
 •  
В настоящее время в IDLE отсутствует возможность очистки экрана. По-  
хоже, что эта возможность является наиболее востребованной (возможно   
потому, что она присутствует в похожих интегрированных средах разра-  
ботки), и, в конечном счете, когда-нибудь она будет добавлена. Однако в на-  
стоящее время нет никакой возможности выполнить очистку окна инте-  
рактивного сеанса. Если вам потребуется очистить окно, вы можете нажать   
и удерживать некоторое время клавишу Enter или написать цикл на языке   
Python, который будет выводить последовательность пустых строк (конеч-  
но, никто в действительности не использует последний прием, но он выгля-  
дит более технологичным, чем простое удержание клавиши Enter в нажатом   
состоянии!).  
 •  
Многопоточные программы и программы с графическим интерфейсом на   
базе tkinter могут не работать со средой IDLE. Из-за того, что IDLE сама   
является программой Python/tkinter, она может зависать при запуске неко-  
Python/tkinter, она может зависать при запуске неко-  
/tkinter, она может зависать при запуске неко-  
tkinter, она может зависать при запуске неко-  
, она может зависать при запуске неко-  
торых типов программ на языке Python, использующих библиотеку tkint-  
Python, использующих библиотеку tkint-  
, использующих библиотеку tkint-  
tkint-  
er. В более свежих версиях IDLE проблем с этим стало меньше – благодаря   
тому, что пользовательский программный код запускается в виде одного   
процесса, а сам графический интерфейс IDLE работает в виде другого про-  
цесса, но некоторые программы по-прежнему могут вызывать зависание   
графического интерфейса IDLE. Ваш программный код может и не вызы-  
вать проблем такого рода, однако существует эмпирическое правило: вы без   
опаски можете использовать IDLE для редактирования исходных текстов   
программ с графическим интерфейсом, но для их запуска желательно ис-  
пользовать другие способы, например щелчком мыши на ярлыке или из си-  
стемной командной строки. Если ваш программный код не работает в IDLE,   
попробуйте запустить его за пределами среды разработки.  
 •  
Если возникают ошибки соединения, попробуйте запустить IDLE в виде   
единого процесса. Из-за того что для нормальной работы IDLE необходимо   
поддерживать взаимодействие между пользовательским процессом и гра-  
фическим интерфейсом среды разработки, на определенных платформах   
могут проявляться проблемы с запуском (особенно часто проблема с запу-  
ском встречается на некоторых машинах �indows). Если вам доведется   
столкнуться с такими ошибками, попробуйте запустить IDLE из командной   
строки, что вынудит ее запуститься в виде единственного процесса и позво-  
лит избежать проблем с поддержанием соединения: для принудительного   
запуска в этом режиме используйте флаг –n. Например, в операционной си-

Пользовательский интерфейс IDLE   
107  
стеме �indows откройте программу Командная строка (Command Prompt) и за-  
пустите команду idle.py –n из каталога C:\Python30\Lib\idlelib (перед этим,   
в случае необходимости, выполните команду cd).  
 •  
Остерегайтесь использования некоторых особенностей IDLE. Среда IDLE   
обладает множеством особенностей, облегчающих жизнь начинающим   
программистам, но некоторые из них невозможно использовать за предела-  
ми графического интерфейса среды разработки. Например, IDLE запуска-  
ет ваши сценарии в своем окружении, поэтому переменные, определяемые   
сценарием, автоматически становятся доступны в интерактивном сеансе   
IDLE – вам не придется запускать команду import, чтобы получить доступ   
к именам в файлах верхнего уровня, которые уже были запущены. Это мо-  
жет быть удобно, но может вызывать проблемы при работе вне среды IDLE,   
потому что в этом случае всегда необходимо импортировать имена из ис-  
пользуемых файлов.  
Кроме того, IDLE автоматически переходит в каталог, где находится за-  
IDLE автоматически переходит в каталог, где находится за-  
 автоматически переходит в каталог, где находится за-  
пускаемый файл и добавляет свой каталог в путь поиска модулей, что по-  
зволяет импортировать файлы без дополнительной настройки пути поис-  
ка. Эта особенность будет недоступна при запуске сценариев за пределами   
IDLE. Нет ничего предосудительного, если вы будете пользоваться этой воз-  
. Нет ничего предосудительного, если вы будете пользоваться этой воз-  
можностью, но не забывайте, что она доступна только в IDLE.  
Дополнительные возможности IDLE  
Помимо основных функций редактирования и запуска среда IDLE предостав-  
ляет целый ряд дополнительных возможностей, включая отладчик и инспек-  
тор объектов. Отладчик IDLE активируется с помощью меню Debug (Отладка),   
а инспектор объектов – с помощью меню File (Файл). Инспектор объектов позво-  
ляет переходить, перемещаясь по пути поиска модулей, к файлам и объектам   
в файлах – щелчок на файле или объекте приводит к открытию соответствую-  
щего исходного текста в окне редактирования.  
Режим отладки в IDLE инициируется выбором пункта меню Debug (Отладка) →   
Debugger (Отладчик) главного окна, после этого можно запустить отлаживаемый   
сценарий выбором пункта меню Run (Запустить) → Run Module (Запустить модуль)�   
как только отладчик будет активирован, щелчком правой кнопки мыши на   
выбранной строке в окне редактирования вы сможете устанавливать точки   
останова в своем программном коде, чтобы приостанавливать выполнение сце-  
нария, просматривать значения переменных и так далее. Кроме того, вы смо-  
жете следить за ходом выполнения программ – в этом случае текущая выпол-  
няемая строка программного кода выделяется цветом.  
Кроме того, в случае появления ошибок можно щелкнуть правой кнопкой   
мыши на строке с сообщением об ошибке и быстро перейти к строке программ-  
ного кода, которая вызвала эту ошибку. Это позволяет быстро выяснить источ-  
ник ошибки и ликвидировать ее. Помимо этого текстовый редактор IDLE обла-  
дает обширным набором возможностей, которые пригодятся программистам,   
включая автоматическое оформление отступов, расширенный поиск текста   
и файлов и многое другое. Интегрированная среда IDLE обеспечивает интуи-  
тивно понятный графический интерфейс, и потому вы можете поэксперимен-  
тировать с ней, чтобы получить представление об имеющихся возможностях.

108   
Глава 3. Как пользователь запускает программы   
Другие интегрированные среды разработки  
Из-за того что IDLE бесплатна, переносима и является стандартной частью   
Python, она прекрасно подходит на роль инструмента разработки, с которым   
следует познакомиться в первую очередь, если вы вообще собираетесь исполь-  
зовать интегрированную среду разработки. Я еще раз рекомендую использо-  
вать IDLE для выполнения упражнений из этой книги, если вы только начина-  
ете знакомство с языком Python и пока не знакомы с принципами разработки,   
основанными на применении командной строки. Однако существует еще не-  
сколько альтернативных средств разработки, и некоторые из них отличаются   
более высокой устойчивостью и обладают более широкими возможностями по   
сравнению с IDLE. Ниже приводятся некоторые из наиболее популярных инте-  
грированных сред разработки:  
Eclipse и PyDev  
Eclipse – это улучшенная и свободно распространяемая интегрированная   
среда разработки с графическим интерфейсом. Первоначально она соз-  
давалась как среда разработки программного кода на языке Java, но при   
установке модуля расширения PyDev (или подобного ему) она обеспечивает   
возможность разработки программ на языке Python. Eclipse – популярный   
и мощный инструмент для разработки программ на языке Python, возмож-  
ности которой намного шире возможностей IDLE. Одним из ее недостатков   
заключается в том, что она слишком велика, а модуль расширения PyDev   
для получения дополнительных возможностей (включая интегрированную   
интерактивную консоль) требует установки условно-бесплатных пакетов   
расширений, которые не являются свободно распространяемыми. Когда   
ваши потребности перерастут возможности IDLE, обратите внимание на   
комбинацию Eclipse/PyDev.  
Komodo  
Полнофункциональная среда разработки с графическим интерфейсом   
пользователя для Python (и других языков программирования), Komodo   
поддерживает такие возможности, как подсветка синтаксиса, редактиро-  
вание текста, отладка и другие. Кроме того, Komodo обладает множеством   
дополнительных возможностей, отсутствующих в IDLE, включая файлы   
проектов, интеграцию с системами контроля версий исходных текстов,   
отладку регулярных выражений и визуальный построитель графических   
интерфейсов, который генерирует программный код Python/tkinter, реали-  
tkinter, реали-  
kinter, реали-  
зующий графические интерфейсы, создаваемые в интерактивном режиме.   
К моменту написания этих строк среда Komodo не являлась свободно рас-  
пространяемой. Найти ее можно на сайте http://www.activestate.com.  
NetBeans IDE для Python  
NetBeans – это мощная, открытая среда разработки с графическим интер-  
 – это мощная, открытая среда разработки с графическим интер-  
фейсом, поддерживающая массу дополнительных возможностей, которые   
могут пригодиться разработчикам программ на языке Python: функцию   
дополнения программного кода, автоматическое оформление отступов   
и подсветку синтаксиса, вывод подсказок, сворачивание блоков кода, ре-  
факторинг, отладку, тестирование, создание проектов и многое другое. Она   
может использоваться как для разработки программ, выполняющихся   
под управлением CPython, так и для интерпретатора Jython. Как и в слу-  
CPython, так и для интерпретатора Jython. Как и в слу-  
, так и для интерпретатора Jython. Как и в слу-  
Jython. Как и в слу-  
. Как и в слу-  
чае с Eclipse, установка NetBeans является более сложной процедурой, чем   
установка IDLE GUI, но большинство согласится, что преимущества этой

Другие способы запуска   
109  
среды разработки стоят потраченных усилий. Последнюю информацию об   
этой среде и ссылки для загрузки вы без труда найдете в Интернете.  
PythonWin  
Python�in – это свободно распространяемая интегрированная среда раз-  
 – это свободно распространяемая интегрированная среда раз-  
работки на языке Python для операционной системы �indows. Она рас-  
Python для операционной системы �indows. Она рас-  
 для операционной системы �indows. Она рас-  
�indows. Она рас-  
. Она рас-  
пространяется в составе пакета ActivePython компании ActiveState (но ее   
можно также получить отдельно на сайте http://www.python.org). По своим   
возможностям она несколько напоминает IDLE и имеет несколько полез-  
IDLE и имеет несколько полез-  
 и имеет несколько полез-  
ных расширений, специфичных для �indows, например Python�in обла-  
дает поддержкой COM-объектов. В настоящее время IDLE обладает более   
широкими возможностями, чем Python�in (например, благодаря тому,   
что IDLE использует для своей работы два процесса, она реже зависает).   
Однако Python�in предлагает инструменты для разработки программ под   
�indows, которые отсутствуют в IDLE. Более подробную информацию вы   
найдете на сайте http://www.activestate.com.  
Прочие  
Существует еще примерно с полдесятка других известных мне интегриро-  
ванных сред разработки (например, WingIDE, PythonCard), и со временем   
их число будет увеличиваться. Фактически почти в каждом современном   
текстовом редакторе для программистов имеется поддержка языка Python,   
которая устанавливается вместе с редактором по умолчанию или в виде от-  
дельных расширений. Например, редакторы Emacs и Vim обладают суще-  
ственной поддержкой языка Python.   
Я не буду описывать все возможные варианты здесь, – вы сами можете   
узнать о них на сайте http://www.python.org или выполнив поиск в Google по   
строке «Python IDE» (IDE для Python). Можно также попробовать поискать   
по строке «Python editors» (редакторы Python) – это должно привести вас   
на страницу �iki, где содержится информация о множестве интегрирован-  
ных сред разработки и текстовых редакторов для Python.  
Другие способы запуска  
К настоящему моменту мы рассмотрели, как выполнять программный код   
в интерактивной командной оболочке интерпретатора и как запускать про-  
граммный код, сохраненный в файлах, из системной командной строки, из ис-  
полняемых сценариев в системе UNIX, щелчком мыши, с помощью операции   
импортирования модулей, с помощью функции exec и в интегрированной среде   
разработки, такой как IDLE. Это подавляющее большинство способов, кото-  
рые встретятся вам в этой книге. Однако существует еще ряд способов запуска   
программного кода на языке Python, большая часть которых имеет узкоспе-  
Python, большая часть которых имеет узкоспе-  
, большая часть которых имеет узкоспе-  
циализированное назначение. В следующих нескольких разделах мы коротко   
познакомимся с некоторыми из них.  
Встраивание вызовов  
В некоторых особых случаях программный код на языке Python может запу-  
скаться из других программ. В таких ситуациях мы говорим, что программы   
на языке Python встроены в другие программы (то есть запускаются другими   
программами). Сам программный код Python может храниться в текстовом   
файле, в базе данных, извлекаться из страницы HTML или из документа XML

110   
Глава 3. Как пользователь запускает программы   
и так далее. В этом случае уже не вы, а другая программа предлагает интерпре-  
татору выполнить программный код, созданный вами.   
Такой способ запуска программного кода обычно используется для обеспече-  
ния поддержки возможности настройки у конечного пользователя. Например,   
игровая программа может позволять изменять ход игры, запуская в ключевые   
моменты внедренный программный код на языке Python, доступный пользо-  
вателю. Поскольку программный код на языке Python интерпретируется,   
внесение изменений в этот программный код не требует перекомпилировать   
всю систему (о том, как интерпретатор выполняет программный код, расска-  
зывается в главе 2)  
В подобных случаях программа, вызывающая программный код на языке Py-  
Py-  
thon, может быть написана на языке C, C++ и даже Java, когда использует-  
, может быть написана на языке C, C++ и даже Java, когда использует-  
C, C++ и даже Java, когда использует-  
, C++ и даже Java, когда использует-  
C++ и даже Java, когда использует-  
++ и даже Java, когда использует-  
Java, когда использует-  
, когда использует-  
ся интерпретатор Jython. Например, вполне возможно создавать и выполнять   
строки программного кода Python из программ на языке C, вызывая функции   
API времени выполнения интерпретатора Python (набор служб, экспортируе-  
мых библиотеками, созданными при компиляции Python на вашей машине):  
#include <Python.h>  
...  
Py\_Initialize(); // Это язык C, а не Python  
PyRun\_SimpleString(“x = ‘brave ‘ + ‘sir robin’”); // Но он запускает код на   
 // языке Python   
В этом фрагменте программа на языке C, скомпонованная с библиотеками   
Python, инициализирует интерпретатор и передает ему для выполнения стро-  
ку с инструкцией присваивания. Программы на языке C могут также полу-  
чать доступ к объектам Python и взаимодействовать с ними, используя другие   
функции API языка Python.  
Эта книга не рассматривает вопросы интеграции Python/C, но вы должны   
знать, что в зависимости от того, как ваша организация планирует использо-  
вать Python, вы можете оказаться одним из тех, кому действительно придет-  
ся запускать программы на языке Python. При этом более чем вероятно, вы   
по-прежнему сможете использовать интерактивную оболочку интерпретатора   
и приемы запуска файлов, описанные выше, чтобы протестировать программ-  
ный код отдельно от программ, куда этот код внедряется. 1  
Фиксированные исполняемые двоичные файлы  
Фиксированные исполняемые двоичные файлы, описанные в главе 2, пред-  
ставляют собой комбинацию байт-кода программы и интерпретатора Python,   
объединенных в одном исполняемом файле. Благодаря этому такие програм-  
мы могут запускаться точно так же, как любые другие программы (щелчком   
на ярлыке, из командной строки и другими способами). Такая возможность   
замечательно подходит для случая распространения готовых программных   
1   
О встраивании программного кода Python в программы на языке C/C++ подробно   
рассказывается в книге «Программирование на Python» (СПб.: Символ-Плюс, 2002).   
Используя прикладной интерфейс встраиваемого интерпретатора, вы сможете на-  
прямую вызывать функции Python, загружать модули и производить прочие дей-  
ствия. Кроме того, следует отметить, что система Jython позволяет программам на   
языке Java вызывать программный код на языке Python, используя прикладной ин-  
терфейс на языке Java (класс интерпретатора Python).

Другие способы запуска   
111  
продуктов, но она не предназначена для использования в процессе разработки   
программ. Обычно фиксирование файлов производится непосредственно перед   
отправкой (когда разработка уже завершена) программы заказчику. Более под-  
робная информация об этой возможности приводится в предыдущей главе.  
Возможность запуска программ   
из текстового редактора  
Как упоминалось ранее, большинство текстовых редакторов для программи-  
стов, хотя и не являются полноценными интегрированными средами разра-  
ботки, тем не менее поддерживают возможность редактирования и запуска   
программ на языке Python. Такая поддержка может быть изначально встрое-  
на в редактор или доступна в виде расширений, которые можно загрузить из   
Сети. Например, если вы знакомы с текстовым редактором Emacs, вы сможете   
редактировать программный код на языке Python и запускать его, не покидая   
текстовый редактор. Дополнительную информацию о текстовых редакторах   
вы найдете на странице http://www.python.org/editors или поискав в Google по   
фразе «Python editors» (редакторы Python).  
Прочие возможности запуска  
В зависимости от используемой платформы могут существовать и другие спо-  
собы запуска программ Python. Например, в некоторых системах Macintosh   
выполнить программу на языке Python можно, перетащив мышью ярлык фай-  
ла программы на ярлык интерпретатора Python. В �indows сценарии можно   
запускать с помощью пункта Выполнить... (Run…) меню кнопки Пуск (Start). На-  
конец в состав стандартной библиотеки Python входят вспомогательные функ-  
ции, позволяющие запускать программы на языке Python из других программ   
на языке Python (такие, как os.popen, os.system), однако обсуждение этих функ-  
ций выходит за рамки этой главы.  
Будущие возможности  
В этой главе отражены существующие ныне способы запуска, многие из кото-  
рых характерны для определенной платформы и в определенное время. В дей-  
ствительности, многие методы запуска и выполнения, представленные здесь,   
появились между выпусками разных изданий этой книги. Поэтому вполне воз-  
можно, что в будущем появятся новые способы запуска.  
Новые операционные системы и новые версии существующих систем могут   
также обеспечивать способы запуска, не описанные здесь. Вообще, т. к. Python   
продолжает идти в ногу со временем, вы должны быть готовы запускать про-  
граммы на языке Python способом, имеющим смысл для машин, которые вы   
используете сейчас или будете использовать в будущем, – стилом планшетного   
или наладонного компьютера, захватывая ярлыки в виртуальной реальности   
или выкрикивая названия сценариев своим коллегам.  
Изменения в реализации также могут оказывать влияние на способ запуска   
(например, полноценный компилятор мог бы воспроизводить обычные испол-  
няемые файлы, которые запускаются так же, как и фиксированные двоичные   
файлы ныне). Если бы я знал, что будет в будущем, я бы, наверное, начал пере-  
говоры с биржевым маклером, а не писал бы сейчас этих слов!

112   
Глава 3. Как пользователь запускает программы   
Какие способы следует использовать?  
Ознакомившись с таким богатством возможностей, возникает вполне есте-  
ственный вопрос – какой способ лучше? Вообще, если вы начинающий раз-  
работчик, для вас было бы предпочтительнее использовать интегрированную   
среду разработки IDLE. Она предоставляет дружественный графический ин-  
терфейс и скрывает некоторые детали, связанные с необходимостью настрой-  
ки. Кроме того, в ее состав входит платформонезависимый текстовый редак-  
тор, предназначенный для создания сценариев, и она является стандартной   
и свободно распространяемой составляющей системы Python.  
С другой стороны, если вы опытный программист, для вас более комфортным   
может оказаться простой текстовый редактор, а для запуска программ – ис-  
пользовать командную строку системы или щелчок мышью на ярлыке (именно   
таким способом автор разрабатывает программы на языке Python, но это при-  
вычка, выработанная при работе с UNIX). Поскольку выбор среды разработки   
во многом зависит от личных предпочтений, я не могу сказать ничего, кроме   
стандартной рекомендации – лучшей средой разработки для вас будет та, ко-  
торая вам нравится.  
Отладка программ на языке Python  
Разумеется, ни один из моих читателей и студентов никогда не допуска-  
ет ошибок в программном коде (здесь можно улыбнуться), но ошибки   
могут допускать ваши менее удачливые друзья, поэтому здесь мы рас-  
смотрим краткий список стратегий, которые часто используются про-  
граммистами при отладке программ на языке Python:  
••  
Ничего не делать. Здесь я не имею в виду, что программисты не долж-  
ны отлаживать программный код, но, когда вы допускаете ошибку в   
программе, вы обычно получаете весьма информативное сообщение   
об ошибке (вы очень скоро увидите их, если еще не видели). Если вы   
уже знакомы с языком Python и это ваш собственный программный   
код, этих сообщений бывает вполне достаточно, чтобы найти нуж-  
ный файл, строку в нем и исправить ошибку. Для многих в этом и   
заключается отладка программ на языке Python. Однако при разра-  
ботке крупных систем этого может оказаться недостаточно.  
••  
Добавление инструкций print. Пожалуй, самый основной способ от-  
ладки, которым пользуются программисты (и я тоже пользуюсь им)   
заключается в том, чтобы вставить инструкции print и выполнить   
программу еще раз. Так как интерпретатор позволяет запустить про-  
грамму сразу после внесения изменений, этот прием обычно явля-  
ется самым быстрым способом получить дополнительную информа-  
цию сверх той, что содержится в сообщении об ошибке. Инструкции   
print не должны быть слишком сложными – вывода простой стро-  
ки «Я здесь» или отображения значений переменных обычно впол-  
не достаточно, чтобы понять причины ошибки. Только не забудьте   
удалить или закомментировать (то есть добавить символ # перед ин-  
струкцией) вывод отладочных сообщений, прежде чем передать про-  
грамму заказчику!

Какие способы следует использовать?   
113  
••  
Использование отладчиков в интегрированных средах разработки.   
При переходе к разработке крупных систем, и для начинающих про-  
граммистов, желающих проследить во всех подробностях, как вы-  
полняется программный код, можно порекомендовать использовать   
отладчики, встроенные в интегрированные среды разработки с гра-  
фическим интерфейсом. В IDLE также имеется собственный отлад-  
чик, но на практике он используется достаточно редко, скорее всего   
потому, что он не имеет командной строки, или потому, что прием,   
основанный на добавлении инструкций print обычно оказывается   
проще, чем запуск сеанса отладчика с графическим интерфейсом.   
Дополнительные подробности можно почерпнуть в меню Help (Справ-  
ка) программы IDLE или просто опробовать эту возможность само-  
стоятельно – базовый интерфейс программы описывается в разделе   
«Дополнительные возможности IDLE» выше. Другие среды разра-  
ботки, такие как Eclipse, NetBeans, Komodo и �ing IDE, также пред-  
лагают расширенные средства отладки. Если вы соберетесь пользо-  
ваться ими, обратитесь к документации этих программ.   
••  
Использование pdb – отладчика командной строки. В составе Python   
поставляется отладчик исходного программного кода pdb, доступный   
в виде модуля в стандартной библиотеке языка Python. При исполь-  
зовании pdb вы сможете выполнять программный код построчно,   
отображать значения переменных, устанавливать и сбрасывать точ-  
ки останова, возобновлять выполнение программы после остановки в   
контрольной точке и после ошибки и так далее. Отладчик pdb можно   
запустить в интерактивной оболочке, импортировав его или запустив   
его как самостоятельный сценарий. В любом случае вы получаете в   
свое распоряжение мощный инструмент отладки, позволяющий вво-  
дить команды. Кроме того, отладчик pdb позволяет производить по-  
ставарийную отладку уже после возникновения исключения, чтобы   
получить информацию об ошибке. Дополнительные подробности об   
использовании отладчика pdb вы найдете в руководстве к стандарт-  
ной библиотеке Python и в главе 35.  
••  
Другие возможности. Для удовлетворения более специфических   
требований, включая отладку многопоточных программ, внедряе-  
мого программного кода и уже запущенных процессов, можно поис-  
кать инструменты среди проектов, распространяемых с открытыми   
исходными текстами. Например, система Winpdb – автономный и   
платформонезависимый отладчик с расширенными возможностя-  
ми, обладающий как графическим интерфейсом, так и интерфейсом   
командной строки.  
Перечисленные здесь возможности приобретут большее значение, когда   
вы начнете писать большие сценарии. Но самое важное с точки зрения   
отладки состоит в том, что в случае наличия ошибки в программе эта   
ошибка не вызывает крах системы, а обнаруживается интерпретатором   
Python, который выводит информативное сообщение. В действительно-  
сти, сами ошибки – это четко определенный механизм, известный как   
исключения, которые можно перехватывать и обрабатывать (подробнее   
об исключениях рассказывается в седьмой части книги). Конечно же,

114   
Глава 3. Как пользователь запускает программы   
ошибки – это всегда неприятно, но по сравнению с прежними време-  
нами, когда отладка означала детальное изучение распечаток дампов   
памяти с шестнадцатеричным калькулятором в руках, поддержка воз-  
можностей отладки в Python делает поиск и исправление ошибок гораз-  
до менее болезненной процедурой.  
В заключение  
В этой главе мы познакомились с наиболее часто используемыми способами за-  
пуска программ на языке Python: запуск программного кода в интерактивном   
сеансе работы с интерпретатором и запуск файлов с программным кодом из си-  
стемной командной строки, щелчком мыши на ярлыке файла, за счет выпол-  
нения операции импортирования, с помощью функции exec и с помощью ин-  
терфейса интегрированной среды разработки, такой как IDLE. Мы охватили   
здесь значительную часть темы запуска. Цель этой главы состояла в том, что-  
бы дать вам достаточный объем знаний, владея которыми вы сможете присту-  
пить к работе с программным кодом, который мы начнем писать в следующей   
части книги. В этой части мы приступим к изучению самого языка Python, на-  
чав с базовых типов данных.  
Но перед этим ответьте на контрольные вопросы по теме, которую мы изучали   
здесь. Так как это последняя глава первой части книги, она завершается более   
сложными упражнениями, с помощью которых вы сможете проверить усвое-  
ние всех тем, рассматривавшихся в этой части. За справками и для того, чтобы   
освежить свои знания, обращайтесь к приложению B.  
Закрепление пройденного  
Контрольные вопросы  
1. Как запустить интерактивный сеанс работы с интерпретатором?  
2. Где следует вводить команду, которая запустит файл сценария?  
3. Назовите четыре или более способов запуска программного кода в файлах.  
4. Назовите две ловушки, связанные с щелчком мыши на ярлыках в �indows.  
5. Почему может потребоваться перезагрузить модуль?  
6. Как запустить сценарий из среды разработки IDLE?  
7. Назовите две ловушки, связанные со средой разработки IDLE.  
8. Что такое пространство имен, и какое отношение они имеют к файлам мо-  
дулей?  
Ответы  
1. В операционной системе �indows интерактивный сеанс работы с интер-  
претатором можно запустить, щелкнув на кнопке Пуск (Start), открыв пункт   
меню Все программы (All Programs), выбрав пункт меню Python и затем щелкнув   
на пункте меню Python (command line) (Python (командная строка)). Тот же ре-

Закрепление пройденного   
115  
зультат можно получить в �indows и на других платформах, введя коман-  
ду python в системной командной строке, в окне консоли (Командная строка   
(Command Prompt) в �indows). Как вариант можно запустить интегрирован-  
ную среду разработки IDLE, главное окно которой представляет собой ин-  
терактивную командную оболочку интерпретатора. Если в вашей системе   
переменная окружения PATH не включает каталог, в который был установ-  
лен интерпретатор Python, вам может потребоваться выполнить команду   
cd, чтобы перейти в каталог установки Python, или указать полный путь   
к команде python (например, C:\Python30\python в �indows).  
2. Системные команды вводятся в программе, которая в вашей системе ис-  
пользуется в качестве консоли: командная строка (Command Prompt) в �indows�   
xterm или окно терминала в UNIX, Linux и Nac OS X� и так далее.  
3. Программный код в файле сценария (фактически – модуля) можно запу-  
стить с помощью системной командной строки, щелкнув на ярлыке фай-  
ла, импортировав и перезагрузив модуль, с помощью встроенной функции   
exec и с помощью меню среды разработки, например выбрав пункт меню Run   
(Запустить) → Run Module (Запустить модуль) в программе IDLE. В UNIX сцена-  
рий можно также запустить, как обычную программу, воспользовавшись   
трюком со строкой, начинающейся с последовательности символов #!. Не-  
которые платформы поддерживают специализированные способы запуска   
(например, drag-and-drop). Кроме того, некоторые текстовые файлы обеспе-  
чивают собственный механизм запуска файлов с программным кодом на   
языке Python, некоторые программы на языке Python распространяются   
в виде автономных «фиксированных двоичных» выполняемых файлов,   
а некоторые системы обладают встроенной поддержкой выполнения про-  
граммного кода на языке Python, где он автоматически запускается про-  
граммами, написанными на таких языках, как C, C++ или Java. Послед-  
ний способ обычно используется для обеспечения возможности настройки   
систем под условия пользователя.  
4. Если сценарий просто выводит какие-то данные и завершает работу, окно   
с этой информацией исчезает немедленно, еще до того, как вы сможете уви-  
деть, что было выведено (поэтому в таких ситуациях удобно использовать   
функцию input). Сообщения об ошибках, возникших в ходе работы сцена-  
рия, также приводят к немедленному закрытию окна еще до того, как вы   
успеете исследовать его содержимое (поэтому предпочтительнее в ходе раз-  
работки использовать системную командную строку или среду разработки   
IDLE).  
5. Интерпретатор Python по умолчанию выполняет импорт (загрузку) модуля   
один раз за сеанс, поэтому, если вы изменили исходный текст модуля и вам   
необходимо запустить его новую версию, не покидая интерактивный сеанс,   
вам следует перезагрузить модуль. Однако прежде чем выполнить переза-  
грузку, модуль необходимо импортировать. Запуск программного кода из   
системной командной строки, щелчком мыши на ярлыке или в интегриро-  
ванной среде разработки, такой как IDLE, обычно вообще снимает эту про-  
блему, так как в таких случаях система каждый раз выполняет текущую   
версию программного кода.  
6. В окне редактирования текста для файла, который требуется запустить,   
выберите пункт меню Run (Запустить) → Run Module (Запустить модуль). В ре-  
зультате программный код в окне редактирования будет запущен как файл

116   
Глава 3. Как пользователь запускает программы   
сценария верхнего уровня, а вывод, сгенерированный с этим сценарием,   
появится в главном окне интерактивной командной оболочки Python.  
7. Интегрированная среда разработки IDLE может зависать при запуске неко-  
торых типов программ, особенно с графическим интерфейсом пользовате-  
ля, которые выполняются в нескольких потоках (рассмотрение усовершен-  
ствованной методики выходит за рамки данной книги). Кроме того, среда   
IDLE обладает некоторыми удобными особенностями, которые не поддер-  
живаются при работе вне этой среды: переменные сценария автоматически   
импортируются в интерактивную командную оболочку IDLE, но в команд-  
ной строке Python такая возможность отсутствует.  
8. Пространство имен – это просто пакет переменных (то есть имен). В Python   
он приобретает форму объекта с атрибутами. Каждый файл модуля авто-  
матически становится пространством имен, то есть пакетом переменных,   
отражающих выполненные операции присваивания на верхнем уровне   
файла. Пространства имен позволяют избежать конфликтов имен в про-  
граммах на языке Python: поскольку каждый модуль – это самостоятель-  
ное пространство имен, файлы должны явно импортировать другие файлы,   
чтобы использовать имена, определяемые в них.  
Упражнения к первой части  
Пришло время начинать писать программный код самостоятельно. Здесь пред-  
ставлены достаточно простые упражнения, но некоторые из поднимаемых   
вопросов связаны с темами, которые будут рассматриваться в последующих   
главах. Обязательно ознакомьтесь с разделом «Часть I, Введение» в приложе-  
нии с решениями (приложение B), где приводятся ответы, – упражнения и их   
решения иногда содержат дополнительные сведения, не рассматривавшиеся   
в основном тексте части, поэтому рекомендуется ознакомиться с ответами,   
даже если вам удастся ответить на вопросы самостоятельно.  
1. Взаимодействие. Используя системную командную строку, IDLE или дру-  
гой инструмент, запустите интерактивный сеанс интерпретатора Python   
(приглашение к вводу >>>) и введите выражение “Hello World!” (включая   
кавычки). Строка должна быть повторно выведена на экран. Цель этого   
упражнения состоит в том, чтобы помочь вам настроить окружение для   
запуска интерпретатора Python. В некоторых случаях вам может потре-  
ython. В некоторых случаях вам может потре-  
. В некоторых случаях вам может потре-  
боваться сначала выполнить команду cd, ввести полный путь к каталогу,   
куда был установлен выполняемый файл интерпретатора Python, или до-  
бавить путь к этому каталогу в переменную окружения PATH. При желании   
значение переменной PATH в системах UNIX можно установить в файле .cshrc   
или .kshrc� в �indows для этой цели можно использовать файл setup.bat,   
autoexec.bat или выполнить настройку переменной окружения с использо-  
ванием инструмента с графическим интерфейсом. Справку по настройкам   
переменных окружения см. в приложении А.  
2. Программы. В текстовом редакторе, который вы предпочитаете, создайте   
простой файл модуля, содержащий единственную инструкцию print(‘Hello   
module world!’), и сохраните его под именем module1.py. Теперь запустите   
этот файл каким-либо способом: из среды разработки IDLE, щелчком на яр-  
лыке, вызовом интерпретатора Python из командной строки, передав ему   
имя файла в виде аргумента (например, python module1.py), и так далее. По-  
пробуйте поэкспериментировать с разными способами запуска, которые об-

Закрепление пройденного   
117  
суждались в этой главе. Какие способы запуска показались вам проще? (На   
этот вопрос не может быть единственно правильного ответа.)  
3. Модули. Запустите интерактивный сеанс работы с интерпретатором Python   
(приглашение к вводу >>>) и импортируйте модуль, который был создан   
в упражнении 2. Попробуйте переместить файл в другой каталог и импор-  
тировать его снова из первоначального каталога (то есть запустите Python   
в каталоге, где производился импорт в первый раз). Что произошло? (Под-  
сказка: посмотрите, остался ли в первоначальном каталоге файл с байт-  
кодом module1.pyc?)  
4. Сценарии. Если ваша платформа поддерживает такую возможность, до-  
бавьте комбинацию символов #! в начало файла модуля module1.py, дайте   
файлу право на выполнение и попробуйте запустить его как обычный ис-  
полняемый файл. Что должна содержать первая строка? Обычно комбина-  
ция символов #! имеет особое значение только на платформе UNIX, Linux   
и других UNIX-подобных системах, таких как MAC OS X. Если вы работае-  
те в �indows, попробуйте просто запустить файл, введя его имя без пред-  
шествующего ему слова «python» (этот способ работает в последних версиях   
�indows) или с помощью диалога Пуск (Start) → Выполнить… (Run…).  
5. Ошибки  и  отладка. Поэкспериментируйте с математическими выраже-  
ниями и операциями присваивания в интерактивной командной оболоч-  
ке Python. Для начала введите выражения 2 \*\* 500 и 1/0. Что произошло?   
Потом попробуйте ввести имя переменной, которой еще не было присвоено   
значение. Что произошло на этот раз?  
Вы еще можете не знать этого, но вы столкнулись с исключениями (эту тему   
мы подробно будем рассматривать в седьмой части книги). Там вы узнае-  
те, что, с технической точки зрения, ваши действия привели к вызову того,   
что известно под названием обработчик исключений по умолчанию, – про-  
граммного кода, который выводит стандартные сообщения об ошибках.   
Если вы не выполняете перехват ошибок в своих программах, это за вас сде-  
лает обработчик по умолчанию, который выведет сообщение об ошибке.  
Исключения неразрывно связаны с понятием отладки в языке Python.   
Для начала вам вполне будет достаточно стандартного механизма обработ-  
ки ошибок – он позволит узнать причину ошибки, а также покажет, какие   
строки кода выполнялись в момент ее появления. О дополнительных воз-  
можностях отладки рассказывается во врезке «Отладка программ на языке   
Python» выше.  
6. Прерывание программы. В командной строке интерпретатора Python введи-  
те следующие инструкции:  
L = [1, 2] # Создать список с двумя элементами  
L.append(L) # Добавить в конец списка   
L  
Что произошло? Во всех современных версиях Python вы увидите кажу-  
щийся странным результат, который описывается в приложении с решени-  
ями, а также в следующей части книги. При использовании версий Python   
старее 1.5.1 остановить работу этого программного кода на большинстве   
платформ вам поможет комбинация клавиш Ctrl-C. Как вы думаете, в чем   
причина происходящего? Что вывел интерпретатор после нажатия комби-  
нации клавиш Ctrl-C?

118   
Глава 3. Как пользователь запускает программы   
Если вы используете версию Python более старую, чем 1.5.1,   
прежде чем выполнить это упражнение, обязательно проверьте,   
имеется ли возможность прерывать работу программ комбина-  
цией клавиш Ctrl-C, в противном случае вам придется ждать   
очень долго.  
7. Документация. Потратьте хотя бы 17 минут на исследование библиотеки   
Python и руководства по языку программирования, чтобы получить пред-  
ставление о стандартной библиотеке и о структуре комплекта докумен-  
тации. Вам нужно понять, по крайней мере, где в руководстве находятся   
описания основных тем. После этого вы легко сможете отыскать интере-  
сующую вас информацию. В системе �indows это руководство находится   
в разделе Python меню кнопки Пуск (Start), а также в виде пункта Python Docs   
(Документация Python) в меню Help (Справка) в среде разработки IDLE или в Ин-  
тернете по адресу: http://www.python.org/doc. Кроме того, хотелось бы также   
сказать несколько слов о других руководствах и источниках документации,   
описываемых (включая PyDoc и функцию help) в главе 15. Если у вас есть   
свободное время, займитесь исследованием веб-сайтов Python, а также веб-  
сайта расширений сторонних разработчиков PyPy. В частности, ознакомь-  
тесь со страницами документации и поиска на сайте Python.org – они могут   
оказаться для вас весьма значимыми ресурсами.

Часть II.  
Типы и операции

Глава 4.  
   
Введение в типы объектов языка Python  
Начиная с этой главы, мы приступаем к изучению языка Python. В самом   
общем виде можно сказать, что программы на языке Python выполняют не-  
которые действия над чем-то. «Некоторые действия» принимают форму опера-  
ций, таких как сложение или конкатенация, а под «чем-то» подразумеваются   
объекты, над которыми выполняются операции. В этой части книги мы сосре-  
доточимся на этом «что-то» и на действиях, которые могут выполняться про-  
граммой.  
Говоря более формальным языком, данные в языке Python представлены   
в форме объектов – либо встроенных, предоставляемых языком Python, либо   
объектов, которые мы создаем с применением конструкций языка Python или   
других инструментов, таких как библиотеки расширений, написанные на язы-  
ке C. Мы уточним это определение позднее, но если говорить по сути, объек-  
ты – это области памяти со значениями и ассоциированными с ними наборами   
операций.  
Объекты являются самым фундаментальным понятием в программировании   
на языке Python, поэтому эта глава начинается с обзора встроенных объект-  
ных типов языка Python.  
Однако для начала проясним, как эта глава вписывается в общую картину   
языка Python. С более определенной точки зрения программы на языке Python   
можно разложить на такие составляющие, как модули, инструкции, выраже-  
ния и объекты� при этом:  
1. Программы делятся на модули.  
2. Модули содержат инструкции.  
3. Инструкции состоят из выражений.  
4. Выражения создают и обрабатывают объекты.  
В главе 3 рассматривалась самая вершина этой иерархии – модули. Эта часть   
книги начинает рассмотрение с конца иерархии – с исследования встроенных   
объектов и выражений, в которых эти объекты могут участвовать.

122   
Глава 4. Введение в типы объектов языка Python   
Зачем нужны встроенные типы?  
Если вам приходилось использовать языки программирования более низкого   
уровня, такие как C или C++, то вы уже знаете, что значительная доля рабо-  
ты приходится на реализацию объектов, известных также как структуры   
данных, которые предназначены для представления составляющих предмет-  
ной области. В таких языках программирования необходимо заниматься про-  
работкой структур данных, управлять выделением памяти, реализовывать   
функции поиска и доступа к элементам структур и так далее. Это достаточно   
утомительно (и способствует появлению ошибок) и, как правило, отвлекает от   
достижения истинных целей.  
В типичных программах на языке Python в этом нет необходимости. Python   
предоставляет мощную коллекцию объектных типов, встроенных непосред-  
ственно в язык, поэтому обычно нет никакой необходимости создавать соб-  
ственные реализации объектов, предназначенных для решения поставленных   
задач. Фактически если у вас нет потребности в специальных видах обработки,   
которые не обеспечиваются встроенными типами объектов, вам всегда луч-  
ше использовать встроенные объекты вместо реализации своих собственных.   
И вот почему:  
 •  
Встроенные объекты упрощают создание программ. Для решения про-  
стых задач часто вполне достаточно встроенных типов для представления   
структур данных предметной области. В вашем распоряжении имеются та-  
кие мощные инструментальные средства, как коллекции (списки) и табли-  
цы поиска (словари), поэтому вы можете использовать их непосредственно.   
Благодаря встроенным объектным типам языка Python вы можете выпол-  
нить значительный объем работы.  
 •  
Встроенные объекты – это компоненты расширений. Для решения слож-  
ных задач вы можете создавать собственные объекты, используя для этого   
классы языка Python или интерфейсы языка C. Однако, как будет показано   
ниже, объекты, реализованные вручную, обычно основаны на таких встро-  
енных типах, как списки и словари. Например, структура данных типа стек   
может быть реализована как класс, основанный на использовании списков.  
 •  
Встроенные объекты часто более эффективны, чем созданные вручную   
структуры данных. Встроенные типы языка Python используют уже опти-  
мизированные структуры данных, реализованные на языке C для дости-  
жения высокой производительности. Вы можете сами создавать подобные   
типы объектов, но вам придется приложить немало усилий, чтобы достичь   
скорости, которая обеспечивается встроенными типами объектов.  
 •  
Встроенные объекты – это стандартная часть языка. В определенной сте-  
пени Python многое заимствует как из языков, полагающихся на исполь-  
зование встроенных инструментальных средств (таких как LISP), так и по-  
лагающихся на мастерство программиста, который должен выполнить соб-  
ственную реализацию инструментов и структур данных (таких как C++).   
В языке Python вы можете создавать собственные типы объектов, но в са-  
мом начале делать это не рекомендуется. Более того, из-за того, что встроен-  
ные компоненты являются стандартными составляющими языка Python,   
они всегда остаются неизменными, тогда как собственные структуры име-  
ют свойство изменяться от случая к случаю.

Зачем нужны встроенные типы?   
123  
Другими словами, встроенные типы объектов не только упрощают процесс   
программирования, но они обладают большей эффективностью и производи-  
тельностью, чем большинство типов, созданных вручную. Даже если вы соз-  
даете собственные типы объектов, встроенные объекты будут ядром любой   
программы на Python.  
Базовые типы данных в языке Python  
В табл. 4.1 представлены некоторые встроенные типы объектов языка Python   
и некоторые синтаксические конструкции использования этих объектов в виде   
литералов – то есть выражения, которые генерируют эти объекты.1 Некоторые   
из этих типов наверняка покажутся вам знакомыми, если ранее вам приходи-  
лось работать с другими языками программирования. Например, числа и стро-  
ки представляют числовые и текстовые значения соответственно, а файлы обе-  
спечивают интерфейс для работы с файлами, хранящимися в компьютере.  
Таблица 4.1. Некоторые встроенные объекты  
Тип объекта  
Пример литерала/создания  
Числа  
1234, 3.1415, 3+4j, Decimal, Fraction  
Строки  
‘spam’, “guido’s” , b’a\x01c’  
Списки  
[1, [2, ‘three’], 4]  
Словари  
{‘food’: ‘spam’, ‘taste’: ‘yum’}  
Кортежи  
(1,’spam’, 4, ‘U’)  
Файлы  
myfile = open(‘eggs’, ‘r’)  
Множества  
set(‘abc’), {‘a’, ‘b’, ‘c’}  
Прочие базовые типы  
Сами типы, None, логические значения  
Типы структурных элементов   
программ  
Функции, модули, классы (часть IV,   
часть V, часть VI)  
Типы, имеющие отношение   
к реализации  
Компилированный программный код,   
стек вызовов (часть IV, часть VII)  
Таблица 4.1 содержит далеко не полный список, потому что объектами явля-  
ются все данные, которые приходится обрабатывать в программах на языке   
Python. Например, когда на языке Python реализуется поиск текста по шабло-  
ну, – создаются объекты шаблонов, когда программируются сетевые взаимо-  
действия, – используются объекты сокетов. Существуют и другие типы объек-  
тов, которые создаются в результате импорта и использования модулей, и все   
они обладают своим собственным поведением.  
1   
В этой книге под термином литерал подразумевается выражение, создающее объ-  
ект, которое иногда также называется константой. Следует иметь в виду, что тер-  
мин «константа» не означает объекты и переменные, которые никогда не изменяют-  
ся (то есть этот термин никак не связан с директивой const языка C++ или с понятием   
«неизменяемый» («immutable») в языке Python, – эта тема будет рассматриваться   
ниже, в разделе «Неизменяемость»).

124   
Глава 4. Введение в типы объектов языка Python   
В следующих частях книги вы узнаете, что структурные элементы программ,   
такие как функции, модули и классы, также являются объектами в язы-  
ке Python – они создаются с помощью инструкций и выражений, таких как   
def, class, import и lambda, и могут свободно передаваться между различными   
частями сценариев, сохраняться в других объектах и так далее. Кроме того,   
в языке Python имеется множество типов, имеющих отношение к реализации,   
таких как объекты с компилированным программным кодом, – они представ-  
ляют интерес скорее для разработчиков инструментов, чем для прикладных   
программистов. Все эти типы также будут рассматриваться в последующих   
частях книги.  
Типы объектов, перечисленные в табл. 4.1, обычно называют базовыми, по-  
тому что они встроены непосредственно в язык Python, то есть для создания   
большинства из них используется вполне определенный синтаксис. Например,   
когда выполняется следующий программный код:  
>>> ‘spam’  
то, говоря техническим языком, выполняется выражение-литерал, которое ге-  
нерирует и возвращает новый строковый объект. Такова специфика синтакси-  
са Python создания этого объекта. Похожим образом выражение, заключенное   
в квадратные скобки, создает список, заключенное в фигурные скобки – сло-  
варь и так далее. Хотя, как вы сможете убедиться, в языке Python отсутствует   
конструкция объявления типа, сам синтаксис выполняемых выражений зада-  
ет типы создаваемых и используемых объектов. Фактически выражения, соз-  
дающие объекты, подобные тем, что представлены в табл. 4.1, в языке Python   
являются источниками типов.  
Не менее важно отметить, что как только будет создан объект, он будет ассо-  
циирован со своим собственным набором операций на протяжении всего вре-  
мени существования – над строками можно будет выполнять только строковые   
операции, над списками – только операции, применимые к спискам. Как вы   
узнаете далее, в языке Python используется динамическая типизация (типы   
данных определяются автоматически и их не требуется объявлять в программ-  
ном коде), но при этом он является языком со строгой типизацией (вы сможете   
выполнять над объектом только те операции, которые применимы к его типу).  
Функционально типы объектов, представленные в табл. 4.1, являются более   
универсальными и более эффективными, чем может показаться. Например,   
вы узнаете, что списки и словари являются достаточно мощными средствами   
представления данных, обеспечивающими поддержку коллекций и функций   
поиска, которые в низкоуровневых языках программирования приходится ре-  
ализовывать вручную. Говоря коротко, списки обеспечивают поддержку упо-  
рядоченных коллекций других объектов, а словари реализуют возможность   
хранения объектов по ключам. И списки, и словари могут быть вложенными,   
по мере необходимости увеличиваться и уменьшаться в размерах и содержать   
объекты любых типов.  
В последующих главах мы подробно изучим каждый из типов объектов, пере-  
численных в табл. 4.1. Но прежде чем углубляться в детали, давайте позна-  
комимся с базовыми типами объектов Python в действии. Оставшаяся часть   
главы представляет собой обзор операций, которые мы более подробно будем   
исследовать в последующих главах. Не следует надеяться, что в этой главе   
будут даны исчерпывающие объяснения, поскольку основная ее цель состоит

Числа   
125  
в том, чтобы разжечь ваш аппетит и представить некоторые базовые идеи. Од-  
нако лучший способ что-то начать – это просто начать, поэтому перейдем сразу   
к программному коду.  
Числа  
Если в прошлом вам приходилось заниматься программированием, некоторые   
типы объектов из табл. 4.1 скорее всего покажутся вам знакомыми. Но даже   
если это не так, числа являются чрезвычайно простым понятием. Базовый на-  
бор объектов языка Python включает в себя вполне ожидаемые типы: целые   
числа (числа без дробной части), вещественные числа (грубо говоря, числа с де-  
сятичной точкой) и более экзотические типы (комплексные числа с мнимой   
частью, числа с фиксированной точностью, рациональные числа, представлен-  
ные парой целых чисел, – числитель и знаменатель дроби, и множества).  
Несмотря на наличие некоторых необычных типов, базовые числовые типы   
в языке Python действительно являются базовыми. Числа в Python поддер-  
живают набор самых обычных математических операций. Например, символ   
«плюс» (+) означает сложение, символ «звездочка» (\*) – умножение, а два сим-  
вола «звездочка» (\*\*) – возведение в степень:  
>>> 123 + 222 # Целочисленное сложение  
345  
>>> 1.5 \* 4 # Умножение вещественных чисел  
6.0  
>>> 2 \*\* 100 # 2 в степени 100  
1267650600228229401496703205376  
Обратите внимание на результат последней операции: в Python 3.0 целые чис-  
Python 3.0 целые чис-  
 3.0 целые чис-  
ла автоматически обеспечивают неограниченную точность для представления   
больших значений (в Python 2.6 для представления больших целых чисел име-  
ется отдельный тип длинных целых чисел). Например, вы можете попробовать   
вычислить 2 в степени 1 000 000 (но едва ли стоит это делать, так как на экран   
будет выведено число длиной более 300 000 знаков, что может занять продол-  
жительное время!).   
>>> len(str(2 \*\* 1000000)) # Сколько цифр в действительно БОЛЬШОМ числе?  
301030  
Начав экспериментировать с вещественными числами, вы наверняка обратите   
внимание на то, что на первый взгляд может показаться странным:  
>>> 3.1415 \* 2 # repr: как программный код  
6.2830000000000004  
>>> print(3.1415 \* 2) # str: более дружественный формат  
6.283  
Первый результат – это не ошибка, проблема здесь связана с отображением.   
Оказывается, вывести содержимое любого объекта можно двумя способами:   
с полной точностью (как в первом результате), и в форме, более удобной для вос-  
приятия человеком (как во втором результате). Формально первая форма назы-  
вается repr (объект в виде программного кода), а вторая, более дружественная   
к пользователю, – str. Различия между ними станут более понятны, когда мы   
приступим к изучению классов, а пока, если что-то выглядит непонятным, по-  
пробуйте вывести тот же результат с помощью инструкции print.

126   
Глава 4. Введение в типы объектов языка Python   
Помимо выражений для выполнения операций с числами в составе Python есть   
несколько полезных модулей:  
>>> import math  
>>> math.pi  
3.1415926535897931  
>>> math.sqrt(85)  
9.2195444572928871  
Модуль math содержит более сложные математические функции, а модуль random   
реализует генератор случайных чисел и функцию случайного выбора (в данном   
случае из списка, о котором будет рассказываться ниже, в этой же главе):  
>>> import random  
>>> random.random()  
0.59268735266273953  
>>> random.choice([1, 2, 3, 4])  
1  
Кроме того, Python включает в себя более экзотические числовые объекты, та-  
кие как комплексные числа, числа с фиксированной десятичной точкой и ра-  
циональные числа, множества и логические значения, а среди свободно рас-  
пространяемых расширений можно найти и другие числовые типы (например,   
матрицы и векторы). Обсуждение этих типов будет приводиться далее в этой   
книге.  
Пока что мы использовали Python как простой калькулятор, но чтобы иметь   
большую возможность судить о встроенных типах, перейдем к строкам.  
Строки  
Строки используются для записи текстовой информации, а также произволь-  
ных последовательностей байтов. Это наш первый пример последовательно-  
стей, или упорядоченных коллекций других объектов, в языке Python. По-  
следовательности поддерживают порядок размещения элементов, которые они   
содержат, слева направо: элементы сохраняются и извлекаются исходя из их   
позиций в последовательностях. Строго говоря, строки являются последова-  
тельностями односимвольных строк. Другими типами последовательностей   
являются списки и кортежи (будут описаны ниже).  
Операции над последовательностями  
Будучи последовательностями, строки поддерживают операции, предполага-  
ющие определенный порядок позиционирования элементов. Например, если   
имеется четырехсимвольная строка, то с помощью встроенной функции len   
можно определить ее длину, а отдельные элементы строки извлечь с помощью   
выражений индексирования:  
>>> S = ‘Spam’  
>>> len(S) # Длина  
4  
>>> S[0] # Первый элемент в S, счет начинается с позиции 0  
‘S’  
>>> S[1] # Второй элемент слева  
‘p’

Строки   
127  
В языке Python индексы реализованы в виде смещений от начала и потому   
индексация начинается с 0: первый элемент имеет индекс 0, второй – 1 и так   
далее.   
Обратите внимание, как в этом примере выполняется присваивание строки   
переменной с именем S. Подробнее сам процесс присваивания мы будем рас-  
сматривать позднее (в частности, в главе 6), а пока хочу отметить, что в язы-  
ке Python не требуется объявлять переменные заранее. Переменная создается   
в тот момент, когда ей присваивается значение, при этом переменной можно   
присвоить значение любого типа, а при использовании внутри выражения имя   
переменой замещается ее фактическим значением. Кроме того, прежде чем   
появится возможность обратиться к переменной, ей должно быть присвоено   
какое-либо значение. Но пока вам достаточно помнить – чтобы сохранить объ-  
ект для последующего использования, его нужно присвоить переменной.  
В языке Python предусмотрена возможность индексации в обратном порядке,   
от конца к началу – положительные индексы откладываются от левого конца   
последовательности, а отрицательные – от правого:  
>>> S[-1] # Последний элемент в конце S  
‘m’  
>>> S[-2] # Второй элемент с конца  
‘a’  
Формально отрицательные индексы просто складываются с длиной строки,   
поэтому следующие две операции эквивалентны (хотя первая форма записи   
выглядит проще и понятнее):  
>>> S[-1] # Последний элемент в S  
‘m’  
>>> S[len(S)-1] # Отрицательная индексация, более сложный способ  
‘m’  
Примечательно, что внутри квадратных скобок допускается использовать не   
только жестко заданные числовые литералы, но и любые другие выражения –   
везде, где Python ожидает получить значение, можно использовать литералы,   
переменные или любые выражения. Весь синтаксис языка Python следует это-  
му общему принципу.  
В дополнение к простой возможности индексирования по номеру позиции, по-  
следовательности поддерживают более общую форму индексирования, извест-  
ную как получение среза (slicing), которая обеспечивает возможность извлече-  
ния за одну операцию целого сегмента (среза). Например:  
>>> S # Строка из 4 символов  
‘Spam’  
>>> S[1:3] # Срез строки S начиная со смещения 1 и до 2 (не 3)  
‘pa’  
Проще всего можно представить себе срез как способ извлечения целого столб-  
ца из строки за один шаг. В общем виде синтаксис операции получения среза   
выглядит как: X[I:J], и означает: «извлечь из X все, начиная со смещения I и до   
смещения J, но не включая его». В качестве результата возвращается новый   
объект. Например, последняя операция из примера выше вернет все символы   
строки S со смещениями с 1 по 2 (то есть 3 – 1 символов) в виде новой строки.   
В результате получается срез, или «выборка» двух символов из середины.

128   
Глава 4. Введение в типы объектов языка Python   
При выполнении операции получения среза левая граница по умолчанию при-  
нимается равной нулю, а правая – длине последовательности, к которой при-  
меняется операция. В результате мы получаем следующие наиболее распро-  
страненные варианты использования:  
>>> S[1:] # Все, кроме первого элемента (1:len(S))  
‘pam’  
>>> S # Сама строка S без изменений  
‘Spam’  
>>> S[0:3] # Все, кроме последнего элемента  
‘Spa’  
>>> S[:3] # То же, что и S[0:3]  
‘Spa’  
>>> S[:-1] # Еще раз все, кроме последнего элемента, но проще (0:-1)  
‘Spa’  
>>> S[:] # Все содержимое S, как обычная копия (0:len(S))  
‘Spam’  
Обратите внимание, что в качестве границ срезов можно использовать отрица-  
тельные индексы и что последняя операция фактически создает копию всей   
строки. Как мы узнаем позднее, нет смысла копировать строки таким спосо-  
бом, но такая форма копирования очень удобна при работе с другими последо-  
вательностями, такими как списки.  
Наконец, будучи последовательностями, строки поддерживают операцию кон-  
катенации, которая записывается в виде знака плюс (объединение двух строк   
в одну строку), и операцию повторения (новая строка создается за счет много-  
кратного повторения другой строки):  
>>> S  
‘Spam’  
>>> S + ‘xyz’ # Конкатенация  
‘Spamxyz’  
>>> S # S остается без изменений  
‘Spam’  
>>> S \* 8 # Повторение  
‘SpamSpamSpamSpamSpamSpamSpamSpam’  
Обратите внимание, что знак плюс (+) имеет различное значение для разных   
объектов: для чисел – сложение, а для строк – конкатенация. Это универсаль-  
ное свойство языка Python, которое далее в книге будет называться полимор-  
физмом, означает, что фактически выполняемая операция зависит от объек-  
тов, которые принимают в ней участие. Как будет показано, когда мы присту-  
пим к изучению динамической типизации, такой полиморфизм в значитель-  
ной степени обеспечивает выразительность и гибкость программного кода на   
языке Python. Поскольку отсутствуют ограничения, связанные с типами, опе-  
рации в языке Python обычно в состоянии автоматически обрабатывать объек-  
ты самых разных типов, при условии, что они поддерживают совместимый ин-  
терфейс (как в данном случае операция +). В языке Python идея полиморфизма   
является ключевой концепцией, которую мы будем рассматривать далее в этой   
книге.  
Неизменяемость  
Обратите внимание: в предыдущих примерах ни одна из использованных опе-  
раций не изменяла оригинальную строку. Все операции над строками в ре-

Строки   
129  
зультате создают новую строку, потому что строки в языке Python являются   
неизменяемыми – после того, как строка будет создана, ее нельзя изменить.   
Например, вы не сможете изменить строку присвоением значения одной из ее   
позиций, но вы всегда можете создать новую строку и присвоить ей то же самое   
имя. Поскольку Python очищает память, занятую ненужными больше объек-  
тами (как будет показано позднее), такой подход не так уж неэффективен, как   
могло бы показаться на первый взгляд:  
>>> S  
‘Spam’  
>>> S[0] = ‘z’ # Неизменяемые объекты нельзя изменить  
...текст сообщения об ошибке опущен...  
TypeError: ‘str’ object does not support item assignment  
   
>>> S = ‘z’ + S[1:] # Но с помощью выражений мы можем создавать новые объекты   
>>> S  
‘zpam’  
Все объекты в языке Python либо относятся к классу неизменяемых, либо нет.   
Если говорить о базовых типах, то числа, строки и кортежи являются неизме-  
няемыми, а списки и словари – нет (они легко могут изменяться в любой своей   
части). Помимо всего неизменяемость может рассматриваться как гарантия,   
что некоторый объект будет оставаться постоянным на протяжении работы   
программы.  
Методы, специфичные для типа  
Все строковые операции, которые мы до сих пор рассматривали, в действи-  
тельности являются операциями над последовательностями, то есть эти опера-  
ции могут использоваться для работы с любыми последовательностями языка   
Python, включая списки и кортежи. Однако помимо операций, универсальных   
для последовательностей, строки также имеют свои собственные операции, ре-  
ализованные в виде методов (функций, присоединенных к объекту, которые   
запускаются выражением вызова).  
Например, метод строк find выполняет поиск подстроки в строке (он возвраща-  
ет смещение переданной ему подстроки или -1, если поиск не увенчался успе-  
хом), а метод replace производит глобальный поиск с заменой:  
>>> S.find(‘pa’) # Поиск смещения подстроки  
1  
>>> S  
‘Spam’  
>>> S.replace(‘pa’, ‘XYZ’) # Замена одной подстроки другой  
‘SXYZm’  
>>> S  
‘Spam’  
И снова независимо от имен этих строковых методов, применяя методы, мы   
не изменяем оригинальную строку, а создаем новую, т. к. строки являются не-  
изменяемыми, и это следует учитывать. Строковые методы – это первый уро-  
вень в комплекте инструментальных средств обработки текста языка Python.   
Другие методы позволяют разбивать строки на подстроки по определенному   
символу-разделителю (достаточно удобно для простых случаев разбора строк),   
преобразовывать регистр символов, проверять тип содержимого строк (цифры,

130   
Глава 4. Введение в типы объектов языка Python   
алфавитные символы и так далее) и отсекать пробельные символы с обоих кон-  
цов строк.  
>>> line = ‘aaa,bbb,ccccc,dd’  
>>> line.split(‘,’) # Разбивает строку по разделителю и создает список строк   
[‘aaa’, ‘bbb’, ‘ccccc’, ‘dd’]  
>>> S = ‘spam’  
>>> S.upper() # Преобразование символов в верхний и в нижний регистр  
‘SPAM’  
   
>>> S.isalpha() # Проверка содержимого: isalpha, isdigit и так далее  
True  
   
>>> line = ‘aaa,bbb,ccccc,dd\n’  
>>> line = line.rstrip() # Удаляет завершающие пробельные символы  
>>> line  
‘aaa,bbb,ccccc,dd’  
Кроме того, строки поддерживают операции подстановки, известные как фор-  
матирование и доступные как в виде выражений (существовали изначально),   
так и в виде методов строк (появились в версиях 2.6 и 3.0):  
>>> ‘%s, eggs, and %s’ % (‘spam’, ‘SPAM!’) # Выражение (во всех версиях)  
‘spam, eggs, and SPAM!’  
>>> ‘{0}, eggs, and {1}’.format(‘spam’, ‘SPAM!’) # Метод (2.6, 3.0)  
‘spam, eggs, and SPAM!’  
Следует заметить, что в отличие от универсальных операций, применяемых   
к последовательностям, строковые методы могут применяться только к стро-  
кам и ни к каким другим объектам, хотя некоторые типы могут иметь мето-  
ды с похожими именами. Следует понимать, что инструментальные средства   
языка Python делятся на несколько уровней: универсальные операции, кото-  
Python делятся на несколько уровней: универсальные операции, кото-  
 делятся на несколько уровней: универсальные операции, кото-  
рые могут применяться к нескольким типам, реализованы в виде встроенных   
функций и выражений (например, len(X), X[0]), а операции, специфичные для   
определенного типа, реализованы в виде методов (например, aString.upper()).   
Выбор требуемых инструментов из всех этих категорий станет более простым   
по мере изучения языка Python, а в следующем разделе приводится несколько   
рекомендаций, которые вы сможете использовать уже сейчас.  
Получение помощи  
Методы, представленные в предыдущем разделе, являются лишь небольшой   
частью того, что доступно при работе со строковыми объектами. Вообще, эта   
книга не является исчерпывающим источником информации о методах объ-  
ектов. Чтобы получить дополнительную информацию, вы всегда можете вос-  
пользоваться функцией dir, которая возвращает список всех доступных атри-  
бутов заданного объекта. Предположим, что переменная S по-прежнему оста-  
ется строкой� ниже приводится список ее атрибутов в Python 3.0 (в Python 2.6   
этот список немного отличается):   
>>> dir(S)  
[‘\_\_add\_\_’, ‘\_\_class\_\_’, ‘\_\_contains\_\_’, ‘\_\_delattr\_\_’, ‘\_\_doc\_\_’, ‘\_\_eq\_\_’,  
‘\_\_format\_\_’, ‘\_\_ge\_\_’, ‘\_\_getattribute\_\_’, ‘\_\_getitem\_\_’, ‘\_\_getnewargs\_\_’,  
‘\_\_gt\_\_’, ‘\_\_hash\_\_’, ‘\_\_init\_\_’, ‘\_\_iter\_\_’, ‘\_\_le\_\_’, ‘\_\_len\_\_’, ‘\_\_lt\_\_’,  
‘\_\_mod\_\_’, ‘\_\_mul\_\_’, ‘\_\_ne\_\_’, ‘\_\_new\_\_’, ‘\_\_reduce\_\_’, ‘\_\_reduce\_ex\_\_’,

Строки   
131  
‘\_\_repr\_\_’, ‘\_\_rmod\_\_’, ‘\_\_rmul\_\_’, ‘\_\_setattr\_\_’, ‘\_\_sizeof\_\_’, ‘\_\_str\_\_’,  
‘\_\_subclasshook\_\_’, ‘\_formatter\_field\_name\_split’, ‘\_formatter\_parser’,  
‘capitalize’, ‘center’, ‘count’, ‘encode’, ‘endswith’, ‘expandtabs’, ‘find’,  
‘format’, ‘index’, ‘isalnum’,’isalpha’, ‘isdecimal’, ‘isdigit’, ‘isidentifier’,   
‘islower’, ‘isnumeric’, ‘isprintable’, ‘isspace’, ‘istitle’, ‘isupper’, ‘join’,   
‘ljust’, ‘lower’, ‘lstrip’, ‘maketrans’, ‘partition’, ‘replace’, ‘rfind’, ‘rindex’,   
‘rjust’, ‘rpartition’, ‘rsplit’, ‘rstrip’, ‘split’, ‘splitlines’, ‘startswith’,   
‘strip’, ‘swapcase’, ‘title’, ‘translate’, ‘upper’, ‘zfill’]  
Скорее всего, вам не понадобятся имена из этого списка, содержащие символы   
подчеркивания, до того момента, пока мы не приступим к изучению возмож-  
ности перегрузки операторов в классах, – они представляют собой реализа-  
цию строкового объекта и доступны для поддержки специализации. В общем   
случае ведущие и завершающие символы подчеркивания используются ин-  
терпретатором Python для обозначения особенностей внутренней реализации.   
Имена без символов подчеркивания в этом списке обозначают методы строко-  
вых объектов.  
Функция dir возвращает лишь имена методов. Чтобы узнать назначение того   
или иного метода, можно передать его имя функции help:  
>>> help(S.replace)  
Help on built-in function replace:  
(Справка о встроенной функции replace:)  
   
replace(...)  
 S.replace (old, new[, count]) -> str  
   
 Return a copy of S with all occurrences of substring  
 old replaced by new. If the optional argument count is  
 given, only the first count occurrences are replaced.  
 (Возвращает копию S, где все вхождения подстроки old замещены  
 подстрокой new. Если указан необязательный аргумент count,  
 замещаются только первые count вхождений.)  
Функция help – один из немногих интерфейсов к системе программного кода   
Python, поставляемого в составе Python в составе инструмента под названием   
PyDoc, который позволяет извлекать описание из объектов. Далее в этой книге   
вы узнаете, что PyDoc позволяет отображать информацию в формате HTML.  
Можно также запросить информацию и для самого строкового объекта (на-  
пример, help(S)), но в этом случае вы можете получить больше информации,   
чем хотелось бы, – описание всех строковых методов. Часто бывает удобнее   
запрашивать информацию о конкретном методе, как это было продемонстри-  
ровано выше.  
За дополнительной информацией всегда можно обратиться к справочному ру-  
ководству по стандартной библиотеке или к печатным справочным изданиям,   
но функции dir и help в языке Python представляют собой самое первое сред-  
ство получения доступа к документации.  
Другие способы представления строк  
К этому моменту мы познакомились с операциями над последовательностями   
и методами, специфичными для строк. Однако кроме этого язык программи-  
рования Python предоставляет несколько различных способов представления

132   
Глава 4. Введение в типы объектов языка Python   
строк в программном коде, которые мы будем исследовать позднее (включая   
служебные символы, которые представлены, например, в виде последователь-  
ностей, начинающихся с символа обратного слеша:)  
>>> S = ‘A\nB\tC’ # \n – это символ “конец строки”, \t – символ табуляции  
>>> len(S) # Каждая из этих пар соответствует единственному символу  
5  
   
>>> ord(‘\n’) # В ASCII \n – это байт с числовым значением 10   
10  
   
>>> S = ‘A\0B\0C’ # \0 – это двоичный ноль, не является завершителем строки  
>>> len(S)  
5  
Язык Python допускает заключать строки в кавычки или в апострофы (они   
означают одно и то же). Кроме того, имеется специальная форма определения   
многострочных строковых литералов – тройные кавычки или апострофы. Ког-  
да используется такая форма, все строки в программном коде объединяются   
в одну строку, а там, где в исходном тексте выполняется переход на новую стро-  
ку, вставляется символ «конец строки». Это незначительное синтаксическое   
удобство весьма полезно для оформления в сценариях на языке Python круп-  
ных блоков текста, таких как разметка HTML или XML:  
>>> msg = “”” aaaaaaaaaaaaa  
bbb’’’bbbbbbbbbb””bbbbbbb’bbbb  
cccccccccccccc”””  
>>> msg  
‘\naaaaaaaaaaaaa\nbbb\\u2019\\u2019\\u2019bbbbbbbbbb””bbbbbbb\\u2019bbbb\ncccccccccccccc’  
Кроме того, Python предоставляет поддержку литералов «неформатирован-  
ных», «сырых» строк, в которых символ обратного слеша интерпретируется   
как обычный символ (они начинаются с символа r), а также поддержку строк   
с символами Юникода, обеспечивающих интернационализацию. В версии 3.0   
базовый тип строк str также может содержать символы Юникода (при этом   
предполагается, что символы ASCII являются разновидностью символов Юни-  
ASCII являются разновидностью символов Юни-  
 являются разновидностью символов Юни-  
кода), а тип bytes может представлять строки двоичных байтов. В версии 2.6   
строки Юникода были представлены отдельным типом, а строки типа str могли   
содержать 8-битные символы и двоичные данные. Файлы также изменились   
в версии 3.0, и теперь они возвращают и принимают объекты типа str только   
при работе в текстовом режиме, а при работе в двоичном режиме – только объ-  
екты типа bytes. В последующих главах мы еще встретимся с этими специаль-  
ными формами строк.  
Поиск по шаблону  
Прежде чем двинуться дальше, хочется заметить, что ни один из строковых   
объектов не поддерживает возможность обработки текста на основе шаблонов.   
Рассмотрение инструментов, выполняющих поиск текста по шаблону, выхо-  
дит за рамки этой книги, но для читателей, знакомых с другими языками сце-  
нариев, будет интересно узнать, как выполняется поиск по шаблону в языке   
Python – для этого необходимо импортировать модуль с именем re. Этот модуль   
содержит аналогичные функции для выполнения поиска, разбиения и заме-  
ны, но за счет использования шаблонов мы можем использовать более общие   
варианты решения задач:

Списки   
133  
>>> import re  
>>> match = re.match(‘Hello[ \t]\*(.\*)world’, ‘Hello Python world’)  
>>> match.group(1)  
‘Python ‘  
В этом примере выполняется поиск строки, начинающейся со слова «Hello»,   
вслед за которым следуют ноль или более символов табуляции или пробелов,   
за которыми могут следовать произвольные символы, которые будут сохра-  
нены, как группа совпадения, и завершающаяся словом «world». Если такая   
подстрока будет найдена, части ее, соответствующие шаблону, заключенному   
в круглые скобки, будут доступны в виде групп. Например, следующий ша-  
блон извлекает три группы, разделенные символами слеша:  
>>> match = re.match(‘/(.\*)/(.\*)/(.\*)’, ‘/usr/home/lumberjack’)  
>>> match.groups()  
(‘usr’, ‘home’, ‘lumberjack’)  
Поиск по шаблону реализован в виде чрезвычайно сложного механизма обра-  
ботки текста, но в языке Python имеется поддержка еще более сложных меха-  
низмов, включая возможность обработки естественного языка человеческого   
общения. Впрочем, для этого руководства я и так сказал уже достаточно об об-  
работке строк, поэтому теперь мы перейдем к рассмотрению другого типа.  
Списки  
Списки – это самое общее представление последовательностей, реализованных   
в языке Python. Списки – это упорядоченные по местоположению коллекции   
объектов произвольных типов, размер которых не ограничен. Кроме того, в от-  
личие от строк, списки являются изменяемыми – они могут модифицировать-  
ся как с помощью операций присваивания по смещениям, так и с помощью   
разнообразных методов работы со списками.  
Операции над последовательностями  
Поскольку списки являются последовательностями, они поддерживают все   
операции над последовательностями, которые обсуждались в разделе, посвя-  
щенном строкам. Единственное отличие состоит в том, что результатом таких   
операций являются списки, а не строки. Например, для списка, состоящего из   
трех элементов:  
>>> L = [123, ‘spam’, 1.23] # Список из трех объектов разных типов  
>>> len(L) # Число элементов в списке  
3  
Мы можем обращаться к элементам списка по их индексам, получать срезы   
и так далее, точно так же, как и в случае со строками:  
>>> L[0] # Доступ к элементу списка по его индексу  
123  
   
>>> L[:-1] # Операция получения среза возвращает новый список  
[123, ‘spam’]  
   
>>> L + [4, 5, 6] # Операция конкатенации также возвращает новый список  
[123, ‘spam’, 1.23, 4, 5, 6]

134   
Глава 4. Введение в типы объектов языка Python   
>>> L # Наши действия не привели к изменению оригинального списка  
[123, ‘spam’, 1.23]  
Методы, специфичные для типа  
Списки в языке Python являются аналогом массивов в других языках про-  
граммирования, но они обладают более широкими возможностями. С одной   
стороны, они не ограничены одним типом элементов, например, только что   
рассмотренный список содержит три элемента совершенно разных типов (целое   
число, строку и вещественное число). Кроме того, размер списков не ограничен,   
благодаря чему они могут увеличиваться и уменьшаться по мере необходимо-  
сти в результате выполнения операций, характерных для списков:  
>>> L.append(‘NI’) # Увеличение: в конец списка добавляется новый объект  
>>> L  
[123, ‘spam’, 1.23, ‘NI’]  
   
>>> L.pop(2) # Уменьшение: удаляется элемент из середины списка  
1.23  
   
>>> L # Инструкция “del L[2]” также удалит элемент списка  
[123, ‘spam’, ‘NI’]  
В данном примере метод append увеличивает размер списка и вставляет в конец   
новый элемент. Метод pop (или эквивалентная ему инструкция del) удаляет из   
списка элемент с заданным смещением, что приводит к уменьшению списка.   
Другие методы списков позволяют вставлять новые элементы в произвольное   
место списка (insert), удалять элемент, заданный значением (remove), и так да-  
лее. Так как списки являются изменяемыми, большинство методов списков не   
создают новый список, а изменяют оригинальный список:  
>>> M = [‘bb’, ‘aa’, ‘cc’]  
>>> M.sort()  
>>> M  
[‘aa’, ‘bb’, ‘cc’]  
   
>>> M.reverse()  
>>> M  
[‘cc’, ‘bb’, ‘aa’]  
Метод sort, использованный в этом примере, по умолчанию упорядочивает эле-  
менты списка по возрастанию, а метод reverse – по убыванию. В обоих случаях   
происходит непосредственное изменение самого списка.  
Проверка выхода за границы  
Хотя списки не имеют фиксированного размера, язык Python, тем не менее, не   
допускает возможности обращаться к несуществующим элементам списка. Об-  
ращение к элементам списка по индексам, значения которых выходят за преде-  
лы списка, всегда является ошибкой:  
>>> L  
[123, ‘spam’, ‘NI’]  
   
>>> L[99]  
...текст сообщения об ошибке опущен...  
IndexError: list index out of range

Списки   
135  
>>> L[99] = 1  
... текст сообщения об ошибке опущен...  
IndexError: list assignment index out of range  
В этом примере я специально допустил ошибку (особенно неприятную в языке   
C, который не выполняет проверку выхода за границы массива, как Python)   
и попытался выполнить присваивание за пределами списка. Вместо того что-  
бы увеличить размер списка, интерпретатор Python сообщил об ошибке. Чтобы   
увеличить список, необходимо воспользоваться таким методом, как append.  
Вложенные списки  
Одна из замечательных особенностей базовых типов языка Python состоит   
в том, что они поддерживают возможность создания вложенных конструкций   
произвольной глубины и в любых комбинациях (например, можно создать   
список, содержащий словарь, который содержит другой список, и так далее).   
Одно из очевидных применений этой особенности – представление матриц, или   
«многомерных массивов» в языке Python. Делается это с помощью списка, со-  
держащего вложенные списки:  
>>> M = [[1, 2, 3], # Матрица 3 x 3 в виде вложенных списков  
 [4, 5, 6], # Выражение в квадратных скобках может   
 [7, 8, 9]] # занимать несколько строк  
   
>>> M  
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
Здесь мы реализовали список, состоящий из трех других списков. В результа-  
те была получена матрица чисел 3 x 3. Обращаться к такой структуре можно   
разными способами:  
>>> M[1] # Получить строку 2  
[4, 5, 6]  
   
>>> M[1][2] # Получить строку 2, а затем элемент 3 в этой строке  
6  
Первая операция в этом примере возвращает вторую строку целиком, а вто-  
рая – третий элемент в этой строке. Соединение операций индексирования   
позволяет все дальше и дальше погружаться вглубь вложенной структуры   
объектов.1  
Генераторы списков  
Помимо обычных операций над последовательностями и методов списков   
Python предоставляет возможность выполнять более сложные операции над   
1   
Такая организация матриц вполне пригодна для решения небольших задач, но для   
реализации более сложных программ числовой обработки информации желательно   
использовать специализированные расширения, например NumPy. Такого рода ин-  
струменты позволяют хранить и обрабатывать матрицы намного эффективнее, чем   
такая структура, реализованная в виде вложенных списков. Как уже говорилось,   
расширение NumPy превращает Python в свободный и более мощный эквивалент   
системы MatLab, и такие организации, как NASA, Los Alamos и JPMorgan Chase,   
используют его для решения научных и финансовых задач. Дополнительную инфор-  
мацию об этом расширении вы без труда найдете в Сети.

136   
Глава 4. Введение в типы объектов языка Python   
списками, известные как выражения генераторов списков (list comprehension   
expression), которые представляют эффективный способ обработки таких   
структур, как приведенная в примере матрица. Предположим, например, что   
нам требуется извлечь из нашей матрицы второй столбец. Строку легко можно   
получить, выполнив операцию индексирования, потому что матрица хранит-  
ся в виде строк, однако, благодаря генераторам списков, получить столбец ни-  
чуть не сложнее:  
>>> col2 = [row[1] for row in M] # Выбирает элементы второго столбца  
>>> col2  
[2, 5, 8]  
   
>>> M # Матрица не изменилась  
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
Генераторы списков следуют традиции системы представления множеств� они   
позволяют создавать новые списки, выполняя выражение для каждого элемен-  
та в последовательности, по одному за раз, слева направо. Генераторы списков   
заключены в квадратные скобки (чтобы отразить тот факт, что они создают   
список) и составлены из выражения и конструкции цикла, которые использу-  
ют одно и то же имя переменной (в данном случае row). В предыдущем примере   
генератор списков интерпретируется так: «Получить элементы row[1] из каж-  
дой строки матрицы M и создать из них новый список». Результатом является   
новый список, содержащий значения из второго столбца матрицы.  
На практике генераторы списков могут приобретать еще более сложную фор-  
му:  
>>> [row[1] + 1 for row in M] # Добавить 1 к каждому элементу в столбце 2  
[3, 6, 9]  
   
>>> [row[1] for row in M if row[1] % 2 == 0] # отфильтровать нечетные значения  
[2, 8]  
Первая операция в этом примере прибавляет 1 к значениям всех отобранных   
элементов, а вторая использует условный оператор if для исключения из ре-  
зультата нечетных чисел с помощью операции деления по модулю – % (остаток   
от деления). Генераторы списков, применяемые к спискам, возвращают в каче-  
стве результатов новые списки, но могут использоваться и для любых других   
объектов, допускающих выполнение итераций. Например, ниже показано ис-  
пользование генератора списков для обхода жестко заданного в программном   
коде списка координат и строк:  
>>> diag = [M[i][i] for i in [0, 1, 2]] # Выборка элементов диагонали матрицы  
>>> diag  
[1, 5, 9]  
   
>>> doubles = [c \* 2 for c in ‘spam’] # Дублирование символов в строке  
>>> doubles  
[‘ss’, ‘pp’, ‘aa’, ‘mm’]  
Генераторы списков и родственные им встроенные функции map и filter – на   
мой взгляд, достаточно сложная тема, чтобы говорить о них здесь более под-  
робно. Главная цель этого краткого обзора состоит в том, чтобы проиллюстри-  
ровать наличие как простых, так и очень сложных инструментов в арсенале   
Python. Можно обойтись и без генераторов списков, но на практике они ока-  
зываются очень удобными и нередко обеспечивают более высокую производи-

Словари   
137  
тельность при работе со списками. Кроме того, их можно применять к любым   
типам, являющимся последовательностями в языке Python, а также к некото-  
рым типам, которые не являются последовательностями. Мы еще будем гово-  
рить об этих выражениях далее в этой книге.  
Забегая вперед, отмечу, что в последних версиях Python можно также исполь-  
Python можно также исполь-  
 можно также исполь-  
зовать конструкции генераторов списков, заключенные в круглые скобки, для   
создания генераторов, которые воспроизводят результаты по требованию. На-  
пример, ниже показано, как с помощью встроенной функции sum можно сум-  
мировать элементы в последовательности:  
>>> G = (sum(row) for row in M) # Генератор, возвращающий суммы элементов строк  
>>> next(G)  
6  
>>> next(G) # Вызов в соответствии с протоколом итераций  
15  
Того же эффекта можно добиться с помощью встроенной функции map, гене-  
рируя результаты за счет передачи элементов другой функции. Обертывание   
вызова этой функции в список (в версии Python 3.0) вынуждает ее вернуть все   
значения:  
>>> list(map(sum, M)) # Отобразить sum на элементы в M  
[6, 15, 24]  
В Python 3.0 синтаксис генераторов списков может также использоваться для   
создания множеств и словарей:  
>>> {sum(row) for row in M} # Создаст множество сумм строк  
{24, 6, 15}  
>>> {i : sum(M[i]) for i in range(3)} # Таблица пар ключ/значение сумм строк  
{0: 6, 1: 15, 2: 24}  
Фактически в версии 3.0 с помощью подобных выражений-генераторов можно   
создавать списки, множества и словари:  
>>> [ord(x) for x in ‘spaam’] # Список кодов символов  
[115, 112, 97, 97, 109]  
>>> {ord(x) for x in ‘spaam’} # Множества ликвидируют дубликаты  
{112, 97, 115, 109}  
>>> {x: ord(x) for x in ‘spaam’} # Ключи словарей являются уникальными  
{‘a’: 97, ‘p’: 112, ‘s’: 115, ‘m’: 109}  
Однако нам следует двигаться дальше, чтобы познакомиться с такими объек-  
тами, как генераторы, множества и словари.  
Словари  
Словари в языке Python – это нечто совсем иное (по выражению Монти Пайто-  
на)� они вообще не являются последовательностями, это то, что известно как   
отображения. Отображения – это коллекции объектов, но доступ к ним осу-  
ществляется не по определенным смещениям от начала коллекции, а по клю-  
чам. В действительности отображения вообще не подразумевают какого-либо   
упорядочения элементов по их позиции, они просто отображают ключи на свя-  
занные с ними значения. Словари – единственный тип отображения в наборе   
базовых объектов Python – также относятся к классу изменяемых объектов:

138   
Глава 4. Введение в типы объектов языка Python   
они могут изменяться непосредственно и в случае необходимости могут увели-  
чиваться и уменьшаться в размерах подобно спискам.  
Операции над отображениями  
Когда словарь определяется как литерал, программный код определения за-  
ключается в фигурные скобки и состоит из последовательности пар «ключ:   
значение». Словари удобно использовать всегда, когда возникает необходи-  
мость связать значения с ключами, например чтобы описать свойства чего-  
либо. В качестве примера рассмотрим следующий словарь, состоящий из   
трех элементов (с ключами «food» (продукт питания), «quantity» (количество)   
и «color» (цвет)):  
>>> D = {‘food’: ‘Spam’, ‘quantity’: 4, ‘color’: ‘pink’}  
Мы можем обращаться к элементам этого словаря по ключам и изменять зна-  
чения, связанные с ключами. Для доступа к элементам словаря используется   
тот же синтаксис, который используется для обращения к элементам последо-  
вательностей, только в квадратных скобках указывается не смещение относи-  
тельно начала последовательности, а ключ:  
>>> D[‘food’] # Получить значение, связанное с ключом ‘food’  
‘Spam’  
   
>>> D[‘quantity’] += 1 # Прибавить 1 к значению ключа ‘quantity’  
>>> D  
{‘food’: ‘Spam’, ‘color’: ‘pink’, ‘quantity’: 5}  
Несмотря на то, что форма определения словаря в виде литерала, заключен-  
ного в фигурные скобки, достаточно наглядна, на практике чаще встречаются   
другие способы создания словарей. Следующий пример начинается с создания   
пустого словаря, который затем заполняется по одному ключу за раз. В отли-  
чие от списков, не допускающих присваивание значений отсутствующим эле-  
ментам, присваивание значения по несуществующему ключу в словаре приво-  
дит к созданию этого ключа:  
>>> D = {}  
>>> D[‘name’] = ‘Bob’ # В результате присваивания создается ключ  
>>> D[‘job’] = ‘dev’  
>>> D[‘age’] = 40  
   
>>> D  
{‘age’: 40, ‘job’: ‘dev’, ‘name’: ‘Bob’}  
   
>>> print(D[‘name’])  
Bob  
В этом примере ключи словаря играют роль имен полей в записи, которая опи-  
сывает некоторого человека. В других приложениях словари могут использо-  
ваться для замены операций поиска, поскольку обращение к элементу словаря   
по ключу обычно выполняется быстрее, чем поиск, реализованный на языке   
Python.  
Еще раз о вложенности  
В предыдущем примере словарь использовался для описания гипотетической   
персоны с помощью трех ключей. Теперь предположим, что информация име-

Словари   
139  
ет более сложную структуру. Возможно, придется записать имя и фамилию,   
а также несколько названий должностей, занимаемых одновременно. Это   
приводит к необходимости использования вложенных объектов Python. Сло-  
варь в следующем примере определен в виде литерала и имеет более сложную   
структуру:  
>>> rec = {‘name’: {‘first’: ‘Bob’, ‘last’: ‘Smith’},  
 ‘job’: [‘dev’, ‘mgr’],  
 ‘age’: 40.5}  
Здесь мы опять имеем словарь, содержащий три ключа верхнего уровня (клю-  
чи «name» (имя), «job» (должность) и «age» (возраст)), однако значения имеют   
более сложную структуру: для описания имени человека используется вло-  
женный словарь, чтобы обеспечить поддержку имен, состоящих из нескольких   
частей, и для перечисления занимаемых должностей используется вложенный   
список, что обеспечит возможность расширения в будущем. К компонентам   
этой структуры можно обращаться почти так же, как мы делали это в случае   
с матрицей, но на этот раз вместо числовых индексов мы используем ключи   
словаря:  
>>> rec[‘name’] # ‘Name’ – это вложенный словарь  
{‘last’: ‘Smith’, ‘first’: ‘Bob’}  
   
>>> rec[‘name’][‘last’] # Обращение к элементу вложенного словаря  
‘Smith’  
   
>>> rec[‘job’] # ‘Job’ – это вложенный список  
[‘dev’, ‘mgr’]  
   
>>> rec[‘job’][-1] # Обращение к элементу вложенного списка  
‘mgr’  
   
>>> rec[‘job’].append(‘janitor’) # Расширение списка должностей Боба (Bob)  
>>> rec  
{‘age’: 40.5, ‘job’: [‘dev’, ‘mgr’, ‘janitor’], ‘name’: {‘last’: ‘Smith’,   
‘first’: ‘Bob’}}  
Обратите внимание, как последняя операция в этом примере выполняет рас-  
ширение вложенного списка. Так как список должностей – это отдельный от   
словаря участок в памяти, он может увеличиваться и уменьшаться без каких-  
либо ограничений (размещение объектов в памяти будет обсуждаться в этой   
книге позже).  
Основная цель демонстрации этого примера состоит в том, чтобы показать вам   
гибкость базовых типов данных в языке Python. Здесь вы можете видеть, что   
возможность вложения позволяет легко воспроизводить достаточно сложные   
структуры данных. Для создания подобной структуры на языке C потребова-  
лось бы приложить больше усилий и написать больше программного кода: нам   
пришлось бы описать и объявить структуры и массивы, заполнить их значе-  
ниями, связать их между собой и так далее. В языке Python все это делает-  
ся автоматически – запуск выражения приводит к созданию всей структуры   
вложенных объектов. Фактически это одно из основных преимуществ языков   
сценариев, таких как Python.  
Так же, как и в низкоуровневых языках программирования, мы могли бы   
выполнить освобождение памяти, занимаемой объектами, которые стали не   
нужны. В языке Python память освобождается автоматически, когда теряет-

140   
Глава 4. Введение в типы объектов языка Python   
ся последняя ссылка на объект, например в случае присваивания переменной   
какого-либо другого значения:  
>>> rec = 0 # Теперь память, занятая объектом, будет освобождена  
С технической точки зрения, интерпретатор Python обладает такой особенно-  
стью, как сборка  мусора, благодаря которой в ходе выполнения программы   
производится освобождение неиспользуемой памяти, что избавляет нас от   
необходимости предусматривать специальные действия в программном коде.   
Интерпретатор освобождает память сразу же, как только будет ликвидирова-  
на последняя ссылка на объект. С работой этого механизма мы познакомимся   
далее, в этой же книге, а пока достаточно знать, что вы можете работать с объ-  
ектами, не беспокоясь о выделении или освобождении памяти для них.1  
Сортировка по ключам: циклы for  
Будучи отображениями, как мы уже видели, словари поддерживают доступ   
к элементам только по ключам. Однако они кроме того поддерживают ряд   
специфических для данного типа операций, реализованных в виде методов,   
которые удобно использовать в разных случаях.  
Как уже упоминалось ранее, из-за того, что словари не являются последова-  
тельностями, они не предусматривают какой-либо надежный способ упорядо-  
чения позиций элементов. Это означает, что если мы создадим словарь и по-  
пытаемся вывести его содержимое, порядок следования ключей при выводе   
может не совпадать с порядком, в каком они определялись:  
>>> D = {‘a’: 1, ‘b’: 2, ‘c’: 3}  
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2}  
Как же быть, если нам действительно потребуется упорядочить элементы сло-  
варя? В наиболее общем случае мы могли бы получить список всех ключей   
словаря методом keys, отсортировать их с помощью метода списка sort и затем   
выполнить обход значений в цикле for (не забудьте дважды нажать клавишу   
Enter после ввода цикла for ниже – как уже пояснялось в главе 3, пустая строка   
в интерактивной оболочке означает окончание составной инструкции):  
>>> Ks = list(D.keys()) # Неупорядоченный список ключей  
>>> Ks # Список – в версии 2.6, а в 3.0 – “представление”,  
[‘a’, ‘c’, ‘b’] # поэтому необходимо использовать функцию list()  
   
>>> Ks.sort() # Сортировка списка ключей  
>>> Ks  
[‘a’, ‘b’, ‘c’]  
   
>>> for key in Ks: # Обход отсортированного списка ключей  
 print(key, ‘=>’, D[key]) # Здесь дважды нажмите клавишу Enter  
   
1   
Имейте в виду, что запись rec, которую мы создали здесь, в действительности может   
быть записью в базе данных, если бы мы использовали систему хранения объектов   
Python – простейший способ хранения объектов Python в файлах или в базах дан-  
ных, обеспечивающих доступ по ключу. Мы не будем здесь углубляться в подробно-  
сти, однако позднее мы вернемся к этому вопросу, когда будем рассматривать модули   
pickle и shelve.

Словари   
141  
a => 1  
b => 2  
c => 3  
Этот процесс, состоящий из трех этапов, в последних версиях Python можно   
упростить до единственной операции, как будет показано в последующих гла-  
вах, с помощью новой встроенной функции sorted . Эта функция сортирует   
объекты разных типов и возвращает результат:  
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2}  
   
>>> for key in sorted(D):  
 print(key, ‘=>’, D[key])  
   
a => 1  
b => 2  
c => 3  
Этот пример может служить поводом для знакомства с циклом for языка   
Python. Цикл for представляет собой самый простой и эффективный способ   
произвести обход всех элементов в последовательности и выполнить блок про-  
граммного кода для каждого из элементов. Переменная цикла, определяемая   
пользователем (в данном случае key), служит для ссылки на текущий элемент.   
В этом примере выводятся ключи и значения несортированного словаря в отсо-  
ртированном по ключам виде.  
Цикл for и родственный ему цикл while – это основные способы реализации по-  
вторяющихся действий в сценариях. Однако в действительности цикл for (так   
же, как и родственные ему генераторы списков, с которыми мы познакомились   
выше), является операцией над последовательностью. Он способен работать   
с любыми объектами, являющимися последовательностями, а также с неко-  
торыми объектами, которые последовательностями не являются. Ниже при-  
водится пример обхода всех символов в строке и вывод их в верхнем регистре:  
>>> for c in ‘spam’:  
 print(c.upper())  
   
S  
P  
A  
M  
Цикл while в языке Python представляет собой более универсальный инстру-  
Python представляет собой более универсальный инстру-  
 представляет собой более универсальный инстру-  
мент выполнения циклически повторяющихся операций и не имеет прямой   
связи с последовательностями:  
>>> x = 4  
>>> while x > 0:  
 print(‘spam!’ \* x)  
 x -= 1  
   
spam!spam!spam!spam!  
spam!spam!spam!  
spam!spam!  
spam!

142   
Глава 4. Введение в типы объектов языка Python   
Инструкции циклов, их синтаксис и свойства мы подробнее рассмотрим ниже   
в этой книге.  
Итерации и оптимизация  
Не случайно цикл for выглядит похожим на выражения-генераторы, введен-  
ные ранее: каждый из этих инструментов представляет собой универсальное   
средство выполнения итераций. Фактически обе конструкции способны рабо-  
тать с любыми объектами, которые поддерживают протокол итераций – идею,   
недавно появившуюся в Python, которая по сути подразумевает наличие в па-  
мяти последовательности или объекта, который генерирует по одному элемен-  
ту за раз в контексте выполнения итерации. Объект попадает в категорию ите-  
рируемых, если в ответ на вызов встроенной функции iter (с этим объектом   
в качестве аргумента) возвращается объект, который позволяет перемещаться   
по его элементам с помощью функции next. Генераторы списков, с которыми   
мы познакомились выше, являются такими объектами.   
О протоколе итераций я расскажу позднее в этой же книге. А пока просто за-  
помните, что любой инструмент языка Python, сканирующий объект слева   
направо, использует протокол итераций. Именно поэтому функция sorted, ко-  
торая использовалась в предыдущем разделе, способна работать со словарем   
непосредственно – нам не требуется вызывать метод keys для получения после-  
довательности, потому что словари являются итерируемыми объектами, для   
которых функция next возвращает следующий ключ.  
Это также означает, что любой генератор списков, такой, как показано ниже,   
вычисляющий квадраты чисел в списке:  
>>> squares = [x \*\* 2 for x in [1, 2, 3, 4, 5]]  
>>> squares  
[1, 4, 9, 16, 25]  
всегда можно представить в виде эквивалентного цикла for, который создает   
список с результатами, добавляя новые элементы в ходе выполнения итераций:  
>>> squares = []  
>>> for x in [1, 2, 3, 4, 5]: # Эти же операции выполняет и генератор списков,  
 squares.append(x \*\* 2) # следуя протоколу итераций  
   
>>> squares  
[1, 4, 9, 16, 25]  
Однако генераторы списков и родственные им инструменты функционального   
программирования, такие как функции map и filter, обычно выполняются бы-  
стрее, чем цикл for (примерно раза в два), что особенно важно для программ,   
обрабатывающих большие объемы данных. И, тем не менее, следует заметить,   
что оценка производительности – вещь очень хитрая в языке Python, потому   
что в процессе разработки он продолжает оптимизироваться, и производитель-  
ность тех или иных конструкций может изменяться от версии к версии.  
Главное правило, которому желательно следовать при использовании языка   
Python – это простота и удобочитаемость программного кода, а проблему про-  
изводительности следует рассматривать во вторую очередь, уже после того,   
как будет создана работоспособная программа и когда проблема производи-  
тельности программы действительно заслуживает того, чтобы на нее обрати-  
ли внимание. Но чаще всего ваш программный код будет обладать достаточ-

Словари   
143  
но высокой производительностью. Если же вам потребуется оптимизировать   
программу, в составе Python вы найдете инструменты, которые помогут вам   
в этом, включая модули time и timeit, а также модуль profile. Более подробную   
информацию об этих модулях вы найдете далее в книге и в руководствах по   
языку Python.  
Отсутствующие ключи:   
проверка с помощью оператора if  
Необходимо сделать еще одно замечание о словарях, прежде чем двинуться   
дальше. Несмотря на то, что операция присваивания значений элементам с не-  
существующими ключами приводит к расширению словаря, тем не менее, при   
попытке обратиться к несуществующему элементу возникает ошибка:  
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2}  
   
>>> D[‘e’] = 99 # Присваивание по новому ключу приводит к расширению словаря  
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2, ‘e’: 99}  
   
>>> D[‘f’] # Попытка обратиться к несуществующему ключу приводит к ошибке  
...текст сообщения об ошибке опущен...  
KeyError: ‘f’  
Программная ошибка при попытке получить значение несуществующего эле-  
мента – это именно то, что нам хотелось бы получать. Но в общем случае при   
создании программного кода мы не всегда можем знать, какие ключи будут   
присутствовать. Как быть в таких случаях, чтобы не допустить появления   
ошибок? Для этого можно, например, выполнить предварительную проверку.   
Применительно к словарям оператор in, проверки на членство, позволяет опре-  
делить наличие ключа и с помощью условного оператора if выполнить тот или   
иной программный код (как и в случае инструкции for, не забудьте дважды   
нажать клавишу Enter после ввода инструкции if в интерактивной оболочке):  
>>> ‘f’ in D  
False  
   
>>> if not ‘f’ in D:  
 Print(‘missing’)  
   
missing  
Далее в этой книге я расскажу подробнее об инструкции if и ее синтаксисе.   
Однако форма инструкции, которая используется здесь, достаточно очевидна:   
она состоит из ключевого слова if, следующего за ним выражения, результат   
которого интерпретируется как «истина» или «ложь». Далее следует блок про-  
граммного кода, который будет выполнен, если результатом выражения будет   
значение «истина». В полной форме инструкция if предусматривает наличие   
предложения else – для реализации действия по умолчанию, и одно или бо-  
лее предложение elif (else if) для выполнения других проверок. Это основное   
средство выбора в языке Python и именно этим способом мы реализуем логику   
работы в наших сценариях.  
Существуют также и другие способы создания словарей и исключения оши-  
бок обращения к несуществующим элементам словаря: метод get (при обраще-

144   
Глава 4. Введение в типы объектов языка Python   
нии к которому можно указать значение, возвращаемое по умолчанию)� в Py-  
Py-  
thon 2.X имеется метод has\_key (который недоступен в версии 3.0)� инструкция   
try (с которой мы познакомимся в главе 10, позволяющая перехватывать и об-  
рабатывать исключения) и выражение if/else (по сути тот же условный опера-  
тор if, сжатый до размеров одной строки), например:  
>>> value = D.get(‘x’, 0) # Попытка получить значение,   
>>> value # указав значение по умолчанию  
0  
>>> value = D[‘x’] if ‘x’ in D else 0 # Выражение if/else   
>>> value  
0  
Но подробнее об этом мы поговорим в одной из следующих глав. А сейчас рас-  
смотрим кортежи.  
Кортежи  
Объект-кортеж (tuple – произносится как «тъюпл» или «тъюпел», в зависимо-  
uple – произносится как «тъюпл» или «тъюпел», в зависимо-  
 – произносится как «тъюпл» или «тъюпел», в зависимо-  
сти от того, у кого вы спрашиваете) в общих чертах напоминает список, ко-  
торый невозможно изменить – кортежи являются последовательностями, как   
списки, но они являются неизменяемыми, как строки. Синтаксически литерал   
кортежа заключается в круглые, а не в квадратные скобки. Они также поддер-  
живают включение объектов различных типов, вложение и операции, типич-  
ные для последовательностей:  
>>> T = (1, 2, 3, 4) # Кортеж из 4 элементов  
>>> len(T) # Длина  
4  
   
>> T + (5, 6) # Конкатенация  
(1, 2, 3, 4, 5, 6)  
   
>>> T[0] # Извлечение элемента, среза и так далее  
1  
В Python 3.0 кортежи обладают двумя методами, которые также имеются   
у списков:  
>>> T.index(4) # Методы кортежей: значение 4 находится в позиции 3  
3  
>>> T.count(4) # Значение 4 присутствует в единственном экземпляре  
1  
Основное отличие кортежей – это невозможность их изменения после созда-  
ния. То есть кортежи являются неизменяемыми последовательностями:  
>>> T[0] = 2 # Кортежи являются неизменяемыми  
...текст сообщения об ошибке опущен...  
TypeError: ‘tuple’ object does not support item assignment  
Подобно спискам и словарям кортежи способны хранить объекты разных ти-  
пов и допускают возможность вложения, но в отличие от них, не могут изме-  
нять свои размеры, так как являются неизменяемыми объектами:  
>>> T = (‘spam’, 3.0, [11, 22, 33])  
>>> T[1]

Файлы   
145  
3.0  
>>> T[2][1]  
22  
>>> T.append(4)  
AttributeError: ‘tuple’ object has no attribute ‘append’  
Для чего нужны кортежи?  
Зачем нужен тип, который напоминает список, но поддерживает меньшее чис-  
ло операций? Откровенно говоря, на практике кортежи используются не так   
часто, как списки, и главное их достоинство – неизменяемость. Если коллек-  
ция объектов передается между компонентами программы в виде списка, он   
может быть изменен любым из компонентов. Если используются кортежи, та-  
кие изменения становятся невозможны. То есть кортежи обеспечивают своего   
рода ограничение целостности, что может оказаться полезным в крупных про-  
граммах. Далее в книге мы еще вернемся к кортежам. А сейчас перейдем к по-  
следнему базовому типу данных – к файлам.  
Файлы  
Объекты-файлы – это основной интерфейс между программным кодом на язы-  
ке Python и внешними файлами на компьютере. Файлы являются одним из   
базовых типов, но они представляют собой нечто необычное, поскольку для   
файлов отсутствует возможность создания объектов в виде литералов. Вместо   
этого, чтобы создать объект файла, необходимо вызвать встроенную функцию   
open, передав ей имя внешнего файла и строку режима доступа к файлу. На-  
пример, чтобы создать файл для вывода данных, вместе с именем файла функ-  
ции необходимо передать строку режима ‘w’:  
>>> f = open(‘data.txt’, ‘w’) # Создается новый файл для вывода  
>>> f.write(‘Hello\n’) # Запись строки байтов в файл  
6  
>>> f.write(‘world\n’) # В Python 3.0 возвращает количество записанных байтов  
6  
>>> f.close() # Закрывает файл и выталкивает выходные буферы на диск  
В этом примере создается файл в текущем каталоге и в него записывается текст   
(имя файла может содержать полный путь к каталогу, если требуется получить   
доступ к файлу, находящемуся в другом месте). Чтобы прочитать то, что было   
записано в файл, его нужно открыть в режиме ‘r’ (этот режим используется по   
умолчанию, если строка режима в вызове функции опущена). Затем прочитать   
содержимое файла в строку и отобразить ее. Содержимое файла для сценария   
всегда является строкой независимо от типов данных, фактически хранящих-  
ся в файле:  
>>> f = open(‘data.txt’) # ‘r’ – это режим доступа к файлу по умолчанию  
>>> text = f.read() # Файл читается целиком в строку  
>>> text  
‘Hello\nworld\n’  
   
>>> print(text) # Вывод, с попутной интерпретацией служебных символов  
Hello  
world

146   
Глава 4. Введение в типы объектов языка Python   
>>> text.split() # Содержимое файла всегда является строкой  
[‘Hello’, ‘world’]  
Объекты-файлы имеют и другие методы, обеспечивающие поддержку допол-  
нительных возможностей, но пока мы не будем рассматривать их. Например,   
объекты файлов предоставляют различные способы чтения и записи данных   
(метод read принимает необязательный параметр – количество байтов, метод   
readline считывает по одной строке за одно обращение и так далее) и другие   
методы (seek – перемещает позицию чтения/записи в файле). Однако, как будет   
показано позднее, самый лучший на сегодняшний день способ чтения файлов   
состоит в том, чтобы не читать его содержимое целиком – файлы предостав-  
ляют итераторы, которые обеспечивают автоматическое построчное чтение   
содержимого файла в циклах for и в других контекстах.  
Мы будем рассматривать все множество методов позднее в этой книге, но если   
у вас появится желание быстро ознакомиться с ними, запустите функцию dir,   
передав ей слово file (имя типа данных), а затем функцию help с любым из   
имен методов в качестве аргумента:  
>>> dir(f)  
[ ...множество имен опущено...  
‘buffer’, ‘close’, ‘closed’, ‘encoding’, ‘errors’, ‘fileno’, ‘flush’, ‘isatty’,   
‘line\_buffering’, ‘mode’, ‘name’, ‘newlines’, ‘read’, ‘readable’, ‘readline’,   
‘readlines’, ‘seek’, ‘seekable’, ‘tell’, ‘truncate’, ‘writable’, ‘write’,   
‘writelines’]  
>>> help(f.seek)  
...попробуйте и увидите...  
Позднее в этой книге вы узнаете также, что при работе с файлами в Python 3.0   
проводится очень четкая грань между текстовыми и двоичными данными. Со-  
держимое текстовых файлов представляется в виде строк и для них автома-  
тически выполняется кодирование и декодирование символов Юникода. Со-  
держимое двоичных файлов представляется в виде строк специального типа   
bytes, при этом никаких автоматических преобразований содержимого файлов   
не производится:  
>>> data = open(‘data.bin’, ‘rb’).read()# Файл открывается в двоичном режиме  
>>> data # Строка байтов хранит двоичные данные  
b’\x00\x00\x00\x07spam\x00\x08’  
>>> data[4:8]  
b’spam’  
Хотя при работе исключительно с текстовыми данными в формате ASCII о та-  
ASCII о та-  
 о та-  
ких различиях обычно беспокоиться не приходится, однако в Python 3.0 стро-  
Python 3.0 стро-  
 3.0 стро-  
ки и файлы требуют особого внимания – при работе с интернационализиро-  
ванными приложениями или двоичными данными.  
Другие средства, напоминающие файлы  
Функция open – это рабочая лошадка в большинстве операций с файлами, ко-  
торые можно выполнять в языке Python. Для решения более специфичных   
задач Python поддерживает и другие инструментальные средства, напомина-  
ющие файлы: каналы, очереди, сокеты, файлы с доступом по ключу, файлы-  
хранилища объектов, файлы с доступом по дескриптору, интерфейсы к реля-  
ционным и объектно-ориентированным базам данных и многие другие. Файлы

Другие базовые типы   
147  
с доступом по дескриптору, например, поддерживают возможность блокировки   
и другие низкоуровневые операции, а сокеты представляют собой интерфейс   
к сетевым взаимодействиям. В этой книге мы не будем подробно рассматривать   
эти темы, но знание этих особенностей окажется для вас полезным, как только   
вы начнете всерьез программировать на языке Python.  
Другие базовые типы  
Помимо базовых типов данных, которые мы уже рассмотрели, существуют   
и другие, которые могут считаться базовыми в зависимости от широты опреде-  
ления этой категории. Например, множества, совсем недавно появившиеся   
в языке, – которые не являются ни последовательностями, ни отображения-  
ми. Множества – это неупорядоченные коллекции уникальных и неизменяе-  
мых объектов. Множества создаются встроенной функцией set или с помощью   
новых синтаксических конструкций определения литералов и генераторов   
множеств, появившихся в версии 3.0, и поддерживают типичные математиче-  
ские операции над множествами (выбор синтаксической конструкции {...} для   
определения литералов множеств в версии 3.0 не случаен, поскольку множе-  
ства напоминают словари, в которых ключи не имеют значений):  
>>> X = set(‘spam’) # В 2.6 и 3.0 можно создавать из последовательностей  
>>> Y = {‘h’, ‘a’, ‘m’} # В 3.0 можно определять литералы множеств  
>>> X, Y  
({‘a’, ‘p’, ‘s’, ‘m’}, {‘a’, ‘h’, ‘m’})  
   
>>> X & Y # Пересечение  
{‘a’, ‘m’}  
   
>>> X | Y # Объединение  
{‘a’, ‘p’, ‘s’, ‘h’, ‘m’}  
   
>>> X – Y # Разность  
{‘p’, ‘s’}  
>>> {x \*\* 2 for x in [1, 2, 3, 4]} # Генератор множеств в 3.0  
{16, 1, 4, 9}  
Кроме того, недавно в Python появились вещественные числа с фиксирован-  
ной точностью и рациональные числа (числа, представленные дробью, то есть   
парой целых чисел – числителем и знаменателем). Обе разновидности могут   
использоваться для решения проблем, связанных с точностью представления   
простых вещественных чисел:  
>>> 1 / 3 # Вещественное число (в 2.6 числа должны заканчиваться .0)  
0.33333333333333331  
>>> (2/3) + (1/2)   
1.1666666666666665  
   
>>> import decimal # Вещественные числа с фиксированной точностью  
>>> d = decimal.Decimal(‘3.141’)  
>>> d + 1  
Decimal(‘4.141’)  
   
>>> decimal.getcontext().prec = 2  
>>> decimal.Decimal(‘1.00’) / decimal.Decimal(‘3.00’)  
Decimal(‘0.33’)

148   
Глава 4. Введение в типы объектов языка Python   
>>> from fractions import Fraction # Рациональные числа: числитель+знаменатель  
>>> f = Fraction(2, 3)  
>>> f + 1  
Fraction(5, 3)  
>>> f + Fraction(1, 2)  
Fraction(7, 6)  
Кроме того в языке Python имеется логический тип данных (представленный   
предопределенными объектами True и False, которые по сути являются обыч-  
ными целыми числами 1 и 0 с некоторыми особенностями отображения на   
экране), а кроме того, давно уже существует специальный объект None, обычно   
используемый для инициализации переменных и объектов:  
>>> 1 > 2, 1 < 2 # Логические значения  
(False, True)  
>>> bool(‘spam’)  
True  
   
>>> X = None # Специальный объект None  
>>> print(X)  
None  
   
>>> L = [None] \* 100 # Инициализация списка сотней объектов None  
>>> L  
[None, None, None, None, None, None, None, None, None, None, None, None,  
None, None, None, None, None, None, None, ...список из 100 объектов None...]  
Как можно нарушить гибкость программного кода  
Мы еще будем много говорить обо всех этих типах данных далее в книге, но сна-  
чала я хочу сделать важное замечание. Тип объекта, возвращаемый встроенной   
функцией type, в свою очередь сам является объектом. В Python 3.0 этот объект   
несколько отличается от того, что возвращается в версии 2.6, потому что все   
типы были объединены с классами (о которых мы будем говорить при изучении   
классов «нового стиля» в шестой части книги). Допустим, что переменная L по-  
прежнему представляет список, созданный в предыдущем разделе:  
# В Python 2.6:  
   
>>> type(L) # Типы: переменная L представляет объект типа list  
<type ‘list’>  
>>> type(type(L)) # Даже сами типы являются объектами  
<type ‘type’>  
   
# В Python 3.0:  
   
>>> type(L) # 3.0: типы являются классами, и наоборот  
<class ‘list’>  
>>> type(type(L)) # Подробнее о классах типов рассказывается в главе 31  
<class ‘type’>  
Типы объектов можно исследовать не только в интерактивной оболочке, но и в   
программном коде, который использует эти объекты. Сделать это в сценариях   
на языке Python можно как минимум тремя способами:  
>>> if type(L) == type([]): # Проверка типа, если в этом есть необходимость...  
 print(‘yes’)  
   
yes

Другие базовые типы   
149  
>>> if type(L) == list: # С использованием имени типа  
 print(‘yes’)  
   
yes  
   
>>> if isinstance(L, list): # Проверка в объектно-ориентированном стиле  
 print(‘yes’)  
   
yes  
Однако теперь, когда я показал вам все эти способы проверки типа объекта,   
я должен заметить, что использование таких проверок в программном коде   
практически всегда является неверным решением (и отличительным призна-  
ком бывшего программиста на языке C, приступившего к программированию   
на языке Python). Причина, почему такой подход считается неверным, станет   
понятна позднее, когда мы начнем писать более крупные блоки программного   
кода, такие как функции, – но это (пожалуй, самая) основная концепция язы-  
ка Python. Наличие проверок на принадлежность объекта к тому или иному   
типу отрицательно сказывается на гибкости программного кода, потому что   
вы ограничиваете его работой с единственным типом данных. Без таких про-  
верок ваш программный код может оказаться в состоянии работать с более ши-  
роким диапазоном типов.  
Это связано с идей полиморфизма, о которой упоминалось ранее, и это основ-  
ная причина отсутствия необходимости описывать типы переменных в языке   
Python. Как будет говориться далее, программный код на языке Python ориен-  
. Как будет говориться далее, программный код на языке Python ориен-  
Python ориен-  
 ориен-  
тируется на использование интерфейсов объектов (наборов поддерживаемых   
операций), а не их типов. Отсутствие заботы об определенных типах означа-  
ет, что программный код автоматически может обслуживать большинство   
из них – допустимыми будут любые объекты с совместимыми интерфейсами   
независимо от конкретного типа. И хотя контроль типов поддерживается,   
а в редких случаях даже необходим, тем не менее, такой способ мышления   
чужд языку Python. Вы сами убедитесь, что полиморфизм является ключевой   
идеей, обеспечивающей успех использования Python.  
Классы, определяемые пользователем  
Мы подробно рассмотрим объектно-ориентированный стиль программирова-  
ния на языке Python, который позволяет сократить время, затрачиваемое на   
разработку, далее в этой книге. Тем не менее, говоря абстрактными термина-  
ми, классы определяют новые типы объектов, которые расширяют базовый на-  
бор, и потому они заслуживают упоминания здесь. Например, вам мог бы по-  
требоваться такой тип объектов, который моделировал бы сотрудников. В язы-  
ке Python нет такого базового типа, тем не менее, следующий класс вполне мог   
бы удовлетворить ваши потребности:  
>>> class Worker:  
 def \_\_init\_\_(self, name, pay): # Инициализация при создании  
 self.name = name # self – это сам объект  
 self.pay = pay  
 def lastName(self):  
 return self.name.split()[-1] # Разбить строку по символам пробела  
 def giveRaise(self, percent):  
 self.pay \*= (1.0 + percent) # Обновить сумму выплат

150   
Глава 4. Введение в типы объектов языка Python   
Данный класс определяет новый тип объектов, которые обладают атрибута-  
ми name и pay (иногда атрибуты называют информацией о состоянии), а так-  
же двумя описаниями поведения, оформленными в виде функций (которые   
обычно называют методами). Обращение к имени класса как к функции при-  
водит к созданию экземпляра нового типа, а методы класса автоматически   
получают ссылку на текущий экземпляр, обрабатываемый этими методами   
(аргумент self):  
>>> bob = Worker(‘Bob Smith’, 50000) # Создаются два экземпляра и для каждого  
>>> sue = Worker(‘Sue Jones’, 60000) # определяется имя и сумма выплат  
>>> bob.lastName() # Вызов метода: self – это bob  
‘Smith’  
>>> sue.lastName() # self – это sue   
‘Jones’  
>>> sue.giveRaise(.10) # Обновить сумму выплат для sue  
>>> sue.pay  
66000.0  
Модель называется объектно-ориентированной потому, что здесь присутству-  
ет подразумеваемый объект «self»: внутри функций, определяемых в классах,   
всегда присутствует подразумеваемый объект. В некотором смысле типы, осно-  
ванные на классах, просто создаются на базе основных типов и используют   
их функциональные возможности. В данном случае пользовательский класс   
Worker – это всего лишь коллекция, состоящая из строки и числа (name и pay со-  
ответственно), плюс функции, выполняющие обработку этих двух встроенных   
объектов.  
Дополнительно о классах можно сказать, что их механизм наследования под-  
держивает программные иерархии, которые допускают возможность их рас-  
ширения. Возможности программного обеспечения расширяются за счет соз-  
дания новых классов, но при этом не изменяется программный код, который   
уже работает. Кроме того, вы должны понимать, что классы в языке Python не   
являются обязательными к применению и нередко более простые встроенные   
типы, такие как списки и словари, оказываются эффективнее классов, опре-  
деляемых пользователем. Однако все это выходит далеко за рамки ознакоми-  
тельной главы, поэтому рассматривайте этот раздел лишь как предваритель-  
ное знакомство – полное описание возможности создания собственных типов   
данных в виде классов приводится в шестой части книги.  
И все остальное  
Как уже упоминалось ранее, все данные, которые обрабатываются сценариями   
на языке Python, являются объектами, поэтому наш краткий обзор типов объ-  
ектов никак нельзя назвать исчерпывающим. Однако даже при том, что все   
сущее в языке Python является «объектом», только рассмотренные типы обра-  
зуют базовый набор. Другие типы в языке Python являются либо элементами   
программ (такими как функции, модули, классы и объекты скомпилированно-  
го программного кода), к которым мы вернемся позже, либо реализуются им-  
портируемыми модулями и не являются синтаксическими элементами языка.   
Последние обычно имеют узкий круг применения – текстовые шаблоны, ин-  
терфейсы доступа к базам данных, сетевые соединения и так далее.  
Более того, имейте в виду, что объекты, с которыми мы здесь познакомились,   
действительно являются объектами, но для работы с ними не требуется ис-

В заключение   
151  
пользовать объектно-ориентированный подход – концепцию, которая обычно   
подразумевает использование механизма наследования и оператора class, с ко-  
торым мы еще встретимся далее в этой книге. Однако базовые объекты языка   
Python – это рабочие лошадки практически любого сценария, и, как правило,   
они являются основой более крупных типов, не являющихся базовыми.  
В заключение  
На этом мы заканчиваем наш краткий обзор типов данных. В этой главе ва-  
шему вниманию было предложено краткое введение в базовые типы объектов   
языка Python и операции, которые могут к ним применяться. Мы рассмотрели   
наиболее универсальные операции, которые могут применяться к объектам   
различных типов (операции над последовательностями, такие как обращение   
к элементам по их индексам и извлечение срезов), а также операции, специ-  
фичные для определенных типов, реализованные в виде методов (например,   
разбиение строки и добавление элементов в список). Здесь также были даны   
определения некоторых ключевых терминов, такие как неизменность, после-  
довательности и полиморфизм.  
Наряду с этим мы узнали, что базовые типы данных в языке Python облада-  
ют большей гибкостью и более широкими возможностями, чем типы данных,   
доступные в низкоуровневых языках программирования, таких как C. На-  
пример, списки и словари избавляют нас от необходимости реализовать про-  
граммный код поддержки коллекций и поиска. Списки – это упорядоченные   
коллекции объектов, а словари – это коллекции объектов, доступ к которым   
осуществляется по ключу, а не по позиции. И словари, и списки могут быть   
вложенными, могут увеличиваться и уменьшаться по мере необходимости   
и могут содержать объекты любых типов. Более того, память, занимаемая   
ими, автоматически освобождается, как только будет утрачена последняя   
ссылка на них.  
Я опустил большую часть подробностей здесь, чтобы сделать знакомство как   
можно более кратким, поэтому вы не должны считать, что эта глава содержит   
все, что необходимо. В следующих главах мы будем рассматривать базовые   
типы языка более подробно, благодаря чему вы сможете получить более пол-  
ную картину. В следующей главе мы приступим к всестороннему изучению чи-  
сел в языке Python. Но для начала ознакомьтесь с контрольными вопросами.  
Закрепление пройденного  
Контрольные вопросы  
В следующих главах мы более подробно будем исследовать понятия, введенные   
в этой главе, поэтому здесь мы охватим лишь самые общие направления:  
1. Назовите четыре базовых типа данных в языке Python.  
2. Почему они называются базовыми?  
3. Что означает термин «неизменяемый» и какие три базовых типа языка   
Python являются неизменяемыми?  
4. Что означает термин «последовательность», и какие три типа относятся   
к этой категории?

152   
Глава 4. Введение в типы объектов языка Python   
5. Что означает термин «отображение» и какой базовый тип является отобра-  
жением?  
6. Что означает термин «полиморфизм», и почему он имеет такое важное зна-  
чение?  
Ответы  
1. К базовым типам объектов (данных) относятся числа, строки, списки, сло-  
вари, кортежи, файлы и множества. Cами типы, None и логические значе-  
ния также иногда относят к базовым типам. Существует несколько типов   
чисел (целые, вещественные, комплексные, рациональные и фиксирован-  
ной точности) и несколько типов строк (обычные и в кодировке Юникод –   
в Python 2.X� текстовые строки и строки байтов – в Python 3.0).  
2. Базовыми типами они называются потому, что являются частью самого   
языка Python и могут быть использованы в любой момент. Чтобы создать   
объект какого-либо другого типа, обычно бывает необходимо обращаться   
к функции из импортированного модуля. Для большинства базовых ти-  
пов предусмотрен специальный синтаксис создания объектов, например   
‘spam’, – это выражение, создающее строку и определяющее набор операций,   
которые могут применяться к ней. Вследствие этого базовые типы жестко   
вшиты в синтаксис языка Python. Единственное отличие – объекты-файлы,   
для создания которых необходимо вызывать функцию open.  
3. «Неизменяемый» объект – это объект, который невозможно изменить после   
того, как он будет создан. К этой категории объектов относятся числа, стро-  
ки и кортежи. Но даже при том, что вы не можете изменить «неизменяе-  
мый» объект на месте, вы всегда можете создать вместо него новый объект,   
выполнив выражение.  
4. «Последовательность» – это упорядоченная по местоположению коллекция   
объектов. К последовательностям относятся строки, списки и кортежи. Ко   
всем этим типам могут применяться операции, общие для всех последова-  
тельностей, такие как обращение к элементам по их индексам, конкатена-  
ция и получение срезов. Но помимо этого каждый из этих типов имеет ряд   
специфичных методов.  
5. Под термином «отображение» подразумевается объект, который отобража-  
ет ключи на ассоциированные с ними значения. Единственный базовый тип   
данных в языке Python, который является отображением, – это словарь.   
Отображения не подразумевают упорядочение элементов по их позиции, но   
они поддерживают возможность доступа к элементам по ключу, плюс ряд   
специфичных методов.  
6. «Полиморфизм» означает, что фактически выполняемая операция (такая   
как +) зависит от объектов, которые принимают в ней участие. В языке   
Python идея полиморфизма составляет ключевую концепцию (пожалуй,   
самую ключевую) – она не ограничивает применимость программного кода   
каким-то определенным типом данных, благодаря чему этот код обычно   
в состоянии автоматически обрабатывать объекты самых разных типов.

Глава 5.  
   
Числа  
Начиная с этой главы, мы станем погружаться в детали реализации язы-  
ка Python. Данные в этом языке имеют форму объектов – это либо встроен-  
ные объекты, входящие в состав самого языка, либо объекты, создаваемые   
с помощью языковых конструкций Python или других языков, таких как C.   
Фактически объекты являются основой любой программы на языке Python.   
Поскольку объекты представляют самое фундаментальное понятие для про-  
граммирования на языке Python, все наше внимание в первую очередь будет   
сосредоточено на объектах.  
В предыдущей главе мы коротко познакомились с базовыми типами объек-  
тов языка Python. И хотя в ней были даны основные понятия, мы старались   
избегать слишком специфичных подробностей в целях экономии книжного   
пространства. Теперь мы приступаем к более внимательному изучению кон-  
цепции типов данных, чтобы восполнить детали, о которых раньше умалчи-  
валось. Итак, приступим к изучению чисел – первой категории типов данных   
в языке Python.  
Базовые числовые типы  
Большинство числовых типов в языке Python не выделяются ничем необыч-  
ным и наверняка покажутся вам знакомыми, если в прошлом вам уже при-  
ходилось использовать какой-либо язык программирования. Числа могут ис-  
пользоваться для представления информации о состоянии вашего банковского   
счета, о расстоянии до Марса, о числе посетителей вашего веб-сайта и всего,   
что можно выразить в числовой форме.  
Числа в языке Python представлены не единственным типом, а целой катего-  
рией родственных типов. Язык Python поддерживает обычные числовые типы   
(целые и вещественные), а также литералы – для их создания, и выражения –   
для их обработки. Кроме того, Python предоставляет дополнительную под-  
держку чисел и объектов для работы с ними. Ниже приводится полный пере-  
чень числовых типов и инструментов, поддерживаемых в языке Python:  
 •  
Целые и вещественные числа  
 •  
Комплексные числа

154   
Глава 5. Числа   
 •  
Числа фиксированной точности  
 •  
Рациональные числа  
 •  
Множества  
 •  
Логические значения  
 •  
Целые числа неограниченной точности  
 •  
Различные встроенные функции и модули для работы с числами  
В этой главе мы начнем с рассмотрения базовых числовых типов и основных   
приемов работы с ними, а затем перейдем к исследованию остальных пунктов   
списка. Однако прежде чем начать писать программный код, в следующих не-  
скольких разделах мы сначала узнаем, как в сценариях записываются и об-  
рабатываются числа.  
Числовые литералы  
Помимо базовых типов данных язык Python предоставляет самые обычные   
числовые типы: целые числа (положительные и отрицательные) и веществен-  
ные числа (с дробной частью), которые иногда называют числами с плавающей   
точкой. Язык Python позволяет записывать целые числа в виде шестнадцате-  
ричных, восьмеричных и двоичных литералов. Поддерживает комплексные   
числа и обеспечивает неограниченную точность представления целых чисел   
(количество цифр в целых числах ограничивается лишь объемом доступной   
памяти). В табл. 5.1 показано, как выглядят числа различных типов в языке   
Python в тексте программы (то есть в виде литералов).  
Таблица 5.1. Числовые литералы  
Литерал  
Интерпретация  
1234, -24, 0, 9999999999999999999  
Обычные целые числа (с неограничен-  
ной точностью представления)  
1.23, 1., 3.14e-10, 4E210, 4.0e+210  
Вещественные числа  
0177, 0x9ff, 0b101010  
Восьмеричные, шестнадцатеричные   
и двоичные литералы целых чисел   
в версии 2.6  
0o177, 0x9ff, 0b101010  
Восьмеричные, шестнадцатеричные   
и двоичные литералы целых чисел   
в версии 3.0  
3+4j, 3.0+4.0j, 3J  
Литералы комплексных чисел  
Вообще в числовых типах языка Python нет ничего сложного, но мне хотелось   
бы сделать несколько замечаний о принципах использования литералов в про-  
граммном коде:  
Литералы целых и вещественных чисел  
Целые числа записываются как строки, состоящие из десятичных цифр.   
Вещественные числа могут содержать символ десятичной точки и/или не-  
обязательную экспоненту со знаком, которая начинается с символа e или E.   
Если в записи числа обнаруживается точка или экспонента, интерпретатор   
Python создает объект вещественного числа и использует вещественную (не

Базовые числовые типы   
155  
целочисленную) математику, когда такой объект участвует в выражении.   
Правила записи вещественных чисел в языке Python ничем не отличаются   
от правил записи чисел типа double в языке C и потому вещественные числа   
в языке Python обеспечивают такую же точность представления значений,   
как и в языке C.  
Целые числа в Python 2.6: обычные и длинные  
В Python 2.6 имеется два типа целых чисел: обычные (32-битные) и длин-  
ные (неограниченной точности). Если числовой литерал заканчивается   
символом l или L, он рассматривается интерпретатором как длинное целое   
число. Целые числа автоматически преобразуются в длинные целые, если   
их значения не умещаются в отведенные 32 бита, поэтому вам не требуется   
вводить символ L – интерпретатор автоматически выполнит необходимые   
преобразования, когда потребуется увеличить точность представления.  
Целые числа в Python 3.0: один тип  
В Python 3.0 обычные и длинные целые числа были объединены в один тип   
целых чисел, который автоматически поддерживает неограниченную точ-  
ность, как длинные целые в Python 2.6. По этой причине теперь не требует-  
Python 2.6. По этой причине теперь не требует-  
 2.6. По этой причине теперь не требует-  
ся завершать литералы целых чисел символом l или L, и при выводе целых   
чисел этот символ никогда не выводится. Кроме того, для большинства про-  
грамм это изменение в языке никак не скажется на их работоспособности,   
при условии, что они явно не проверяют числа на принадлежность к типу   
длинных целых, имеющемуся в версии 2.6.  
Шестнадцатеричные, восьмеричные и двоичные литералы  
Целые числа могут записываться как десятичные (по основанию 10), шест-  
надцатеричные (по основанию 16), восьмеричные (по основанию 8) и двоич-  
ные (по основанию 2). Шестнадцатеричные литералы начинаются с комби-  
нации символов 0x или 0X, вслед за которыми следуют шестнадцатеричные   
цифры (0-9 и A-F). Шестнадцатеричные цифры могут вводиться как в ниж-  
нем, так и в верхнем регистре. Литералы восьмеричных чисел начинают-  
ся с комбинации символов 0o или 0O (ноль и следующий за ним символ «o»   
в верхнем или нижнем регистре), вслед за которыми следуют восьмеричные   
цифры (0-7). В Python 2.6 и в более ранних версиях восьмеричные литералы   
могут начинаться только с символа 0 (без буквы «o»), но в версии 3.0 такая   
форма записи считается недопустимой (при использовании прежней формы   
записи восьмеричные числа легко можно перепутать с десятичными, поэто-  
му была принята новая форма записи, когда восьмеричные литералы начи-  
наются с комбинации символов 0o). Двоичные литералы впервые появились   
в версиях 2.6 и 3.0, они начинаются с комбинации символов 0b или 0B, вслед   
за которыми следуют двоичные цифры (0 – 1)  
Примечательно, что все эти литералы создают объекты целых чисел – они   
являются всего лишь альтернативными формами записи значений. Для   
преобразования целого числа в строку с представлением в любой из трех   
систем счисления можно использовать встроенные функции hex(I), oct(I)   
и bin(I), кроме того, с помощью функции int(str, base) можно преобразовать   
строку в целое число с учетом указанного основания системы счисления.  
Комплексные числа  
Литералы комплексных чисел в языке Python записываются в формате   
действительная\_часть+мнимая\_часть, где мнимая\_часть завершается символом j

156   
Глава 5. Числа   
или J. С технической точки зрения, действительная\_часть является необяза-  
тельной, поэтому мнимая\_часть может указываться без действительной со-  
ставляющей. Во внутреннем представлении комплексное число реализо-  
вано в виде двух вещественных чисел, но при работе с числами этого типа   
используется математика комплексных чисел. Комплексные числа могут   
также создаваться с помощью встроенной функции complex(real, imag).  
Литералы других числовых типов  
Как мы увидим ниже в этой главе, существуют и другие числовые типы,   
не включенные в табл. 5.1. Объекты некоторых из этих типов создаются   
с помощью функций, объявленных в импортируемых модулях (например,   
вещественные числа с фиксированной точностью и рациональные числа),   
другие имеют свой синтаксис литералов (например, множества).  
Встроенные числовые операции и расширения  
Помимо числовых литералов, которые приводятся в табл. 5.1, язык Python   
предоставляет набор операций для работы с числовыми объектами:  
Операторы выражений  
+, -, \*, /, >>, \*\*, & и другие.  
Встроенные математические функции  
pow, abs , round, int, hex, bin и другие.  
Вспомогательные модули  
random, math и другие.  
По мере продвижения мы встретимся со всеми этими компонентами.  
Для работы с числами в основном используются выражения, встроенные   
функции и модули, при этом числа имеют ряд собственных, специфических   
методов, с которыми мы также встретимся в этой главе. Вещественные чис-  
ла, например, имеют метод as\_integer\_ratio, который удобно использовать для   
преобразования вещественного числа в рациональное, а также метод is\_integer   
method, который проверяет – можно ли представить вещественное число как це-  
лое значение. Целые числа тоже обладают различными атрибутами, включая   
новый метод bit\_length, который появился в версии Python 3.1, – он возвраща-  
Python 3.1, – он возвраща-  
 3.1, – он возвраща-  
ет количество битов, необходимых для представления значения числа. Кроме   
того, множества, отчасти являясь коллекциями, а отчасти числами, поддер-  
живают собственные методы и операторы выражений.  
Так как выражения наиболее часто используются при работе с числами, мы   
рассмотрим их в первую очередь.  
Операторы выражений  
Пожалуй, самой фундаментальной возможностью обработки чисел являются   
выражения: комбинации чисел (или других объектов) и операторов, которые   
возвращают значения при выполнении интерпретатором Python. Выражения   
в языке Python записываются с использованием обычной математической но-  
тации и символов операторов. Например, сложение двух чисел X и Y записы-  
вается в виде выражения X + Y, которое предписывает интерпретатору Python   
применить оператор + к значениям с именами X и Y. Результатом выражения   
X + Y будет другой числовой объект.

Базовые числовые типы   
157  
В табл. 5.2 приводится перечень всех операторов, имеющихся в языке Python.   
Многие из них достаточно понятны сами по себе, например обычные матема-  
тические операторы (+, -, \*, / и так далее). Некоторые будут знакомы тем, кто   
в прошлом использовал другие языки программирования: оператор % вычисля-  
ет остаток от деления, оператор << производит побитовый сдвиг влево, оператор   
& выполняет побитовую операцию И и т. д. Другие операторы более характерны   
для языка Python, и не все имеют математическую природу. Например, опера-  
тор is проверяет идентичность объектов (это более строгая форма проверки на   
равенство), оператор lambda создает неименованные функции и так далее.   
Таблица 5.2. Операторы выражений в языке Python и правила определения   
старшинства  
Операторы  
Описание  
yield x  
Поддержка протокола send в функциях-генераторах  
lambda args: expression  
Создает анонимную функцию  
x if y else z  
Трехместный оператор выбора (значение x вычисля-  
ется, только если значение y истинно)  
x or y  
Логическая операция ИЛИ (значение y вычисляет-  
ся, только если значение x ложно)  
x and y  
Логический оператор И (значение y вычисляется,   
только если значение x истинно)  
not x  
Логическое отрицание  
x in y, x not in y  
Проверка на вхождение (для итерируемых объектов   
и множеств)  
x is y, x is not y  
Проверка идентичности объектов  
x < y, x <= y, x > y,   
x >= y  
x == y, x != y  
Операторы сравнения, проверка на подмножество   
и надмножество  
Операторы проверки на равенство  
x | y  
Битовая операция ИЛИ, объединение множеств  
x ^ y  
Битовая операция «исключающее ИЛИ» (XOR),   
симметрическая разность множеств  
x & y  
Битовая операция И, пересечение множеств  
x << y, x >> y  
Сдвиг значения x влево или вправо на y битов  
x, + y   
x – y  
Сложение, конкатенация   
Вычитание, разность множеств  
x \* y   
x % y   
x / y, x // y  
Умножение, повторение   
Остаток, формат   
Деление: истинное и с округлением вниз  
-x, +x  
Унарный знак «минус», тождественность  
~x  
Битовая операция НЕ (инверсия)

158   
Глава 5. Числа   
Операторы  
Описание  
x \*\* y  
Возведение в степень  
x[i]  
Индексация (в последовательностях, отображениях   
и других объектах)  
x[i:j:k]  
Извлечение среза  
x(...)  
Вызов (функций, классов и других вызываемых   
объектов)  
x.attr  
Обращение к атрибуту  
(...)  
Кортеж, подвыражение, выражение-генератор  
[...]  
Список, генератор списков  
{...}  
Словарь, множество, генератор словарей и мно-  
жеств  
Так как в этой книге описываются обе версии Python 2.6 и 3.0, ниже приводят-  
ся некоторые примечания, касающиеся различий между версиями и послед-  
них дополнений, имеющих отношение к операторам из табл. 5.2:  
 •  
В версии Python 2.6 неравенство значений можно проверить двумя спосо-  
бами, как X != Y или как X <> Y. В Python 3.0 последний вариант был убран.   
В обеих версиях рекомендуется использовать выражение X != Y для провер-  
ки на неравенство.  
 •  
В версии Python 2.6 выражение в обратных кавычках `X` действует точно   
так же, как вызов функции repr(X), и преобразует объект в строковое пред-  
ставление. Из-за неочевидности функционального назначения это выраже-  
ние было убрано из Python 3.0 – используйте более очевидные встроенные   
функции str и repr, которые описываются в разделе «Форматы отображе-  
ния чисел».  
 •  
Операция деления с округлением вниз (X // Y) всегда усекает дробную часть   
в обеих версиях Python 2.6 и 3.0. Операция деления X / Y в версии 3.0 вы-  
полняет истинное деление (возвращает результат с дробной частью), а в вер-  
сии 2.6 – классическое деление (усекает результат до целого числа). Подроб-  
ности ищите в разделе «Деление: классическое, с округлением вниз и ис-  
тинное».  
 •  
Синтаксическая конструкция [...] используется для определения литера-  
лов и выражений-генераторов списков. В последнем случае предполагается   
выполнение цикла и накопление результатов в новом списке. Примеры ис-  
пользования этой конструкции приводятся в главах 4, 14 и 20.  
 •  
Синтаксическая конструкция (...) используется для определения корте-  
жей, подвыражений и выражений-генераторов – разновидности генерато-  
ров списков, которая воспроизводит значения по требованию. Примеры ис-  
пользования этой конструкции приводятся в главах 4 и 20. Во всех трех   
случаях круглые скобки могут опускаться.  
 •  
Синтаксическая конструкция {...} используется для определения литера-  
лов словарей, а в версии Python 3.0 еще и для определения литералов мно-  
Таблица 5.2 (продолжение)

Базовые числовые типы   
159  
жеств, и генераторов словарей и множеств. Описание этой конструкции   
приводится в этой главе, а примеры использования – в главах 4, 8, 14 и 20.  
 •  
Инструкция yield и трехместный оператор выбора if/else доступны в вер-  
сии Python 2.5 и выше. Инструкция yield в функциях-генераторах возвра-  
щает аргументы функции send(...)� оператор выбора if/else является крат-  
кой формой записи многострочной инструкции if. Если инструкция yield –   
не единственное, что находится справа от оператора присваиваивания, она   
должна заключаться в круглые скобки.  
 •  
Операторы отношений могут объединяться в цепочки: например X < Y < Z   
воспроизводит тот же результат, что и конструкция X < Y and Y < Z. Подроб-  
ности приводятся в разделе «Операции сравнения: простые и составные»,   
ниже.  
 •  
В последних версиях Python выражение извлечения среза X[I:J:K] являет-  
ся эквивалентом операции индексирования с применением объекта среза:   
X[slice(I, J, K)].  
 •  
В Python 2.X при выполнении операций сравнения числовые значения при-  
Python 2.X при выполнении операций сравнения числовые значения при-  
 2.X при выполнении операций сравнения числовые значения при-  
X при выполнении операций сравнения числовые значения при-  
 при выполнении операций сравнения числовые значения при-  
водятся к общему типу, а упорядочение несовместимых типов выполняется   
посредством сравнивания имен типов значений и считается допустимым.   
В Python 3.0 не допускается сравнивать несовместимые типы, не являющи-  
Python 3.0 не допускается сравнивать несовместимые типы, не являющи-  
 3.0 не допускается сравнивать несовместимые типы, не являющи-  
еся числами, а попытка такого сравнения вызывает исключение.  
 •  
Операторы отношений для словарей также больше не поддерживаются   
в Python 3.0 (однако поддерживается операция проверки на равенство).   
Один из возможных способов сравнивания словарей заключается в исполь-  
зовании функции sorted(dict.items()).  
Примеры использования большинства операторов из табл. 5.2 будут показаны   
позже, а пока посмотрим, как операторы могут объединяться в выражения.  
Смешанные операторы   
и определение старшинства операторов  
Как и в большинстве других языков программирования, в языке Python можно   
создавать сложные выражения, объединяя несколько операторов из табл. 5.2   
в одной инструкции. Например, вычисление суммы двух произведений можно   
записать следующим образом:  
A \* B + C \* D  
Как в этом случае интерпретатор узнает, какие операторы должны выполнять-  
ся в первую очередь? Ответ на этот вопрос заключается в старшинстве опера-  
торов. Когда интерпретатор Python встречает выражение, содержащее более   
одного оператора, он делит его на отдельные части в соответствии с правила-  
ми старшинства и определяет порядок вычисления этих частей выражения.   
В табл. 5.2 операторы расположены в порядке возрастания старшинства:  
 •  
Чем выше приоритет оператора, тем ниже он находится в таблице и тем   
раньше он выполняется в смешанных выражениях.   
 •  
Если в выражении имеется несколько операторов, находящихся в табл. 5.2   
в одной строке, они выполняются в направлении слева направо (исключе-  
ние составляет оператор возведения в степень – эти операторы выполня-  
ются справа налево, и операторы отношений, которые объединяются в на-  
правлении слева направо).

160   
Глава 5. Числа   
Например, если вы запишете выражение X + Y \* Z, интерпретатор Python сна-  
чала выполнит умножение (Y \* Z), а затем прибавит результат к значению X,   
потому что оператор \* имеет более высокий приоритет (в табл. 5.2 он находится   
ниже), чем оператор +. Точно так же в первом примере этого раздела сначала   
будут найдены произведения (A \* B и C \* D), а затем будет выполнено сложение.  
Группировка подвыражений с помощью круглых скобок  
Вы можете навсегда забыть о старшинстве операторов, если будете группиро-  
вать части выражений с помощью круглых скобок. Когда часть выражения за-  
ключается в круглые скобки, они отменяют правила старшинства операторов –   
Python всегда в первую очередь вычисляет подвыражения в круглых скобках,   
а затем использует их результаты в объемлющем выражении.  
Например, выражение X + Y \* Z можно записать одним из следующих спосо-  
бов, чтобы вынудить Python произвести вычисления в требуемом порядке:  
(X + Y) \* Z  
X + (Y \* Z)  
В первом случае сначала будет выполнено сложение значений X и Y, потому что   
это подвыражение заключено в круглые скобки. Во втором случае первой будет   
выполнена операция умножения (точно так же, как если бы скобки вообще от-  
сутствовали). Вообще говоря, использование круглых скобок в сложных вы-  
ражениях можно только приветствовать – они не только определяют порядок   
выполнения вычислений, но и повышают удобочитаемость.  
Смешивание типов и их преобразование  
Помимо смешивания операторов вы можете также смешивать различные чис-  
ловые типы. Допустим, что требуется найти сумму целого и вещественного   
числа:  
40 + 3.14  
Но это влечет за собой другой вопрос: какого типа будет результат – целое чис-  
ло или вещественное число? Ответ прост, особенно для тех, кто уже имеет опыт   
работы с любыми другими языками программирования: в выражениях, где   
участвуют значения различных типов, интерпретатор сначала выполняет пре-  
образование типов операндов к типу самого сложного операнда, а потом при-  
меняет математику, специфичную для этого типа. Если вам уже приходилось   
использовать язык C, вы найдете, что такое поведение соответствует порядку   
преобразования типов в этом языке.  
Интерпретатор Python ранжирует сложность числовых типов следующим об-  
разом: целые числа проще, чем вещественные числа, которые проще комплекс-  
ных чисел. Поэтому, когда в выражении участвуют целое число и веществен-  
ное число, как в предыдущем примере, то целое число сначала будет преобра-  
зовано в вещественное число, а затем будет выполнена операция из математики   
вещественных чисел, что дает в результате вещественное число. Точно так же,   
когда один из операндов в выражении является комплексным числом, другой   
операнд будет преобразован в комплексное число и выражение вернет в резуль-  
тате также комплексное число. (Кроме того, в Python 2.6 обычные целые числа   
преобразуются в длинные целые, если значение не может быть представлено   
в виде обычного целого числа� в версии 3.0 все целые числа попадают в катего-  
рию длинных целых.)

Базовые числовые типы   
161  
Существует возможность принудительного преобразования типов с помощью   
встроенных функций:  
>>> int(3.1415) # Усекает дробную часть вещественного числа  
3  
>>> float(3) # Преобразует целое число в вещественное  
3.0  
Однако в обычных ситуациях делать это не приходится, потому что Python ав-  
томатически выполняет преобразование типов и тип результата, как правило,   
соответствует вашим ожиданиям.  
Кроме того, имейте в виду, что все эти преобразования производятся только   
при смешивании числовых типов (то есть целых и вещественных чисел) в вы-  
ражении, включая выражения, выполняющие математические операции   
и операции сравнения. Вообще, Python не выполняет автоматическое преоб-  
Python не выполняет автоматическое преоб-  
 не выполняет автоматическое преоб-  
разование других типов. Например, попытка выполнить операцию сложения   
строки и числа приведет к появлению ошибки, если вы вручную не выполните   
преобразование типа одного из операндов, – примеры таких преобразований   
встретятся вам в главе 7, когда мы будем обсуждать строки.  
В Python 2.6 допускается сравнивание разнотипных нечисло-  
вых значений, но при этом преобразование типов операндов не   
выполняется (сравнивание разнотипных значений выполняется   
в соответствии с фиксированным, но достаточно произвольным   
правилом). В версии 3.0 сравнивание разнотипных нечисловых   
значений не допускается и приводит к исключению.  
Обзор: перегрузка операторов и полиморфизм  
Несмотря на то, что сейчас в центре нашего внимания находятся встроенные   
числа, вы должны знать, что в языке Python существует возможность выпол-  
нить (то есть реализовать) перегрузку любого оператора с помощью классов   
Python или расширений на языке C для работы с создаваемыми объектами.   
Например, как будет показано позже, объекты, реализованные в виде классов,   
могут участвовать в операции сложения, индексироваться с помощью выраже-  
ния [i] и так далее.  
Кроме того, Python сам автоматически перегружает некоторые операторы, что-  
бы с их помощью можно было выполнять различные действия, в зависимости   
от типа встроенных объектов. Например, оператор + выполняет операцию сло-  
жения, когда применяется к числам, но когда он применяется к последователь-  
ностям, таким как строки или списки, он выполняет операцию конкатенации.   
В действительности оператор + может выполнять любые действия, когда при-  
меняется к объектам, которые вы определяете с помощью классов.  
Как было показано в предыдущей главе, эта особенность обычно называется   
полиморфизмом – этот термин означает, что выполняемая операция зависит от   
типов объектов-операндов, над которыми она выполняется. Мы еще вернемся   
к этому понятию в главе 16, когда будем рассматривать функции, потому что   
в этом контексте суть полиморфизма становится более очевидной.

162   
Глава 5. Числа   
Числа в действии  
Приступим к программированию! Самый лучший способ понять числа и вы-  
ражения состоит в том, чтобы увидеть их в действии, поэтому давайте запу-  
стим интерактивный сеанс работы с интерпретатором и попробуем выполнить   
некоторые простые, но весьма показательные операции (если вы забыли, как   
запускается интерактивный сеанс, обращайтесь к главе 3).  
Переменные и простые выражения  
Прежде всего, рассмотрим основные арифметические операции. Для начала   
присвоим двум переменным (a и b) целочисленные значения, чтобы потом ис-  
пользовать их в более сложных выражениях. Переменные – это всего лишь   
имена, создаваемые в языке Python, которые используются для обозначения   
информации в программах. Об этом мы будем говорить более подробно в сле-  
дующей главе, а пока вы должны знать, что в языке Python:  
 •  
Переменные создаются с помощью операции присваивания.  
 •  
При вычислении выражений имена переменных замещаются их значения-  
ми.  
 •  
Прежде чем переменная сможет участвовать в выражениях, ей должно   
быть присвоено значение.  
 •  
Переменные являются ссылками на объекты и никогда не объявляются за-  
ранее.  
Другими словами, следующие операции присваивания автоматически приво-  
дят к созданию переменных a и b:  
% python  
>>> a = 3 # Создается имя  
>>> b = 4  
Здесь я также использовал комментарий. Вспомните, что в программном коде   
на языке Python текст, следующий за символом #, считается комментарием   
и игнорируется интерпретатором. Комментарии – это один из способов описать   
программный код на удобном для восприятия языке. Поскольку программный   
код, который пишется в ходе интерактивного сеанса, является временным,   
вам не требуется писать комментарии, я же буду добавлять их в примеры, что-  
бы объяснять работу программного кода.1 В следующей части книги мы позна-  
комимся с еще одной похожей особенностью – со строками документирования,   
которые включают текст комментариев в объекты.  
А теперь попробуем использовать наши первые целочисленные объекты в вы-  
ражениях. В настоящий момент переменные a и b все еще имеют значения 3 и 4   
соответственно. Когда переменные, подобные этим, участвуют в выражении,   
они замещаются их значениями, а при работе в интерактивном режиме резуль-  
тат вычисления выражения тут же выводится на экран:  
1   
Если вы пробуете примеры на практике, вам не нужно вводить текст, начинающийся   
с символа # и продолжающийся до конца строки� комментарии просто игнорируются   
интерпретатором и не являются необходимой частью инструкций, которые мы вы-  
полняем.

Числа в действии   
163  
>>> a + 1, a - 1 # Сложение (3 + 1), вычитание (3 - 1)  
(4, 2)  
>>> b \* 3, b / 2 # Умножение (4 \* 3), деление (4 / 2)  
(12, 2.0)  
>>> a % 2, b \*\* 2 # Деление по модулю (остаток), возведение в степень  
(1, 16)  
>>> 2 + 4.0, 2.0 \*\* b # Смешивание типов, выполняется преобразование  
(6.0, 16.0)  
С технической точки зрения результатами этих инструкций являются кортежи,   
состоящие из двух значений, потому что вводимые строки содержат по два выра-  
жения, разделенные запятыми. Именно по этой причине результаты отобража-  
ются в круглых скобках (подробнее о кортежах будет рассказываться позднее).   
Следует заметить, что эти выражения выполняются без ошибок потому, что ра-  
нее переменным a и b были присвоены значения. Если использовать переменную,   
которой еще не было присвоено значение, Python выведет сообщение об ошибке:  
>>> c \* 2  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in ?  
NameError: name ‘c’ is not defined  
В языке Python от вас не требуется заранее объявлять переменные, но прежде   
чем их можно будет использовать, им должны быть присвоены некоторые зна-  
чения. На практике это означает, что перед тем как к счетчикам можно будет   
прибавлять некоторые значения, их необходимо инициализировать нулевым   
значением� прежде чем к спискам можно будет добавлять новые элементы, их   
необходимо инициализировать пустыми списками, и так далее.  
Ниже приводятся два более сложных выражения, чтобы проиллюстрировать   
порядок выполнения операторов и производимые преобразования:  
>>> b / 2 + a # То же, что и ((4 / 2) + 3)  
5.0  
>>> print(b / (2.0 + a)) # То же, что и (4 / (2.0 + 3))  
0.8  
В первом выражении отсутствуют круглые скобки, поэтому интерпретатор   
Python автоматически группирует компоненты выражения в соответствии   
с правилами определения старшинства – оператор / находится ниже в табл. 5.2,   
чем оператор +, поэтому его приоритет считается выше и он выполняется пер-  
вым. Результат вычисляется так, как если бы выражение включало скобки,   
как показано в комментарии.   
Кроме того, обратите внимание, что в первом выражении все числа являются   
целыми, поэтому в версии 2.6 будут выполнены целочисленные операции деле-  
ния и сложения, что даст в результате значение 5. В Python 3.0 будет выполне-  
Python 3.0 будет выполне-  
 3.0 будет выполне-  
на операция истинного деления и получен результат, как показано в примере.   
Если в версии 3.0 потребуется выполнить целочисленное деление, выражение   
можно записать так: b // 2 + a (подробнее об операции деления рассказывается   
чуть ниже).  
Во втором выражении круглые скобки окружают операцию сложения, чтобы   
вынудить интерпретатор выполнить ее в первую очередь (то есть до операции /).   
Кроме того, один из операндов является вещественным числом, т. к. в нем при-  
сутствует десятичная точка: 2.0. Вследствие такого смешения типов перед вы-  
полнением операции сложения Python преобразует целое число, на которое ссы-

164   
Глава 5. Числа   
лается имя a, в вещественное число (3.0). Если бы все числа в этом выражении   
были целыми, то операция целочисленного деления (4 / 5) в Python 2.6 вернула   
бы число 0, но в Python 3.0 она возвращает вещественное число 0.8 (подробнее об   
операции деления рассказывается чуть ниже).  
Форматы отображения чисел  
Обратите внимание: в последнем примере была использована инструкция   
print. Без этой инструкции результат мог бы показаться немного странным:  
>>> b / (2.0 + a) # Автоматический вывод: выводится большее число цифр  
0.80000000000000004  
   
>>> print(b / (2.0 + a)) # Инструкция print отбрасывает лишние цифры  
0.8  
Причина появления такого немного странного результата кроется в ограниче-  
ниях аппаратных средств, реализующих вещественную математику, и в невоз-  
можности обеспечить точное представление некоторых значений. Обсуждение   
аппаратной архитектуры компьютера выходит далеко за рамки этой книги,   
тем не менее, я замечу, что все цифры в первом результате действительно при-  
сутствуют в аппаратной части компьютера, выполняющей операции над чис-  
лами с плавающей точкой, просто вы не привыкли видеть их. Я использовал   
этот пример, чтобы продемонстрировать различия в форматах отображения   
чисел – при автоматическом выводе результатов в ходе интерактивного сеанса   
отображается больше цифр, чем при использовании инструкции print. Если вы   
не желаете, чтобы выводились лишние цифры, используйте инструкцию print.   
Во врезке «Форматы представления repr и str» на стр. 9 рассказывается, как   
добиться более дружественного формата отображения результатов.  
Однако надо заметить, что не всегда значения отображаются с таким большим   
числом цифр:  
>>> 1 / 2.0  
0.5  
и что кроме применения функции print и автоматического вывода результатов   
существуют и другие способы отображения чисел:  
>>> num = 1 / 3.0  
>>> num # Автоматический вывод  
0.33333333333333331  
>>> print num # Инструкция print выполняет округление  
0.333333333333  
   
>>> “%e” % num # Вывод с использованием выражения форматирования строк  
‘3.333333e-001’  
>>> ‘%4.2f’ % num # Альтернативный формат представления вещественных чисел  
‘0.33’  
>>> ‘{0:4.2f}’.format(num) # Метод форматирования строк (Python 2.6 и 3.0)  
‘0.33’   
В последних трех случаях была использована операция форматирования   
строк – инструмент, обладающий гибкой возможностью определять формат   
представления, но об этом мы поговорим в главе 7, когда займемся исследова-  
нием строк. Результатом этой операции обычно являются строки, предназна-  
ченные для вывода.

Числа в действии   
165  
Форматы представления repr и str  
C технической точки зрения различие между функцией автоматическо-  
го вывода в интерактивной оболочке и инструкцией print соответствует   
различию между встроенными функциями repr и str:  
>>> num = 1 / 3  
>>> repr(num)# Используется для автоматического вывода: в форме как есть  
‘0.33333333333333331’  
>>> str(num) # Используется функцией print: дружественная форма  
‘0.333333333333’  
Обе функции преобразуют произвольные объекты в их строковое пред-  
ставление: repr (и функция автоматического вывода в интерактивной   
оболочке) выводит результаты в том виде, в каком они были бы указаны   
в программном коде� str (и операция print) обычно выполняет преобра-  
зование значения в более дружественное представление. Некоторые объ-  
екты имеют оба варианта строкового представления: str – для использо-  
вания в обычных случаях и repr – для вывода в расширенном варианте.   
Эта идея еще всплывет далее, при обсуждении строк и возможности пе-  
регрузки операторов в классах, и тогда вы получите более полное пред-  
ставление об этих встроенных функциях.  
Помимо операции получения строкового представления произвольных   
объектов имя str так же является именем типа строковых данных и мо-  
жет вызываться с названием кодировки в качестве аргумента для преоб-  
разования строк байтов в строки Юникода. Эту дополнительную особен-  
ность мы рассмотрим в главе 36.  
Операции сравнения: простые и составные  
До сих пор мы имели дело со стандартными числовыми операциями (такими   
как сложение или умножение), но, кроме того, числа можно сравнивать. Обыч-  
ные операторы сравнения действуют именно так, как и можно было бы ожи-  
дать, – они сравнивают значения операндов и возвращают логический резуль-  
тат (который, как правило, проверяется объемлющей инструкцией):  
>>> 1 < 2 # Меньше чем  
True  
>>> 2.0 >= 1 # Больше или равно: число 1 преобразуется 1.0  
True  
>>> 2.0 == 2.0 # Проверка на равенство значений  
True  
>>> 2.0 != 2.0 # Проверка на неравенство значений  
False  
Обратите внимание, смешивание разнотипных операндов допускается, только   
если оба они принадлежат к числовым типам. Во второй инструкции, в при-  
мере выше, интерпретатор выполняет сравнивание с позиции более сложного   
типа, преобразуя целое число в вещественное.

166   
Глава 5. Числа   
Самое интересное, что Python позволяет составлять цепочки из нескольких   
операторов сравнения, для выполнения проверки на принадлежность диапазо-  
ну значений. Цепочка операторов сравнения является, своего рода, сокращен-  
ной формой записи более длинных логических выражений. Проще говоря, Py-  
Py-  
thon позволяет объединить несколько операций сравнения, чтобы реализовать   
проверку на вхождение в диапазон значений. Выражение (A < B < C), например,   
проверяет, входит ли значение B в диапазон от A до C, и является эквивалентом   
логическому выражению (A < B and B < C), но выглядит гораздо понятнее (и ко-  
роче). Например, допустим, что в программе имеются следующие инструкции   
присваивания:  
>>> X = 2  
>>> Y = 4  
>>> Z = 6  
Следующие два выражения дают совершенно идентичный результат, но первое   
из них короче и, вполне возможно, выполняется немного быстрее, потому что   
интерпретатору приходится вычислять значение Y только один раз:  
>>> X < Y < Z # Составная операция сравнения: принадлежность диапазону  
True  
>>> X < Y and Y < Z  
True  
То же самое относится и к выражениям с ложным результатом� кроме того, до-  
пускается составлять цепочки произвольной длины:  
>>> X < Y > Z  
False  
>>> X < Y and Y > Z  
False  
>>> 1 < 2 < 3.0 < 4  
True  
>>> 1 > 2 > 3.0 > 4  
False  
В цепочках сравнений можно использовать и другие операторы сравнения.   
При этом получающиеся результаты могут на первый взгляд не иметь смысла,   
если не попытаться вычислить выражение тем же способом, как это делает ин-  
терпретатор Python. Например, следующее выражение ложно уже потому, что   
число 1 не равно числу 2:  
>>> 1 == 2 < 3 # То же, что и: 1 == 2 and 2 < 3  
False # Но не то же самое, что и: False < 3 (что означает утверждение   
 # 0 < 3, которое истинно)  
Интерпретатор не сравнивает значение False (результат операции 1 == 2) с чис-  
лом 3 – с технической точки зрения это соответствовало бы выражению 0 < 3,   
которое должно было бы вернуть True (как мы увидим ниже, в этой же главе,   
True и False – это всего лишь числа 1 и 0, расширенные приписанными им свой-  
ствами).  
Деление: классическое, с округлением вниз и истинное  
В предыдущих разделах вы уже видели, как работает операция деления, и по-  
этому должны помнить, что в Python 2.6 и 3.0 она действует по-разному. Фак-  
Python 2.6 и 3.0 она действует по-разному. Фак-  
 2.6 и 3.0 она действует по-разному. Фак-

Числа в действии   
167  
тически существует три версии операции деления и два различных оператора   
деления, поведение одного из которых изменилось в версии 3.0:  
X / Y  
Классическое и истинное деление. В Python 2.6 и в более ранних версиях   
этот оператор выполняет операцию классического деления, когда дробная   
часть результата усекается при делении целых чисел и сохраняется при   
делении вещественных чисел. В Python 3.0 этот оператор выполняет опе-  
Python 3.0 этот оператор выполняет опе-  
 3.0 этот оператор выполняет опе-  
рацию истинного деления, которая всегда сохраняет дробную часть неза-  
висимо от типов операндов.  
X // Y  
Деление с округлением вниз. Этот оператор впервые появился в Python 2.2   
и доступен в обеих версиях Python, 2.6 и 3.0. Он всегда отсекает дробную   
часть, округляя результат до ближайшего наименьшего целого независимо   
от типов операндов.  
Истинное деление было добавлено по той причине, что в текущей модели клас-  
сического деления результаты зависят от типов операндов и с трудом могут   
быть оценены заранее в языке программирования с динамической типизаци-  
ей, каким является Python. Операция классического деления была ликвиди-  
рована в Python 3.0 из-за этого ограничения – в версии 3.0 операторы / и // реа-  
лизуют истинное деление и деление с округлением вниз.  
Подводя итоги:  
 •  
В версии 3.0 оператор / всегда выполняет операцию истинного деления,   
возвращает вещественный результат, включающий дробную часть, неза-  
висимо от типов операндов. Оператор // выполняет деление с округлением   
вниз, усекая дробную часть и возвращая целочисленный результат, если   
оба операнда являются целыми числами, и вещественный – если хотя бы   
один операнд является вещественным числом.  
 •  
В версии 2.6 оператор / выполняет операцию классического деления, произ-  
водя целочисленное деление с усечением дробной части, если оба операнда   
являются целыми числами, и вещественное деление (с сохранением дроб-  
ной части) – в противном случае. Оператор // выполняет деление с окру-  
глением вниз и действует так же, как и в версии 3.0, выполняя деление   
с усечением дробной части, если оба операнда являются целыми числами,   
и деление с округлением вниз, если хотя бы один из операндов является ве-  
щественным числом.  
Ниже эти два оператора демонстрируются в действии в версиях 3.0 и 2.6:  
C:\misc> C:\Python30\python  
>>>  
>>> 10 / 4 # Изменился в версии 3.0: сохраняет дробную часть  
2.5  
>>> 10 // 4 # В версии 3.0 действует так же: усекает дробную часть  
2  
>>> 10 / 4.0 # В версии 3.0 действует так же: сохраняет дробную часть  
2.5  
>>> 10 // 4.0 # В версии 3.0 действует так же: округляет вниз  
2.0  
   
C:\misc> C:\Python26\python  
>>>

168   
Глава 5. Числа   
>>> 10 / 4  
2  
>>> 10 // 4  
2  
>>> 10 / 4.0  
2.5  
>>> 10 // 4.0  
2.0  
Обратите внимание, что в версии 3.0 тип результата операции // по-прежнему   
зависит от типов операндов: если хотя бы один из операндов является веще-  
ственным числом, результат также будет вещественным числом� в противном   
случае результат будет целым числом. Несмотря на то, что это может пока-  
заться похожим на поведение оператора /, зависящее от типа операндов, в вер-  
сиях 2.X, которое послужило побудительной причиной его изменения в вер-  
X, которое послужило побудительной причиной его изменения в вер-  
, которое послужило побудительной причиной его изменения в вер-  
сии 3.0, нужно признать, что тип возвращаемого значения является гораздо   
менее существенным – в сравнении с разницей в самих возвращаемых значе-  
ниях. Кроме того, оператор // был сохранен с целью обеспечения обратной со-  
вместимости для программ, которые используют операцию целочисленного   
деления с усечением (она используется намного чаще, чем можно было бы по-  
думать), которая должна возвращать целочисленный результат, когда оба опе-  
ранда являются целыми числами.  
Поддержка обеих версий Python  
Хотя поведение оператора / различно в версиях 2.6 и 3.0, тем не менее, суще-  
ствует возможность обеспечить поддержку обеих версий в своем программном   
коде. Если в программе требуется выполнить операцию целочисленного деле-  
ния с усечением, используйте оператор // в обеих версиях, 2.6 и 3.0. Если про-  
грамме требуется получить вещественный результат при делении целых чисел   
без усечения дробной части используйте в операторе / функцию float, чтобы   
гарантировать, что один из операндов всегда будет вещественным числом, при   
использовании версии 2.6:  
X = Y // Z # Всегда усекает, для целочисленных операндов всегда возвращает  
 # целочисленный результат в обеих версиях, 2.6 и 3.0  
X = Y / float(Z) # Гарантирует вещественное деление с сохранением дробной   
 # части в обеих версиях, 2.6 и 3.0  
Кроме того, в версии 2.6 режим истинного деления, который используется   
в версии 3.0, можно включить, если вместо использования преобразования   
float импортировать модуль \_\_future\_\_:  
C:\misc> C:\Python26\python  
>>> from \_\_future\_\_ import division # Включает поведение “/” как версии 3.0  
>>> 10 / 4  
2.5  
>>> 10 // 4  
2  
Округление вниз и усечение дробной части  
Существует один очень тонкий момент: оператор // обычно называют операто-  
ром деления с усечением, но более точно было бы называть его оператором деле-  
ния с округлением результата вниз – он округляет результат до ближайшего

Числа в действии   
169  
меньшего целого значения, то есть до целого числа, расположенного ниже ис-  
тинного результата. Округление вниз – это совсем не то же самое, что усечение   
дробной части, – это обстоятельство приобретает значение при работе с отри-  
цательными числами. Разницу можно наглядно продемонстрировать, обратив-  
шись к модулю math (прежде чем использовать модуль, его необходимо импор-  
тировать, но подробнее об этом мы поговорим позже):  
>>> import math  
>>> math.floor(2.5)  
2  
>>> math.floor(-2.5)  
-3  
>>> math.trunc(2.5)  
2  
>>> math.trunc(-2.5)  
-2  
При выполнении операции деления в действительности усечение выполняется   
только в случае положительного результата, поскольку для положительных   
чисел усечение и округление вниз – суть одно и то же� в случае отрицательного   
результата выполняется округление вниз (в действительности и в том и в дру-  
гом случае выполняется округление вниз, но в случае положительных чисел   
округление вниз дает тот же эффект, что и усечение дробной части). Ниже при-  
водятся примеры деления в версии 3.0:  
C:\misc> c:\python30\python  
>>> 5 / 2, 5 / -2  
(2.5, -2.5)  
>>> 5 // 2, 5 // -2 # Округление вниз: результат 2.5 округляется до 2,  
(2, -3) # а -2.5 округляется до -3  
>>> 5 / 2.0, 5 / -2.0  
(2.5, -2.5)  
>>> 5 // 2.0, 5 // -2.0 # То же относится и к вещественному делению,  
(2.0, -3.0) # только результат имеет вещественный тип  
Деление в версии 2.6 выполняется точно так же, только результат действия   
оператора / отличается:  
C:\misc> c:\python26\python  
>>> 5 / 2, 5 / -2 # Отличается от результата в версии 3.0  
(2, -3)  
>>> 5 // 2, 5 // -2 # Эта и следующие операции дают одинаковый результат   
(2, -3) # в обеих версиях, 2.6 и 3.0  
>>> 5 / 2.0, 5 / -2.0  
(2.5, -2.5)  
>>> 5 // 2.0, 5 // -2.0  
(2.0, -3.0)  
Если требуется реализовать усечение дробной части независимо от знака, ре-  
зультат вещественного деления всегда можно передать функции math.trunc не-  
зависимо от используемой версии Python (кроме того, взгляните на функцию   
round, которая обладает сходной функциональностью):  
C:\misc> c:\python30\python  
>>> import math  
>>> 5 / -2 # Сохранит дробную часть  
-2.5

170   
Глава 5. Числа   
>>> 5 // -2 # Округлит результат вниз  
-3  
>>> math.trunc(5 / -2) # Вместо округления будет выполнено усечение  
-2  
   
C:\misc> c:\python26\python  
>>> import math  
>>> 5 / float(-2) # Сохранит дробную часть в версии 2.6  
-2.5  
>>> 5 / -2, 5 // -2 # Округлит результат вниз в версии 2.6  
(-3, -3)  
>>> math.trunc(5 / float(-2)) # Вместо округления будет выполнено   
-2 # усечение в версии 2.6  
Почему усечение так важно?  
Для тех, кто использует Python 3.0, ниже приводится пример, позволяющий   
понять принцип действия операторов деления:  
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2) # Истинное деление   
(2.5, 2.5, -2.5, -2.5)  
>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2) # Деление с округлением вниз (2,   
2.0, -3.0, -3)  
>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0) # Оба оператора  
(3.0, 3.0, 3, 3.0)  
В версии 2.6 операторы деления действуют, как показано ниже:  
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2) # Классическое деление  
(2, 2.5, -2.5, -3)  
>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2) # Деление с округлением вниз  
(2, 2.0, -3.0, -3)  
>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0) # Оба оператора  
(3, 3.0, 3, 3.0)  
Хотя это еще и не проявилось в полной мере, тем не менее, можно предполо-  
жить, что такое поведение оператора / в версии 3.0 может отрицательно ска-  
заться на правильной работе большого числа программ. Благодаря опыту, при-  
обретенному при работе с языком C, многие программисты интуитивно пола-  
C, многие программисты интуитивно пола-  
, многие программисты интуитивно пола-  
гаются на то, что операция деления целых чисел будет выполнена с усечением,   
и пройдет какое-то время, прежде чем они научатся пользоваться оператором   
// в таких случаях. Рассмотрите простой пример цикла while в главе 13 и соот-  
ветствующее упражнение в конце четвертой части книги, которые иллюстри-  
руют программный код, на котором может сказаться это изменение в поведе-  
нии оператора /. Кроме того, обратите внимание на специальную команду from,   
использованную в этом разделе� подробнее она будет рассмотрена в главе 24.  
Точность представления целых чисел  
Операция деления может немного отличаться в разных версиях Python, но   
в основе своей она является достаточно стандартной. А теперь добавим немного   
экзотики. Как уже упоминалось выше, в Python 3.0 поддерживаются целые   
числа неограниченной точности:  
>>> 999999999999999999999999999999 + 1  
1000000000000000000000000000000

Числа в действии   
171  
В версии Python 2.6 для представления длинных целых чисел имеется отдель-  
Python 2.6 для представления длинных целых чисел имеется отдель-  
 2.6 для представления длинных целых чисел имеется отдель-  
ный тип, но интерпретатор автоматически преобразует обычное целое число   
в длинное целое, если его значение превысит величину, которая может быть   
представлена обычным целым числом. При этом вам не требуется использо-  
вать какой-либо специальный синтаксис, чтобы определить длинное целое   
число, а единственным признаком в версии 2.6, свидетельствующим, что ис-  
пользуется длинное целое число, является завершающий символ L, который   
появляется при выводе таких чисел:  
>>> 999999999999999999999999999999 + 1  
1000000000000000000000000000000L  
Целые числа с неограниченной точностью – это достаточно удобный инстру-  
мент. Например, с их помощью можно представить национальный бюджет   
в копейках (если вам это интересно и в вашем компьютере имеется достаточ-  
ный объем памяти). Именно по этой причине нам удалось в примерах из гла-  
вы 3 вычислить такую большую степень числа 2. Ниже приводятся примеры   
для версий 3.0 и 2.6:  
>>> 2 \*\* 200  
1606938044258990275541962092341162602522202993782792835301376  
   
>>> 2 \*\* 200  
1606938044258990275541962092341162602522202993782792835301376L  
Так как интерпретатору приходится производить дополнительные действия,   
чтобы обеспечить поддержку высокой точности, с ростом значений целочис-  
ленные операции могут ощутимо замедляться. Однако если высокая точность   
важна, то фактор низкой производительности отходит на задний план.  
Комплексные числа  
Комплексные числа используются не так часто, как другие числовые типы,   
тем не менее, они также относятся к базовым типам данных в Python. Если вы   
знаете о существовании комплексных чисел, вы должны знать, когда они ис-  
пользуются. В противном случае этот раздел можно пропустить.  
Комплексные числа состоят из двух вещественных чисел, представляющих   
действительную и мнимую части, и в тексте программы отличаются наличием   
суффикса j, или J после мнимой части. Если действительная часть не равна   
нулю, комплексное число записывается как сумма двух частей с помощью сим-  
вола +. Например, комплексное число, действительная часть которого равна 2,   
а мнимая часть – -3, записывается как 2 + -3j.Ниже приводятся примеры не-  
которых действий над комплексными числами:  
>>> 1j \* 1J  
(-1+0j)  
>>> 2 + 1j \* 3  
(2+3j)  
>>> (2 + 1j) \* 3  
(6+3j)  
Кроме того, комплексные числа позволяют обращаться к своим частям как   
к атрибутам, поддерживают все обычные математические операции и могут   
обрабатываться с помощью стандартного модуля cmath (версия модуля math,

172   
Глава 5. Числа   
предназначенная для работы с комплексными числами). Комплексные числа   
обычно используются в инженерных программах. Поскольку это инструмент   
повышенной сложности, ищите подробности в справочном руководстве к язы-  
ку Python.  
Шестнадцатеричная, восьмеричная и двоичная   
формы записи чисел  
Как уже говорилось ранее, целые числа в языке Python могут записываться не   
только в десятичной, но еще и в шестнадцатеричной, восьмеричной и двоичной   
форме. Правила записи литералов целых чисел рассматривались в начале этой   
главы. Теперь рассмотрим несколько практических примеров.  
Имейте в виду, что это всего лишь альтернативный синтаксис задания значе-  
ний целочисленных объектов. Например, следующие литералы создают обыч-  
ные целые числа с заданными значениями:  
>>> 0o1, 0o20, 0o377 # Восьмеричные литералы  
(1, 16, 255)  
>>> 0x01, 0x10, 0xFF # Шестнадцатеричные литералы  
(1, 16, 255)  
>>> 0b1, 0b10000, 0b11111111 # Двоичные литералы  
(1, 16, 255)  
Здесь восьмеричное значение 0o377, шестнадцатеричное значение 0xFF и дво-  
ичное значение 0b11111111 соответствуют десятичному значению 255. По умол-  
чанию интерпретатор Python выводит числа в десятичной системе счисления   
(по основанию 10), но предоставляет встроенные функции, которые позволяют   
преобразовывать целые числа в последовательности цифр в других системах   
счисления:  
>>> oct(64), hex(64), bin(64)  
(‘0o100’, ‘0x40’, ‘0b1000000’)  
Функция oct преобразует десятичное число в восьмеричное представление,   
функция hex – в шестнадцатеричное, а функция bin – в двоичное. Кроме того,   
существует возможность обратного преобразования – встроенная функция int   
преобразует строку цифр в целое число. Во втором необязательном аргументе   
она может принимать основание системы счисления:  
>>> int(‘64’), int(‘100’, 8), int(‘40’, 16), int(‘1000000’, 2)  
(64, 64, 64, 64)  
   
>>> int(‘0x40’, 16), int(‘0b1000000’, 2) # Допускается использовать литералы  
(64, 64)  
Функция eval, с которой мы встретимся далее в книге, интерпретирует стро-  
ку во входном аргументе как программный код на языке Python. Поэтому она   
может воспроизводить похожий эффект. Правда, обычно она работает заметно   
медленнее, потому что ей приходится компилировать и выполнять строку как   
часть программы, а это предполагает, что вы должны иметь безграничное до-  
верие к источнику запускаемой строки, – достаточно грамотный пользователь   
мог бы подсунуть вашей программе строку, которая при выполнении в функ-  
ции eval удалит все файлы на вашем компьютере!:

Числа в действии   
173  
>>> eval(‘64’), eval(‘0o100’), eval(‘0x40’), eval(‘0b1000000’)  
(64, 64, 64, 64)  
Наконец, целые числа могут быть преобразованы в восьмеричное и шестнад-  
цатеричное представления с помощью строкового  метода  форматирования   
и оператора форматирования строк:  
>>> ‘{0:o}, {1:x}, {2:b}’.format(64, 64, 64)  
‘100, 40, 1000000’  
   
>>> ‘%o, %x, %X’ % (64, 255, 255)  
‘100, ff, FF’   
Форматирование строк будет подробно рассматриваться в главе 7.  
Прежде чем двинуться дальше, считаю своим долгом сделать два замечания.   
Во-первых, пользователи Python 2.6 должны помнить, что литералы восьме-  
ричных чисел в этой версии начинаются с одного нуля – это первоначальный   
формат представления восьмеричных чисел в языке Python:  
>>> 0o1, 0o20, 0o377 # Новый формат восьмеричных литералов в 2.6 (как в 3.0)  
(1, 16, 255)  
>>> 01, 020, 0377 # Прежний формат восьмеричных литералов в 2.6  
(1, 16, 255) # (и в более ранних версиях)  
В версии 3.0 попытка выполнить второй пример приведет к ошибке. Хотя в вер-  
сии 2.6 такой синтаксис считается допустимым, не начинайте строки цифр   
с нуля, если вы не предполагаете использовать их в качестве восьмеричных   
литералов. В версии Python 2.6 такие строки будут восприниматься как числа   
в восьмеричной системе счисления, что может не соответствовать вашим ожи-  
даниям – число 010 всегда соответствует десятичному числу 8, а не десятично-  
му 10 (независимо от того, что вы имели в виду!). Чтобы обеспечить непротиво-  
речивость по аналогии с шестнадцатеричной и двоичной формами, восьмерич-  
ный формат был изменен в версии 3.0 – в версии 3.0 вы должны использовать   
форму записи 0o010, и желательно этот же формат использовать в версии 2.6.  
Во-вторых, обратите внимание, что с помощью литералов можно создавать   
целые числа произвольной величины. Например, ниже с помощью шестнад-  
цатеричного литерала создается целое число, а затем выводится его значение,   
сначала в десятичном, затем в восьмеричном и, наконец, в двоичном представ-  
лениях:  
>>> X = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFF  
>>> X  
5192296858534827628530496329220095  
>>> oct(X)  
‘017777777777777777777777777777777777777’  
>>> bin(X)  
‘0b1111111111111111111111111111111111111111111111111111111111 ...и т.д. ...  
Раз уж разговор зашел о двоичных числах, рассмотрим теперь средства для ра-  
боты с отдельными битами.  
Битовые операции  
Помимо обычных числовых операций (сложение, вычитание и так далее) язык   
Python поддерживает большую часть видов числовых выражений, доступных

174   
Глава 5. Числа   
в языке C. Например, ниже приводится пример выполнения операций пораз-  
рядного сдвига и логических операций:  
>>> x = 1 # 0001  
>>> x << 2 # Сдвиг влево на 2 бита: 0100  
4  
>>> x | 2 # Побитовое ИЛИ: 0011  
3  
>>> x & 1 # Побитовое И: 0001  
1  
В первом выражении двоичное значение 1 (по основанию 2, 0001) сдвигается   
влево на две позиции, в результате получается число 4 (0100). В последних двух   
выражениях выполняются двоичная операция ИЛИ (0001|0010 = 0011) и двоич-  
ная операция И (0001&0001 = 0001). Такого рода операции позволяют хранить сра-  
зу несколько флагов и других значений в одном целом числе.  
Это одна из областей, где поддержка двоичной и шестнадцатеричной форм за-  
писи чисел в Python 2.6 и 3.0 оказывается наиболее полезной, – она позволяет   
записывать и выводить числа в виде строк битов:  
>>> X = 0b0001 # Двоичные литералы  
>>> X << 2 # Сдвиг влево  
4  
>>> bin(X << 2) # Строка двоичных цифр  
‘0b100’  
   
>>> bin(X | 0b010) # Битовая операция ИЛИ  
‘0b11’  
>>> bin(X & 0b1) # Битовая операция И   
‘0b1’  
   
>>> X = 0xFF # Шестнадцатеричные литералы  
>>> bin(X)  
‘0b11111111’  
>>> X ^ 0b10101010 # Битовая операция ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR)  
85  
>>> bin(X ^ 0b10101010)  
‘0b1010101’  
   
>>> int(‘1010101’, 2) # Преобразование изображения числа в число по основанию  
85  
>>> hex(85) # Вывод числа в шестнадцатеричном представлении  
‘0x55’  
Мы не будем здесь слишком углубляться в «жонглирование битами». Вам пока   
достаточно знать, что битовые операции поддерживаются языком, и они могут   
пригодиться, когда вы будете иметь дело, например, с сетевыми пакетами или   
упакованными двоичными данными, которые производятся программами на   
языке C. Тем не менее вы должны понимать, что в языках высокого уровня,   
таких как Python, битовые операции не имеют такого большого значения, как   
в низкоуровневых языках, подобных языку C. Как правило, если у вас возни-  
кает желание использовать битовые операции в программах на языке Python,   
вам необходимо вспомнить, на каком языке вы программируете. В Python име-  
ются гораздо лучшие способы представления информации, чем последователь-  
ности битов.

Числа в действии   
175  
В версии Python 3.1 у целых чисел появился новый метод bit\_  
length, который возвращает количество битов, необходимых для   
представления числа в двоичном виде. Тот же результат можно   
получить, уменьшив на 2 длину строки с двоичным представле-  
нием числа, воспользовавшись встроенной функцией len, с кото-  
рой мы встречались в главе 4, однако этот способ работает менее   
эффективно:  
>>> X = 99  
>>> bin(X), X.bit\_length()  
(‘0b1100011’, 7)  
>>> bin(256), (256).bit\_length()  
(‘0b100000000’, 9)  
>>> len(bin(256)) - 2  
9  
Другие встроенные средства для работы с числами  
В дополнение к этим базовым типам объектов Python предоставляет встроен-  
ные функции и модули для работы с числами. Встроенные функции pow и abs,   
например, вычисляют степень и абсолютное значение числа соответственно.   
В следующем примере демонстрируется использование модуля math (который   
содержит реализацию множества функций из библиотеки math языка C) и не-  
скольких встроенных функций:  
>>> import math  
>>> math.pi, math.e # Распространенные константы  
(3.1415926535897931, 2.7182818284590451)  
   
>>> math.sin(2 \* math.pi / 180) # Синус, тангенс, косинус  
0.034899496702500969  
   
>>> math.sqrt(144), math.sqrt(2) # Квадратный корень  
(12.0, 1.4142135623730951)  
   
>>> pow(2, 4), 2 \*\* 4 # Возведение в степень  
(16, 16)  
   
>>> abs(-42.0), sum((1, 2, 3, 4)) # Абсолютное значение, сумма   
(42.0, 10)  
   
>>> min(3, 1, 2, 4), max(3, 1, 2, 4) # Минимум, максимум  
(1, 4)  
Функция sum, которая демонстрируется здесь, принимает последовательность   
чисел, а функции min и max могут принимать либо последовательность чисел,   
либо множество отдельных аргументов. Существует множество способов окру-  
гления десятичных знаков в вещественных числах. Ранее мы уже встречались   
с операциями усечения и округления вниз� мы точно так же можем выполнять   
арифметическое округление и производить снижение разрядности при выводе:  
>>> math.floor(2.567), math.floor(-2.567) # Округление вниз   
(2, -3) # (до ближайшего наименьшего целого)  
   
>>> math.trunc(2.567), math.trunc(-2.567) # Усечение   
(2, -2) # (отбрасывание дробной части)

176   
Глава 5. Числа   
>>> int(2.567), int(-2.567) # Усечение   
(2, -2) # (преобразование в целое число)  
   
>>> round(2.567), round(2.467), round(2.567, 2) # Округление ( в Python 3.0)  
(3, 2, 2.5699999999999998)  
   
>>> ‘%.1f’ % 2.567, ‘{0:.2f}’.format(2.567) # Округление при отображении (‘2.6’, ‘2.5  
7’) # (глава 7)  
В последней инструкции используются операции, с которыми мы уже встреча-  
лись выше, позволяющие получать строки для вывода и поддерживающие рас-  
ширенные возможности форматирования. Кроме того, о чем также говорилось   
выше, предпоследняя инструкция в этом примере вывела бы (3, 2, 2.57), если   
бы мы заключили ее в вызов функции print. Несмотря на внешнее сходство по-  
лучаемых результатов, две последние инструкции имеют существенные отли-  
чия – функция round округляет вещественное число и возвращает веществен-  
ное число, тогда как операции форматирования строк возвращают строку, а не   
измененное число:  
>>> (1 / 3), round(1 / 3, 2), (‘%.2f’ % (1 / 3))  
(0.33333333333333331, 0.33000000000000002, ‘0.33’)  
Интересно, что в языке Python существует три способа вычисления квадрат-  
Python существует три способа вычисления квадрат-  
 существует три способа вычисления квадрат-  
ных корней: с помощью функции из модуля math, с помощью выражения и с   
помощью встроенной функции (если вам интересно узнать, какой способ имеет   
наивысшую производительность, ознакомьтесь с упражнением в конце четвер-  
той части и с его решением, где приводятся результаты тестирования):  
>>> import math  
>>> math.sqrt(144) # Функция из модуля math  
12.0  
>>> 144 \*\* .5 # Выражение  
12.0  
>>> pow(144, .5) # Встроенная функция  
12.0  
   
>>> math.sqrt(1234567890) # Большие числа  
35136.418286444619  
>>> 1234567890 \*\* .5  
35136.418286444619  
>>> pow(1234567890, .5)  
35136.418286444619  
Обратите внимание, что модули из стандартной библиотеки, такие как math,   
необходимо импортировать, а встроенные функции, такие как abs и round, до-  
ступны всегда, без выполнения операции импорта. Говоря другими словами,   
модули – это внешние компоненты, а встроенные функции постоянно распола-  
гаются в пространстве имен, которое используется интерпретатором Python   
по умолчанию для поиска имен, используемых программой. В Python 3.0 это   
пространство имен соответствует модулю с именем builtin (\_\_builtin\_\_ в вер-  
сии 2.6). В этой книге мы подробнее поговорим о разрешении имен, а пока вся-  
кий раз, когда слышите слово «модуль», думайте: «импорт».  
Модуль random из стандартной библиотеки также необходимо импортировать.   
Этот модуль предоставляет возможность получения случайных вещественных   
чисел в диапазоне от 0 до 1, случайных целых чисел в заданном диапазоне,   
случайного выбора элементов последовательности и многое другое:

Другие числовые типы   
177  
>>> import random  
>>> random.random()  
0.44694718823781876  
>>> random.random()  
0.28970426439292829  
>>> random.randint(1, 10)  
5  
>>> random.randint(1, 10)  
4  
   
>>> random.choice([‘Life of Brian’, ‘Holy Grail’, ‘Meaning of Life’])  
‘Life of Brian’  
>>> random.choice([‘Life of Brian’, ‘Holy Grail’, ‘Meaning of Life’])  
‘Holy Grail’  
Модуль random может использоваться, например, для перемешивания колоды   
карт в игре, случайного выбора изображения в программе демонстрации слай-  
дов, в программах статистического моделирования и так далее. За дополни-  
тельной информацией обращайтесь к руководству по стандартной библиотеке   
языка Python.  
Другие числовые типы  
В этой главе мы рассматривали базовые числовые типы языка Python – целые   
числа, вещественные числа и комплексные числа. Их вполне достаточно для   
решения математических задач, с которыми придется столкнуться большин-  
ству программистов. Однако язык Python содержит несколько более экзоти-  
ческих числовых типов, которые заслуживают того, чтобы коротко познако-  
миться с ними.  
Числа с фиксированной точностью  
В версии Python 2.4 появился новый базовый числовой тип: числа с фиксиро-  
ванной точностью представления, формально известные, как числа типа Deci-  
mal. Синтаксически такие числа создаются вызовом функции из импортируе-  
мого модуля и не имеют литерального представления. Функционально числа   
с фиксированной точностью напоминают вещественные числа, но с фиксиро-  
ванным числом знаков после запятой, отсюда и название «числа с фиксирован-  
ной точностью».   
Например, с помощью таких чисел можно хранить значение, которое всегда   
будет иметь два знака после запятой. Кроме того, можно указать, как долж-  
ны обрабатываться лишние десятичные цифры – усекаться или округляться.   
И хотя скорость работы с такими числами несколько ниже, чем с обычными   
вещественными числами, тем не менее, тип чисел с фиксированной точностью   
идеально подходит для представления величин, имеющих фиксированную   
точность, таких как денежные суммы, и для достижения лучшей точности   
представления.  
Основы  
Предыдущий абзац требует дополнительных пояснений. Как вы уже знаете   
(или еще не знаете), вещественная арифметика страдает некоторой долей не-

178   
Глава 5. Числа   
точности из-за ограничения объема памяти, выделяемого для хранения веще-  
ственных чисел. Например, результат следующего выражения должен быть   
равен нулю, но точность вычислений страдает из-за недостаточного числа би-  
тов в представлении вещественных чисел:  
>>> 0.1 + 0.1 + 0.1 - 0.3  
5.5511151231257827e-017  
Попытка вывести результат в более дружественной форме мало помогает, пото-  
му что проблема связана с ограниченной точностью представления веществен-  
ных чисел:  
>>> print(0.1 + 0.1 + 0.1 - 0.3)  
5.55111512313e-017  
Однако при использовании чисел с фиксированной точностью результат полу-  
чается точным:  
>>> from decimal import Decimal  
>>> Decimal(‘0.1’) + Decimal(‘0.1’) + Decimal(‘0.1’) - Decimal(‘0.3’)  
Decimal(‘0.0’)  
Как показано в этом примере, числа с фиксированной точностью представле-  
ния создаются вызовом функции конструктора Decimal из модуля decimal, ко-  
торому передается строка, содержащая желаемое число знаков после запятой   
(при необходимости можно воспользоваться функцией str, чтобы преобразо-  
вать вещественное число в строку). Когда в выражении участвуют числа с раз-  
личной точностью представления, Python автоматически выбирает наиболь-  
Python автоматически выбирает наиболь-  
 автоматически выбирает наиболь-  
шую точность для представления результата:  
>>> Decimal(‘0.1’) + Decimal(‘0.10’) + Decimal(‘0.10’) - Decimal(‘0.30’)  
Decimal(‘0.00’)  
В версии Python 3.1 (которая вышла) существует дополнитель-  
ная возможность создавать объекты класса Decimal из объектов   
вещественных чисел путем вызова метода decimal.Decimal.from\_  
float(1.25). Преобразование выполняется точно, но иногда мо-  
жет порождать большое число десятичных знаков.  
Глобальная настройка точности  
В модуле decimal имеются дополнительные инструменты, позволяющие за-  
давать точность представления всех таких чисел и многое другое. Например,   
объект контекста в этом модуле позволяет задавать точность (число знаков по-  
сле запятой) и режим округления (вниз, вверх и так далее). Точность задается   
глобально, для всех чисел с фиксированной точностью, создаваемых в теку-  
щем потоке управления:  
>>> import decimal  
>>> decimal.Decimal(1) / decimal.Decimal(7)  
Decimal(‘0.1428571428571428571428571429’)  
   
>>> decimal.getcontext( ).prec = 4  
>>> decimal.Decimal(1) / decimal.Decimal(7)  
Decimal(‘0.1429’)

Другие числовые типы   
179  
Эта возможность особенно удобна для финансовых приложений, где в де-  
нежных суммах копейки представлены двумя десятичными знаками. Числа   
с фиксированной точностью по сути представляют альтернативный способ   
округления и форматирования числовых значений, например:  
>>> 1999 + 1.33  
2000.3299999999999  
>>>  
>>> decimal.getcontext().prec = 2  
>>> pay = decimal.Decimal(str(1999 + 1.33))  
>>> pay  
Decimal(‘2000.33’)  
Менеджер контекста объектов класса Decimal  
В версиях Python 2.6 и 3.0 (и выше) имеется также возможность временно пе-  
Python 2.6 и 3.0 (и выше) имеется также возможность временно пе-  
 2.6 и 3.0 (и выше) имеется также возможность временно пе-  
реопределять точность с помощью инструкции with менеджера контекста. По-  
сле выхода за пределы инструкции настройки точности восстанавливаются:  
C:\misc> C:\Python30\python  
>>> import decimal  
>>> decimal.Decimal(‘1.00’) / decimal.Decimal(‘3.00’)  
Decimal(‘0.3333333333333333333333333333’)  
>>>  
>>> with decimal.localcontext() as ctx:  
... ctx.prec = 2  
... decimal.Decimal(‘1.00’) / decimal.Decimal(‘3.00’)  
...  
Decimal(‘0.33’)  
>>>  
>>> decimal.Decimal(‘1.00’) / decimal.Decimal(‘3.00’)  
Decimal(‘0.3333333333333333333333333333’)  
При всей полезности этой инструкции для работы с ней требуется больше зна-  
ний, чем вы успели приобрести к настоящему моменту. Подробное описание   
инструкции with приводится в главе 33.  
Поскольку тип чисел с фиксированной точностью достаточно редко использу-  
ется на практике, за дополнительными подробностями я отсылаю вас к руко-  
водствам по стандартной библиотеке Python и к интерактивной справочной си-  
стеме. Числа с фиксированной точностью, как и рациональные числа, решают   
некоторые проблемы с точностью, присущие вещественным числам, поэтому   
сейчас мы перейдем к следующему разделу, чтобы сравнить их.  
Рациональные числа  
В Python 2.6 и 3.0 появился новый числовой тип – Fraction, который реализует   
объекты рациональных чисел. Объекты этого типа в явном виде хранят числи-  
тель и знаменатель рациональной дроби, что позволяет избежать неточности   
и некоторых других ограничений, присущих вещественным числам.  
Основы  
Тип Fraction является своего рода родственником типа Decimal, описанного   
в предыдущем разделе, и точно так же может использоваться для управления   
точностью представления чисел за счет определения количества десятичных

180   
Глава 5. Числа   
разрядов и политики округления. Оба типа используются похожими спосо-  
бами – как и класс Decimal, класс Fraction находится в модуле. Чтобы создать   
объект этого типа, необходимо импортировать модуль и вызвать конструктор   
класса, передав ему числитель и знаменатель. Как это делается, показано   
в примере ниже:  
>>> from fractions import Fraction  
>>> x = Fraction(1, 3) # Числитель, знаменатель  
>>> y = Fraction(4, 6) # Будет упрощено до 2, 3 с помощью функции gcd  
   
>>> x  
Fraction(1, 3)  
>>> y  
Fraction(2, 3)  
>>> print(y)  
2/3  
После создания объекты типа Fraction могут использоваться в математических   
выражениях как обычные числа:  
>>> x + y  
Fraction(1, 1)  
>>> x – y # Точный результат: числитель, знаменатель  
Fraction(-1, 3)  
>>> x \* y  
Fraction(2, 9)  
Рациональные числа могут создаваться из строк с представлением веществен-  
ных чисел, как и числа с фиксированной точностью:  
>>> Fraction(‘.25’)  
Fraction(1, 4)  
>>> Fraction(‘1.25’)  
Fraction(5, 4)  
>>>  
>>> Fraction(‘.25’) + Fraction(‘1.25’)  
Fraction(3, 2)  
Точность  
Обратите внимание на отличия от вещественной арифметики, точность кото-  
рой ограничивается возможностями аппаратных средств, реализующих веще-  
ственную математику. Для сравнения ниже приводятся некоторые операции   
над вещественными числами с примечаниями, касающимися ограничений   
точности:  
>>> a = 1 / 3.0 # Точность ограничивается аппаратными средствами  
>>> b = 4 / 6.0 # В процессе вычислений точность может быть потеряна  
>>> a  
0.33333333333333331  
>>> b  
0.66666666666666663  
>>> a + b  
1.0  
>>> a - b  
-0.33333333333333331

Другие числовые типы   
181  
>>> a \* b  
0.22222222222222221  
Ограничения точности, свойственные вещественным числам, становятся осо-  
бенно заметны для значений, которые не могут быть представлены точно, из-за   
ограниченного объема памяти, выделяемого для хранения вещественного чис-  
ла. Оба типа, Fraction и Decimal, предоставляют возможность получить точный   
результат, хотя и за счет некоторой потери производительности. Например,   
в следующем примере (взятом из предыдущего раздела) при использовании   
вещественных чисел получается неточный результат, но два других типа по-  
зволяют получить его:  
>>> 0.1 + 0.1 + 0.1 - 0.3 # Должен быть получен ноль (близко, но не точно)  
5.5511151231257827e-17  
   
>>> from fractions import Fraction  
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)  
Fraction(0, 1)  
   
>>> from decimal import Decimal  
>>> Decimal(‘0.1’) + Decimal(‘0.1’) + Decimal(‘0.1’) - Decimal(‘0.3’)  
Decimal(‘0.0’)  
Кроме того, использование рациональных чисел и чисел с фиксированной точ-  
ностью позволяет получить более понятные и точные результаты, чем исполь-  
зование вещественных чисел (за счет использования представления в виде ра-  
циональной дроби и ограничения точности):  
>>> 1 / 3 # В Python 2.6 используйте знаменатель 3.0, чтобы  
0.33333333333333331 # выполнить операцию истинного деления  
   
>>> Fraction(1, 3) # Точное представление  
Fraction(1, 3)  
   
>>> import decimal  
>>> decimal.getcontext().prec = 2  
>>> decimal.Decimal(1) / decimal.Decimal(3)  
Decimal(‘0.33’)  
Фактически рациональные числа сохраняют точность и автоматически упро-  
щают результат. В продолжение предыдущего примера:  
>>> (1 / 3) + (6 / 12) # В Python 2.6 используйте знаменатель “.0”, чтобы  
0.83333333333333326 # выполнить операцию истинного деления  
   
>>> Fraction(6, 12) # Будет упрощено автоматически   
Fraction(1, 2)  
   
>>> Fraction(1, 3) + Fraction(6, 12)  
Fraction(5, 6)  
   
>>> decimal.Decimal(str(1/3)) + decimal.Decimal(str(6/12))  
Decimal(‘0.83’)  
   
>>> 1000.0 / 1234567890  
8.1000000737100011e-07  
>>> Fraction(1000, 1234567890)  
Fraction(100, 123456789)

182   
Глава 5. Числа   
Преобразование и смешивание   
в выражениях значений разных типов  
Для поддержки преобразования в рациональные числа в объектах веществен-  
ных чисел был реализован метод as\_integer\_ratio, возвращающий соответству-  
ющие числу числитель и знаменатель� объекты рациональных чисел обладают   
методом from\_float� а функция float теперь может принимать объекты типа   
Fraction в качестве аргумента. Взгляните на следующий пример, где показано,   
как используются эти методы (символ \* во втором примере – это специальный   
синтаксис распаковывания кортежа в отдельные аргументы – подробнее об   
этом будет рассказываться в главе 18, когда мы будем обсуждать вопросы пере-  
дачи аргументов функциям):  
>>> (2.5).as\_integer\_ratio() # метод объекта типа float  
(5, 2)  
   
>>> f = 2.5  
>>> z = Fraction(\*f.as\_integer\_ratio()) # Пеобразование float -> fraction:   
>>> z # два аргумента  
Fraction(5, 2) # То же самое, что и Fraction(5, 2)  
   
>>> x # x – из предыдущего примера сеанса   
Fraction(1, 3)  
>>> x + z  
Fraction(17, 6) # 5/2 + 1/3 = 15/6 + 2/6  
   
>>> float(x) # Преобразование fraction -> float  
0.33333333333333331  
>>> float(z)  
2.5  
>>> float(x + z)  
2.8333333333333335  
>>> 17 / 6  
2.8333333333333335  
   
>>> Fraction.from\_float(1.75) # Преобразование float -> fraction:   
Fraction(7, 4) # другой способ  
>>> Fraction(\*(1.75).as\_integer\_ratio())  
Fraction(7, 4)  
Наконец, в выражениях допускается смешивать некоторые типы, при этом   
иногда, чтобы сохранить точность, необходимо вручную выполнить преобра-  
зование в тип Fraction. Взгляните на следующие примеры, чтобы понять, как   
это делается:  
>>> x  
Fraction(1, 3)  
>>> x + 2 # Fraction + int -> Fraction  
Fraction(7, 3)  
>>> x + 2.0 # Fraction + float -> float  
2.3333333333333335  
>>> x + (1./3) # Fraction + float -> float  
0.66666666666666663  
   
>>> x + (4./3)  
1.6666666666666665  
>>> x + Fraction(4, 3) # Fraction + Fraction -> Fraction  
Fraction(5, 3)

Другие числовые типы   
183  
Предупреждение: несмотря на то, что имеется возможность преобразовать ве-  
щественное число в рациональное, в некоторых случаях это может приводить   
к потере точности, потому что в своем первоначальном виде вещественное чис-  
ло может быть неточным. В случае необходимости в подобных случаях можно   
ограничить максимальное значение знаменателя:  
>>> 4.0 / 3  
1.3333333333333333  
>>> (4.0 / 3).as\_integer\_ratio() # Произойдет потеря точности  
(6004799503160661, 4503599627370496)  
   
>>> x  
Fraction(1, 3)  
>>> a = x + Fraction(\*(4.0 / 3).as\_integer\_ratio())  
>>> a  
Fraction(22517998136852479, 13510798882111488)  
   
>>> 22517998136852479 / 13510798882111488. # 5 / 3 (или близкое к нему!)  
1.6666666666666667  
>>> a.limit\_denominator(10) # Упростить до ближайшего рационального  
Fraction(5, 3)  
Чтобы получить дополнительные сведения о типе данных Fraction, попробуйте   
поэкспериментировать с ним самостоятельно и поискать информацию в спра-  
вочном руководстве по стандартной библиотеке Python 2.6 и 3.0 и в других до-  
Python 2.6 и 3.0 и в других до-  
 2.6 и 3.0 и в других до-  
кументах.  
Множества  
В версии Python 2.4 также появился новый тип коллекций – множество, не-  
упорядоченная коллекция уникальных и неизменяемых объектов, которая   
поддерживает операции, соответствующие математической теории множеств.   
По определению, каждый элемент может присутствовать в множестве в един-  
ственном экземпляре независимо от того, сколько раз он будет добавлен. Мно-  
жества могут применяться в самых разных прикладных областях, но особенно   
часто они используются в приложениях обработки числовых данных и при ра-  
боте с базами данных.  
Поскольку этот тип является коллекцией других объектов, он обладает неко-  
торыми особенностями, присущими таким объектам, как списки и словари,   
обсуждение которых выходит за рамки этой главы. Например, множества под-  
держивают итерации, могут изменяться в размерах при необходимости и мо-  
гут содержать объекты разных типов. Как мы увидим ниже, множество на-  
поминает словарь, ключи в котором не имеют значений, но поддерживает ряд   
дополнительных операций.  
Поскольку множества являются неупорядоченными коллекциями и не ото-  
бражают ключи на значения, они не могут быть отнесены ни к последователь-  
ностям, ни к отображениям. Множества – это отдельная категория типов. Но   
так как множества поддерживают набор математических операций (а для мно-  
гих читателей их изучение может представлять лишь академический интерес,   
и они будут использовать множества намного реже, чем более распространен-  
ные объекты, такие как словари), мы познакомимся с основами использования   
множеств в этой главе.

184   
Глава 5. Числа   
Основы множеств в Python 2.6  
В настоящее время существует несколько способов создания объекта множе-  
ства, которые зависят от того, какая версия Python используется – 2.6 или 3.0.   
Так как в этой книге рассматриваются обе версии, начнем со способа, доступ-  
ного в версии 2.6, который также доступен (и иногда является единственно воз-  
можным) в версии 3.0. Чуть ниже будут описаны дополнительные способы, до-  
ступные в версии 3.0. Чтобы создать объект множества, нужно передать после-  
довательность или другой объект, поддерживающий возможность итераций по   
его содержимому, встроенной функции set:  
>>> x = set(‘abcde’)  
>>> y = set(‘bdxyz’)  
Обратно функция возвращает объект множества, который содержит все эле-  
менты объекта, переданного функции (примечательно, что множества не пред-  
усматривают возможность определения позиций элементов, а, кроме того, они   
не являются последовательностями):  
>>> x  
set([‘a’, ‘c’, ‘b’, ‘e’, ‘d’]) # Формат отображения в версии 2.6  
Множества, созданные таким способом, поддерживают обычные математиче-  
ские операции над множествами посредством операторов. Обратите внимание:   
эти операции не могут выполняться над простыми последовательностями –   
чтобы использовать их, нам требуется создать из них множества:  
>>> ‘e’ in x # Проверка вхождения в множество  
True  
   
>>> x – y # Разность множеств  
set([‘a’, ‘c’, ‘e’])  
   
>>> x | y # Объединение множеств  
set([‘a’, ‘c’, ‘b’, ‘e’, ‘d’, ‘y’, ‘x’, ‘z’])  
   
>>> x & y # Пересечение множеств  
set([‘b’, ‘d’])  
   
>>> x ^ y # Симметрическая разность (XOR)  
set([‘a’, ‘c’, ‘e’, ‘y’, ‘x’, ‘z’])  
   
>>> x > y, x < y # Надмножество, подмножество  
(False, False)  
Помимо поддержки операторов выражений, объекты множеств обладают ме-  
тодами, которые реализуют те же и ряд дополнительных операций и обеспечи-  
вают изменение самих множеств, – метод add вставляет новый элемент в мно-  
жество, метод update – выполняет объединение, а метод remove удаляет элемент   
по его значению (вызовите функцию dir, передав ей любой экземпляр типа set,   
чтобы получить полный перечень всех доступных методов). Допустим, что пе-  
ременные x и y сохранили свои значения, присвоенные в предыдущем примере   
интерактивного сеанса:  
>>> z = x.intersection(y) # То же, что и выражение x & y  
>>> z  
set([‘b’, ‘d’])  
>>> z.add(‘SPAM’) # Добавит один элемент  
>>> z

Другие числовые типы   
185  
set([‘b’, ‘d’, ‘SPAM’])  
>>> z.update(set([‘X’, ‘Y’])) # Объединение множеств  
>>> z  
set([‘Y’, ‘X’, ‘b’, ‘d’, ‘SPAM’])  
>>> z.remove(‘b’) # Удалит один элемент  
>>> z  
set([‘Y’, ‘X’, ‘d’, ‘SPAM’])  
Будучи итерируемыми контейнерами, множества могут передаваться функ-  
ции len, использоваться в циклах for и в генераторах списков. Однако так как   
множества являются неупорядоченными коллекциями, они не поддерживают   
операции над последовательностями, такие как индексирование и извлечение   
среза:  
>>> for item in set(‘abc’): print(item \* 3)  
...  
aaa  
ccc  
bbb  
Наконец, операторы выражений, которые были продемонстрированы выше,   
обычно применяются к двум множествам, однако родственные им методы ча-  
сто способны работать с любыми объектами итерируемых типов:  
>>> S = set([1, 2, 3])  
   
>>> S | set([3, 4]) # Операторы выражений требуют,   
set([1, 2, 3, 4]) # чтобы оба операнда были множествами  
>>> S | [3, 4]  
TypeError: unsupported operand type(s) for |: ‘set’ and ‘list’  
   
>>> S.union([3, 4]) # Но их методы способны принимать   
set([1, 2, 3, 4]) # любые итерируемые объекты  
>>> S.intersection((1, 3, 5))  
set([1, 3])  
>>> S.issubset(range(-5, 5))  
True  
За дополнительной информацией по множествам обращайтесь к руководству   
по стандартной библиотеке языка Python или к справочной литературе. Хотя   
операции над множествами в языке Python можно реализовать вручную, при-  
меняя другие типы данных, такие как списки и словари (часто так и делалось   
в прошлом), встроенные операции над множествами обеспечивают высокую   
скорость выполнения стандартных операций, опираясь на эффективные алго-  
ритмы и приемы реализации.  
Литералы множеств в Python 3.0  
Если вы уже были знакомы с множествами и считали их отличным инструмен-  
том, то теперь они стали еще лучше. В Python 3.0 сохранилась возможность ис-  
Python 3.0 сохранилась возможность ис-  
 3.0 сохранилась возможность ис-  
пользовать встроенную функцию set для создания объектов множеств, но при   
этом появилась новая форма литералов множеств, в которых используются   
фигурные скобки, прежде зарезервированные для литералов словарей. В вер-  
сии 3.0 следующие инструкции являются эквивалентными:  
set([1, 2, 3, 4]) # Вызов встроенной функции  
{1, 2, 3, 4} # Литерал множества в версии 3.0

186   
Глава 5. Числа   
Такой синтаксис приобретает определенный смысл, если рассматривать мно-  
жества как словари, в которых ключи не имеют соответствующих им значений.   
А поскольку множества являются неупорядоченными коллекциями уникаль-  
ных и неизменяемых объектов, элементы множеств близко напоминают ключи   
словарей. Сходство становится еще более явным, если учесть, что в версии 3.0   
списки ключей в словарях являются объектами представлений, поддерживаю-  
щими такие операции над множествами, как пересечение и объединение (под-  
робнее об объектах представлений словарей рассказывается в главе 8).  
Независимо от того, каким способом были созданы множества, в версии 3.0   
они отображаются в новой форме литерала. В версии 3.0 встроенная функция   
set все еще необходима для создания пустых множеств и конструирования   
множеств на основе существующих итерируемых объектов (в дополнение к ге-  
нераторам множеств, о которых будет рассказываться ниже, в этой же главе),   
однако для создания множеств с известным содержимым удобнее использовать   
новую форму литералов:  
C:\Misc> c:\python30\python  
>>> set([1, 2, 3, 4]) # Встроенная функция: та же, что и в 2.6  
{1, 2, 3, 4}  
>>> set(‘spam’) # Добавить все элементы итерируемого объекта  
{‘a’, ‘p’, ‘s’, ‘m’}  
   
>>> {1, 2, 3, 4} # Литералы множеств: нововведение в 3.0  
{1, 2, 3, 4}  
>>> S = {‘s’, ‘p’, ‘a’, ‘m’}  
>>> S.add(‘alot’)  
>>> S  
{‘a’, ‘p’, ‘s’, ‘m’, ‘alot’}  
Все операции над множествами, которые описывались в предыдущем разделе,   
в версии 3.0 действуют точно так же, но вывод результатов выглядит несколько   
иначе:  
>>> S1 = {1, 2, 3, 4}  
>>> S1 & {1, 3} # Пересечение  
{1, 3}  
>>> {1, 5, 3, 6} | S1 # Объединение  
{1, 2, 3, 4, 5, 6}  
>>> S1 - {1, 3, 4} # Разность  
{2}  
>>> S1 > {1, 3} # Надмножество  
True  
Обратите внимание, что конструкция {} по-прежнему создает пустой словарь.   
Чтобы создать пустое множество, следует вызвать встроенную функцию set�   
результат операции вывода пустого множества выглядит несколько иначе:  
>>> S1 - {1, 2, 3, 4} # Пустое множество выводится иначе  
set()  
>>> type({}) # Литерал {} обозначает пустой словарь  
<class ‘dict’>  
   
>>> S = set() # Инициализация пустого множества  
>>> S.add(1.23)  
>>> S  
{1.23}

Другие числовые типы   
187  
Как и в Python 2.6, множества, созданные с помощью литералов в версии 3.0,   
поддерживают те же методы, часть из которых способна принимать итерируе-  
мые объекты в качестве операндов, чего не позволяют операторы выражений:  
>>> {1, 2, 3} | {3, 4}  
{1, 2, 3, 4}  
>>> {1, 2, 3} | [3, 4]  
TypeError: unsupported operand type(s) for |: ‘set’ and ‘list’  
   
>>> {1, 2, 3}.union([3, 4])  
{1, 2, 3, 4}  
>>> {1, 2, 3}.union({3, 4})  
{1, 2, 3, 4}  
>>> {1, 2, 3}.union(set([3, 4]))  
{1, 2, 3, 4}  
>>> {1, 2, 3}.intersection((1, 3, 5))  
{1, 3}  
>>> {1, 2, 3}.issubset(range(-5, 5))  
True  
Ограничения, связанные с неизменяемостью   
и фиксированные множества  
Множества – это гибкие и мощные объекты, но они имеют одно ограничение   
в обеих версиях, 3.0 и 2.6, о котором следует помнить – из-за особенностей   
реализации множества могут включать объекты только неизменяемых (или   
так называемых «хешируемых») типов. Отсюда следует, что списки и словари   
не могут добавляться в множества, однако вы можете использовать кортежи,   
если появится необходимость сохранять составные значения. В операциях над   
множествами кортежи сравниваются по своим полным значениям:  
>>> S  
{1.23}  
>>> S.add([1, 2, 3]) # Добавляться могут только неизменяемые объекты  
TypeError: unhashable type: ‘list’  
>>> S.add({‘a’:1})  
TypeError: unhashable type: ‘dict’  
>>> S.add((1, 2, 3))  
>>> S # Ни список, ни словарь не могут быть добавлены  
{1.23, (1, 2, 3)} # но с кортежем таких проблем нет  
   
>>> S | {(4, 5, 6), (1, 2, 3)} # Объединение: то же, что и S.union(...)  
{1.23, (4, 5, 6), (1, 2, 3)}  
>>> (1, 2, 3) in S # Членство: выявляется по полным значениям  
True  
>>> (1, 4, 3) in S  
False  
Кортежи в множествах могут использоваться, например, для хранения дат,   
записей, IP-адресов и так далее (подробнее о кортежах рассказывается ниже,   
в этой же части книги). Сами по себе множества являются изменяемыми объ-  
ектами и потому не могут вкладываться в другие множества, однако, если вам   
потребуется сохранить одно множество внутри другого множества, создайте   
множество с помощью встроенной функции frozenset, которая действует точно   
так же, как функция set, но создает неизменяемое множество, которое невоз-  
можно изменить и потому можно встраивать в другие множества.

188   
Глава 5. Числа   
Генераторы множеств в Python 3.0  
Кроме литералов, в версии 3.0 появилась конструкция генератора множеств.   
По своему виду она напоминает конструкцию генератора списков, с которой   
мы предварительно познакомились в главе 4� только генератор множеств за-  
ключен не в квадратные, а в фигурные скобки, и в результате создает множе-  
ство, а не список. Генератор множеств выполняет цикл и собирает результаты   
выражения в каждой итерации – доступ к значению в текущей итерации обе-  
спечивает переменная цикла. Результатом работы генератора является новое   
множество, обладающее всеми особенностями обычного множества:  
>>> {x \*\* 2 for x in [1, 2, 3, 4]} # Генератор множеств в 3.0  
{16, 1, 4, 9}  
Цикл, который генерирует значения, находится в этом выражении справа,   
а выражение, осуществляющее формирование окончательных значений, –   
слева (x \*\* 2). Как и в случае с генераторами списков, генератор множеств   
возвращает именно то, что говорит: «Вернуть новое множество, содержащее   
квадраты значений X, для каждого X из списка». В генераторах можно также   
использовать другие виды итерируемых объектов, такие как строки (первый   
пример в следующем листинге иллюстрирует создание множества с помощью   
генератора на основе существующего итерируемого объекта):  
>>> {x for x in ‘spam’} # То же, что и: set(‘spam’)  
{‘a’, ‘p’, ‘s’, ‘m’}  
   
>>> {c \* 4 for c in ‘spam’} # Множество результатов выражения  
{‘ssss’, ‘aaaa’, ‘pppp’, ‘mmmm’}  
>>> {c \* 4 for c in ‘spamham’}  
{‘ssss’, ‘aaaa’, ‘hhhh’, ‘pppp’, ‘mmmm’}  
   
>>> S = {c \* 4 for c in ‘spam’}  
>>> S | {‘mmmm’, ‘xxxx’}  
{‘ssss’, ‘aaaa’, ‘pppp’, ‘mmmm’, ‘xxxx’}  
>>> S & {‘mmmm’, ‘xxxx’}  
{‘mmmm’}  
Поскольку для дальнейшего обсуждения генераторов необходимо знание не-  
которых базовых концепций, которые мы еще не рассматривали, мы отложим   
его до последующих глав книги. В главе 8 мы познакомимся с его ближайшим   
родственником в версии 3.0 – генератором словарей, а еще позже я расскажу   
подробнее обо всех генераторах (списков, множеств, словарей и о выражениях-  
генераторах), особенно подробно в главах 14 и 20. Как вы узнаете позже, все   
генераторы, включая генераторы множеств, поддерживают дополнительный   
синтаксис, не рассматривавшийся здесь, например вложенные циклы и услов-  
ные инструкции if, который вам сложно будет понять, пока вы не познакоми-  
тесь с большинством основных инструкций языка.  
Где могут использоваться множества?  
Помимо математических вычислений множества могут использоваться еще   
в целом ряде прикладных областей. Например, поскольку элементы множеств   
являются уникальными, множества можно использовать для фильтрации по-  
вторяющихся значений в других коллекциях. Для этого достаточно просто   
преобразовать коллекцию в множество, а затем выполнить обратное преобра-

Другие числовые типы   
189  
зование (благодаря тому, что множства являются итерируемыми объектами,   
их можно передавать функции list):  
>>> L = [1, 2, 1, 3, 2, 4, 5]  
>>> set(L)  
{1, 2, 3, 4, 5}  
>>> L = list(set(L)) # Удаление повторяющихся значений  
>>> L  
[1, 2, 3, 4, 5]  
Множества могут также использоваться для хранения пунктов, которые уже   
были посещены в процессе обхода графа или другой циклической структуры.   
Например, транзитивный загрузчик модулей и программа вывода дерева на-  
следования, которые мы будем рассматривать в главах 24 и 30 соответственно,   
должны запоминать элементы, которые уже были посещены, чтобы избежать   
зацикливания. Посещенные элементы достаточно эффективно можно запоми-  
нать в виде ключей словаря, однако множества предлагают еще более эффек-  
тивный эквивалент (и для многих – более простой).   
Наконец, множества удобно использовать при работе с большими массивами   
данных (например, с результатами запроса к базе данных) – операция пере-  
сечения двух множеств позволяет получить объекты, присутствующие сразу   
в обеих категориях, а объединение – все объекты, присутствующие в любом   
из множеств. Чтобы продемонстрировать это, ниже приводится несколько   
практичных примеров использования операций над множествами, которые   
применяются к списку людей – служащих гипотетической компании. Здесь   
используются литералы множеств, появившиеся в версии 3.0 (в версии 2.6 ис-  
пользуйте функцию set):  
>>> engineers = {‘bob’, ‘sue’, ‘ann’, ‘vic’}  
>>> managers = {‘tom’, ‘sue’}  
   
>>> ‘bob’ in engineers # bob – инженер?  
True  
   
>>> engineers & managers # Кто одновременно является   
{‘sue’} # инженером и менеджером?  
   
>>> engineers | managers # Все сотрудники из обеих категорий  
{‘vic’, ‘sue’, ‘tom’, ‘bob’, ‘ann’}  
   
>>> engineers – managers # Инженеры, не являющиеся менеджерами  
{‘vic’, ‘bob’, ‘ann’}  
   
>>> managers – engineers # Менеджеры, не являющиеся инженерами  
{‘tom’}  
   
>>> engineers > managers # Все менеджеры являются инженерами?   
False # (надмножество)  
   
>>> {‘bob’, ‘sue’} < engineers # Оба сотрудника - инженеры? (подмножество)  
True  
   
>>> (managers | engineers) > managers # Множество всех сотрудников является   
True # надмножеством менеджеров?  
   
>>> managers ^ engineers # Сотрудники, принадлежащие к какой-то одной   
{‘vic’, ‘bob’, ‘ann’, ‘tom’} # категории

190   
Глава 5. Числа   
>>> (managers | engineers) - (managers ^ engineers) # Пересечение!  
{‘sue’}  
Дополнительные сведения об операциях над множествами вы найдете в спра-  
вочном руководстве по стандартной библиотеке Python и в книгах, посвящен-  
Python и в книгах, посвящен-  
 и в книгах, посвящен-  
ных математической теории и теории реляционных баз данных. Кроме того,   
в главе 8 мы еще раз вернемся к операциям над множествами, которые виде-  
ли здесь, – в контексте объектов представлений словарей, появившихся в Py-  
Py-  
thon 3.0.  
Логические значения  
Иногда утверждается, что логический тип bool в языке Python по своей при-  
роде является числовым, потому что два его значения True и False – это всего   
лишь целые числа 1 и 0, вывод которых настроен особым образом. Хотя этих   
сведений вполне достаточно для большинства программистов, тем не менее,   
я предлагаю исследовать этот тип немного подробнее.  
Официально в языке Python имеется самостоятельный логический тип с име-  
нем bool, с двумя предопределенными значениями True и False. Эти значения   
являются экземплярами класса bool, который в свою очередь является всего   
лишь подклассом (в объектно-ориентированном смысле) встроенного целочис-  
ленного типа int. True и False ведут себя точно так же, как и целые числа 1 и 0,   
за исключением того, что для их вывода на экран используется другая логика –   
они выводятся как слова True и False вместо цифр 1 и 0. Технически это достига-  
ется за счет переопределения в классе bool методов str и repr.  
В соответствии с интерпретацией этих значений значения выражений логи-  
ческого типа выводятся в интерактивной оболочке как слова True и False, а не   
как числа 1 и 0. Можно считать, что логический тип делает истинные значения   
более явными. Например, теперь бесконечный цикл можно оформить как while   
True:, а не как менее очевидный while 1:. Точно так же более понятной становит-  
ся инициализация флагов, например flag = False. Более подробно мы рассмо-  
трим эти инструкции в третьей части книги.  
Во всех остальных практических применениях значения True и False можно   
интерпретировать как предопределенные переменные с целочисленными зна-  
чениями 1 и 0. В любом случае, раньше большинство программистов создавали   
переменные True и False, которым присваивали значения 1 и 0� таким образом,   
тип bool просто следует этому стандартному приему. Его реализация может   
приводить к неожиданным результатам: так как True – это всего лишь целое   
значение 1, которое выводится на экран особым образом, выражение True + 4   
в языке Python даст результат 5:  
>>> type(True)  
<class ‘bool’>  
>>> isinstance(True, int)  
True  
>>> True == 1 # То же самое значение  
True  
>>> True is 1 # Но разные объекты: смотрите следующую главу  
False  
>>> True or False # То же, что и: 1 or 0  
True  
>>> True + 4 # (М-да)  
5

Числовые расширения   
191  
Так как, скорее всего, вам не придется встречаться на практике с выражения-  
ми, такими как последнее выражение в примере, вы можете спокойно игнори-  
ровать его глубокий метафизический смысл....  
Мы еще вернемся к логическому типу в главе 9 (где будет дано определение ис-  
тины в языке Python) и в главе 12 (где познакомимся с такими логическими   
операторами, как and и or).  
Числовые расширения   
Наконец, помимо встроенных числовых типов собственно языка Python вы   
можете встретить различные свободно распространяемые расширения сторон-  
них разработчиков, реализующие еще более экзотические числовые типы. По-  
скольку числовая обработка данных является популярной областью примене-  
ния языка Python, вы без труда найдете массу дополнительных инструментов:  
Например, для реализации массивных вычислений можно использовать рас-  
ширение NumPy (Numeric Python), предоставляющее дополнительные воз-  
Numeric Python), предоставляющее дополнительные воз-  
 Python), предоставляющее дополнительные воз-  
Python), предоставляющее дополнительные воз-  
), предоставляющее дополнительные воз-  
можности, такие как реализация матричных и векторных операций, и обшир-  
ные библиотеки реализации численных алгоритмов. Язык Python и расши-  
hon и расши-  
 и расши-  
рение NumPy используется группами программирования в таких известных   
организациях, как Лос-Аламосская Национальная Лаборатория (Los Alamos)   
и Национальное управление по аэронавтике и исследованию космического   
пространства (NASA), для реализации разнообразных задач, которые ранее   
были написаны на языках C++, FORTRAN и Matlab. Нередко комбинацию   
Python и NumPy рассматривают как свободную и более гибкую альтернативу   
пакету Matlab – вы получаете производительность расширения NumPy плюс   
язык Python и все его библиотеки.  
Расширение NumPy является достаточно сложным, поэтому мы не будем воз-  
вращаться к нему в этой книге. Поддержку дополнительных возможностей   
численного программирования в языке Python, включая инструменты для   
анализа и построения графических изображений, библиотеки реализации   
статистических методов и популярный пакет SciPy, вы можете найти на сайте   
PyPI или в Сети. Кроме того, обратите внимание, что расширение NumPy в на-  
 или в Сети. Кроме того, обратите внимание, что расширение NumPy в на-  
NumPy в на-  
 в на-  
стоящее время не входит в состав стандартной библиотеки Python и должно   
устанавливаться отдельно.  
В заключение  
В этой главе были рассмотрены типы числовых объектов языка Python и опе-  
рации, применяемые к ним. Здесь мы познакомились со стандартными цело-  
численными типами и с вещественными числами, а также с некоторыми экзо-  
тическими, нечасто используемыми типами, такими как комплексные числа,   
рациональные числа и множества. Мы также исследовали синтаксис выраже-  
ний в языке Python, порядок преобразования типов, битовые операции и раз-  
личные литеральные формы представления чисел в сценариях.  
Далее в этой части книги мы рассмотрим следующий тип объектов – строки.   
Однако в следующей главе мы потратим некоторое время на более подробное   
ознакомление с механизмом присваивания значений переменным. Это, пожа-  
луй, самая фундаментальная идея в языке Python, поэтому вам обязательно

192   
Глава 5. Числа   
нужно прочитать следующую главу. Но перед этим ответьте на контрольные   
вопросы главы.  
Закрепление пройденного  
Контрольные вопросы  
1. Каким будет значение выражения 2 \* (3 + 4) в языке Python?  
2. Каким будет значение выражения 2 \* 3 + 4 в языке Python?  
3. Каким будет значение выражения 2 + 3 \* 4 в языке Python?  
4. Какие функции можно использовать для вычисления квадратного корня   
числа и квадрата?  
5. Какой тип будет иметь результат следующего выражения: 1 + 2.0 +3?  
6. Как можно выполнить усечение и округление вещественного числа?  
7. Как можно преобразовать целое число в вещественное?  
8. Как можно вывести целое число в восьмеричном, шестнадцатеричном   
и двоичном представлениях?  
9. Как можно преобразовать строковое представление восьмеричного, шест-  
надцатеричного или двоичного числа в простое целое число?  
Ответы  
1. Результатом выражения будет число 14, то есть результат произведения 2 \* 7,   
потому что круглые скобки предписывают выполнить операцию сложения   
до операции умножения.  
2. В данном случае результат будет равен 10, то есть результату выражения   
6 + 4. При отсутствии круглых скобок в языке Python используются правила   
старшинства операторов, согласно которым оператор умножения имеет бо-  
лее высокий приоритет, чем оператор сложения (стоящий левее).  
3. Данное выражение даст в результате число 14, то есть результат выражения   
2 + 12, согласно тем же правилам старшинства, что и в предыдущем вопросе.  
4. Функции вычисления квадратного корня, а также числа пи, тангенса   
и многие другие доступны в импортируемом модуле math. Чтобы найти ква-  
дратный корень числа, необходимо импортировать модуль math и вызвать   
функцию math.sqrt(N). Чтобы найти квадрат числа, можно воспользовать-  
ся либо оператором возведения в степень X \*\* 2, либо встроенной функцией   
pow(X, 2). Любой из последних двух способов можно также использовать для   
вычисления квадратного корня, выполнив возведение в степень .5 (напри-  
мер, X \*\* .5) .  
5. Результатом будет вещественное число: целые числа будут преобразованы   
в вещественные числа как к наиболее сложному числовому типу в выраже-  
нии и для вычисления результата будет использоваться математика веще-  
ственных чисел.  
6. Функции int(N) и math.trunc(N) выполняют отсечение дробной части, а функ-  
ция round(N, digits) выполняет округление. Округление вниз можно также   
выполнить с помощью функции math.floor(N), а простое округление при ото-  
бражении – за счет использования операций форматирования строк.

Закрепление пройденного   
193  
7. Функция float(I) выполняет преобразование целых чисел в вещественные.   
Смешивание целых и вещественных чисел в одном выражении также при-  
водит к выполнению преобразования. В некотором смысле операцию деле-  
ния / в Python 3.0 также можно рассматривать как средство преобразова-  
ния – она всегда возвращает вещественный результат, включающий дроб-  
ную часть, даже если оба операнда являются целыми числами.  
8. Встроенные функции oct(I) и hex(I) возвращают строковые представления   
целых чисел в восьмеричном и в шестнадцатеричном форматах. В версиях   
Python 2.6 и 3.0 имеется также функция bin(I), которая возвращает стро-  
ковое представление целого числа в двоичном формате. Для этих же целей   
могут использоваться оператор % форматирования строк и строковый метод   
format.  
9. Для преобразования строковых представлений восьмеричных, шестнадца-  
теричных и двоичных чисел в обычные целые числа может использоваться   
функция int(S, base) (достаточно передать в аргументе base значение 8, 16   
или 2, соответственно). Для этих целей может также использоваться функ-  
ция eval(S), но она более медленная и может стать источником проблем,   
связанных с безопасностью. Обратите внимание: целые числа в памяти   
компьютера всегда хранятся в двоичном формате, для их вывода просто ис-  
пользуются разные форматы представления.

Глава 6.  
   
Интерлюдия   
о динамической типизации  
В предыдущей главе мы приступили к детальному исследованию базовых   
типов объектов в языке Python, начав с чисел. Мы продолжим наши исследо-  
вания типов в следующей главе, но прежде чем продолжить, вам необходимо   
разобраться в самой фундаментальной, на мой взгляд, идее программирования   
на языке Python, которая составляет основу гибкости языка, – динамической   
типизации и полиморфизме.  
Как будет показано в этой главе и далее в книге, в сценариях на языке Python   
не производится объявление объектов определенных типов. В действительно-  
сти нам даже не приходится беспокоиться о конкретных типах� более того, они   
могут применяться в более широком диапазоне ситуаций, чем можно было бы   
предусмотреть заранее. Поскольку динамическая типизация составляет осно-  
ву этой гибкости, давайте коротко ознакомимся с этой моделью.  
Отсутствие инструкций объявления  
Если у вас имеется опыт работы с компилирующими языками или с языка-  
ми, обладающими статической типизацией, такими как C, C++ или Java, ве-  
роятно, эта тема в книге вызовет у вас недоумение. Мы все время использовали   
переменные, не объявляя ни их самих, ни их типы, и все как-то работало. На-  
пример, когда вводится инструкция a = 3 в интерактивной оболочке интерпре-  
татора или в файле сценария, как интерпретатор Python узнает, что речь идет   
о целом числе? И вообще, как Python узнает, что есть что?  
Как только вы начинаете задавать такие вопросы, вы попадаете в сферу дей-  
ствия модели динамической типизации, используемой в языке Python. Типы   
данных в языке Python определяются автоматически во время выполнения,   
а не в результате объявлений в программном коде. Это означает, что вам не   
требуется заранее объявлять переменные (эту концепцию проще понять, если   
иметь в виду, что все сводится к переменным, объектам и ссылкам между   
ними).

Отсутствие инструкций объявления   
195  
Переменные, объекты и ссылки  
Как было видно из примеров, приводившихся до сих пор в книге, когда выпол-  
няется операция присваивания, такая как a = 3, интерпретатор выполняет ее,   
хотя перед этим ему нигде не сообщалось, что a – это имя переменной, или что   
она представляет объект целочисленного типа. В языке Python все это решает-  
Python все это решает-  
 все это решает-  
ся весьма естественным способом, как описано ниже:  
Создание переменной  
Переменная (то есть имя), такая как a, создается автоматически, когда в про-  
граммном коде ей впервые присваивается некоторое значение. Все последу-  
ющие операции присваивания просто изменяют значение, ассоциированное   
с уже созданным именем. Строго говоря, интерпретатор Python определяет   
некоторые имена до запуска программного кода, но вполне допустимо ду-  
мать, что переменные создаются первой операцией присваивания.  
Типы переменных  
Переменные не имеют никакой информации о типе или ограничениях, свя-  
занных с ним. Понятие типа присуще объектам, а не именам. Переменные   
универсальны по своей природе – они всегда являются всего лишь ссылка-  
ми на конкретные объекты в конкретные моменты времени.  
Использование переменной  
Когда переменная участвует в выражении, ее имя замещается объектом, на   
который она в настоящий момент ссылается, независимо от того, что это за   
объект. Кроме того, прежде чем переменную можно будет использовать, ей   
должно быть присвоено значение – использование неинициализированной   
переменной приведет к ошибке.  
В итоге складывается следующая картина: переменные создаются при выпол-  
нении операции присваивания, могут ссылаться на объекты любых типов и им   
должны быть присвоены некоторые значения, прежде чем к ним можно будет   
обратиться. Это означает, что от вас не требуется заранее объявлять перемен-  
ные в сценарии, но вы должны инициализировать их перед использованием –   
счетчик, например, должен быть инициализирован нулевым значением, пре-  
жде чем его можно будет наращивать.  
Модель динамической типизации существенно отличается от моделей типов   
в традиционных языках программирования. Динамическая типизация обыч-  
но проще поддается пониманию начинающих, особенно если они четко осозна-  
ют разницу между именами и объектами. Например, если ввести такую ин-  
струкцию:  
>>> a = 3  
интерпретатор Python выполнит эту инструкцию в три этапа, по крайней мере,   
концептуально. Следующие этапы отражают все операции присваивания   
в языке Python:  
1. Создается объект, представляющий число 3.  
2. Создается переменная a, если она еще отсутствует.  
3. В переменную a записывается ссылка на вновь созданный объект, представ-  
ляющий число 3.

196   
Глава 6. Интерлюдия о динамической типизации   
Результатом выполнения этих этапов будет структура, которая показана на   
рис. 6.1. Как показано на схеме, переменные и объекты хранятся в разных   
частях памяти и связаны между собой ссылкой (ссылка на рисунке показа-  
на в виде стрелки). Переменные всегда ссылаются на объекты и никогда – на   
другие переменные, но крупные объекты могут ссылаться на другие объекты   
(например, объект списка содержит ссылки на объекты, которые включены   
в список).  
Имена  
Ссылки  
Объекты  
a  
3  
Рис. 6.1. Имена и объекты после выполнения операции присваивания a = 3.   
Переменная превращается в ссылку на объект 3. Во внутреннем представ-  
лении переменная в действительности является указателем на простран-  
ство памяти с объектом, созданным в результате интерпретации лите-  
рального выражения 3  
Эти ссылки на объекты в языке Python так и называются ссылки, то есть ссыл-  
ки – это своего рода ассоциативная связь, реализованная в виде указателя на   
область памяти.1 Когда бы ни использовалась переменная (то есть ссылка), ин-  
терпретатор Python автоматически переходит по ссылке от переменной к объ-  
екту. Все на самом деле гораздо проще, чем кажется. В конкретных терминах:  
 •  
Переменные – это записи в системной таблице, где предусмотрено место для   
хранения ссылок на объекты.  
 •  
Объекты – это области памяти с объемом, достаточным для представления   
значений этих объектов.  
 •  
Ссылки – это автоматически разыменовываемые указатели на объекты.  
Всякий раз, по крайней мере концептуально, когда в сценарии в результате вы-  
полнения выражения создается новое значение, интерпретатор Python создает   
новый объект (то есть выделяет область памяти), представляющий это значе-  
ние. Внутренняя реализация интерпретатора с целью оптимизации кэширу-  
ет и повторно использует некоторые типы неизменяемых объектов, такие как   
малые целые числа и строки (каждый 0 в действительности не является новой   
областью в памяти� подробнее о механизме кэширования мы поговорим поз-  
1   
Читатели, знакомые с языком C, сочтут, что между ссылками в языке Python и ука-  
зателями в языке C много общего (адреса в памяти). И действительно, ссылки реа-  
лизованы как указатели и они часто играют ту же роль, особенно когда объект отно-  
сится к категории изменяемых (подробнее об этом будет говориться ниже). Однако,   
так как ссылки при использовании всегда автоматически разыменовываются, вы   
никак не сможете использовать саму ссылку – эта особенность ликвидирует огром-  
ный пласт ошибок, которые можно допустить в языке C. Вы можете считать ссылки   
указателями типа «void» в языке C, которые автоматически разыменовываются при   
любом использовании.

Отсутствие инструкций объявления   
197  
же). Но с логической точки зрения все выглядит так, как если бы результат   
каждого выражения был представлен отдельным объектом и каждый объект   
занимал отдельную область памяти.  
С технической точки зрения объекты имеют более сложную структуру, чем   
просто пространство в памяти, необходимое для хранения значения. Каждый   
объект имеет два стандартных поля: описатель типа, используемый для хра-  
нения информации о типе объекта, и счетчик ссылок, используемый для опре-  
деления момента, когда память, занимаемая объектом, может быть освобож-  
дена. Чтобы понять, какую роль играют эти два поля в модели динамической   
типизации, нам необходимо двинуться дальше.  
Информация о типе хранится в объекте,   
но не в переменной  
Чтобы увидеть, как используется информация о типах объектов, посмотрим,   
что произойдет, если выполнить несколько операций присваивания одной   
и той же переменной:  
>>> a = 3 # Это целое число  
>>> a = ‘spam’ # Теперь это строка  
>>> a = 1.23 # Теперь это вещественное число  
Это не совсем типичный программный код на языке Python, но он работает –   
сначала создается целое число, потом строка и, наконец, вещественное число.   
Этот пример особенно странно выглядит для программистов, использовавших   
язык C, поскольку он выглядит так, как если бы при выполнении инструкции   
a = ‘spam’ производилось изменение типа переменной с целочисленного на стро-  
ковый.  
Однако в действительности этого не происходит. В языке Python все реализо-  
Python все реализо-  
 все реализо-  
вано намного проще: имена не имеют типов, как уже указывалось ранее, тип –   
это свойство объекта, а не имени. В предыдущем листинге просто изменяется   
ссылка на объект. Так как переменные не имеют типов, мы в действительности   
не изменяем типы переменных – мы просто записываем в переменные ссылки   
на объекты других типов. На самом деле, все, что можно сказать о переменных   
в языке Python, – это то, что они ссылаются на конкретные объекты в конкрет-  
Python, – это то, что они ссылаются на конкретные объекты в конкрет-  
 – это то, что они ссылаются на конкретные объекты в конкрет-  
ные моменты времени.  
С другой стороны, объекты знают, к какому типу они относятся, – каждый   
объект содержит поле, в котором хранится информация о его типе. Целочис-  
ленный объект 3, например будет содержать значение 3 плюс информацию,   
которая сообщит интерпретатору Python, что объект является целым числом   
(строго говоря – это указатель на объект с названием int, которое играет роль   
имени целочисленного типа). Описатель типа для строки ‘spam’ указывает на   
строковый тип (с именем str). Поскольку информация о типе хранится в объ-  
ектах, ее не нужно хранить в переменных.  
Итак, типы в языке Python – это свойства объектов, а не переменных. В ти-  
Python – это свойства объектов, а не переменных. В ти-  
 – это свойства объектов, а не переменных. В ти-  
пичном программном коде переменная обычно ссылается на объекты только   
одного типа. Т. к. для языка Python это не является обязательным требовани-  
Python это не является обязательным требовани-  
 это не является обязательным требовани-  
ем, программный код на языке Python обладает более высокой гибкостью, чем   
вам может показаться, – при умелом использовании особенностей языка про-  
граммный код смог бы работать со многими типами автоматически.

198   
Глава 6. Интерлюдия о динамической типизации   
Я уже упоминал, что каждый объект имеет два поля – описатель типа и счет-  
чик ссылок. Чтобы разобраться с последним из них, нам нужно пойти еще   
дальше и взглянуть, что происходит в конце жизненного цикла объекта.  
Объекты уничтожаются автоматически   
В листинге из предыдущего раздела мы присваивали одной и той же перемен-  
ной объекты различных типов. Но что происходит с прежним значением, когда   
выполняется новое присваивание? Например, что произойдет с объектом 3 по-  
сле выполнения следующих инструкций�  
>>> a = 3  
>>> a = ‘spam’  
Дело в том, что всякий раз, когда имя ассоциируется с новым объектом, ин-  
терпретатор Python освобождает память, занимаемую предыдущим объектом   
(если на него не ссылается какое-либо другое имя или объект). Такое автома-  
тическое освобождение памяти, занимаемой объектами, называется сборкой   
мусора (garbage collection).  
Чтобы проиллюстрировать сказанное, рассмотрим следующий пример, где   
с имя x ассоциируется с разными объектами при выполнении каждой опера-  
ции присваивания:   
>>> x = 42  
>>> x = ‘shrubbery’ # Освобождается объект 42 (если нет других ссылок)  
>>> x = 3.1415 # Теперь освобождается объект ‘shrubbery’  
>>> x = [1,2,3] # Теперь освобождается объект 3.1415  
Первое, что следует отметить, – каждый раз с именем x связывается объект   
другого типа. Снова, хотя в действительности это не так, возникает ощуще-  
ние изменения типа переменной x с течением времени. Напомню, что в языке   
Python типы связаны с объектами, а не с именами. Так как имена – это всего   
лишь ссылки на объекты, данный код работает.  
Во-вторых, следует помнить, что попутно уничтожаются ссылки на объекты.   
Каждый раз, когда имя x ассоциируется с новым объектом, интерпретатор Py-  
Py-  
thon освобождает пространство, занятое прежним объектом. Например, когда   
с именем x связывается строка ‘shrubbery’, объект 42 немедленно уничтожается   
(при условии, что на него не ссылается никакое другое имя), а пространство   
памяти, занимаемое объектом, возвращается в пул свободной памяти для по-  
вторного использования в дальнейшем.  
Достигается это за счет того, что в каждом объекте имеется счетчик ссылок,   
с помощью которого интерпретатор следит за количеством ссылок, указываю-  
щих на объект в настоящий момент времени. Как только (и именно в этот мо-  
мент) значение счетчика достигает нуля, память, занимаемая объектом, авто-  
матически освобождается. В предыдущем листинге мы исходили из предполо-  
жения, что всякий раз, когда имя x ассоциируется с новым объектом, счетчик   
предыдущего объекта уменьшается до нуля, заставляя интерпретатор осво-  
бождать память.  
Основная выгода от сборки мусора состоит в том, что программист может сво-  
бодно распоряжаться объектами, не будучи обязан освобождать память в сво-

Разделяемые ссылки   
199  
ем сценарии. Интерпретатор сам будет выполнять очистку неиспользуемой па-  
мяти в ходе выполнения программы. На практике эта особенность позволяет   
существенно уменьшить объем программного кода по сравнению с низкоуров-  
невыми языками программирования, такими как C и C++.  
Строго говоря, механизм сборки мусора в основном опирается на   
счетчики ссылок, как было описано выше, однако он способен   
также обнаруживать и удалять объекты с циклическими ссыл-  
ками. Эту особенность можно отключить, если вы уверены, что   
в программе не создаются объекты с циклическими ссылками,   
но по умолчанию она включена.  
Поскольку ссылки реализованы в виде указателей, существу-  
ет возможность создать в объекте ссылку на самого себя или   
на другой объект, который ссылается сам на себя. Например,   
упражнение 6 в конце первой части и его решение в приложении   
B демонстрируют возможность создания циклической ссылки,   
за счет включения в список ссылки на сам список. То же может   
произойти при присваивании значений атрибутам объектов   
пользовательских классов. Несмотря на относительную ред-  
кость таких ситуаций, должен быть предусмотрен механизм их   
обработки, потому что счетчики ссылок в таких объектах никог-  
да не достигнут нуля.  
Дополнительную информацию об обнаружении циклических   
ссылок можно найти в описании модуля gc, в справочном ру-  
ководстве по стандартной библиотеке языка Python. Обратите   
также внимание, что данное описание применимо только к ме-  
ханизму сборки мусора в стандартной реализации CPython. Ре-  
CPython. Ре-  
. Ре-  
ализации Jython и IronPython могут использовать иные прин-  
Jython и IronPython могут использовать иные прин-  
 и IronPython могут использовать иные прин-  
IronPython могут использовать иные прин-  
 могут использовать иные прин-  
ципы работы, хотя и дающие тот же эффект – неиспользуемое   
пространство освобождается автоматически.  
Разделяемые ссылки  
До сих пор мы рассматривали вариант, когда ссылка на объект присваивает-  
ся единственной переменной. Теперь введем в действие еще одну переменную   
и посмотрим, что происходит с именами и объектами в этом случае:  
>>> a = 3  
>>> b = a  
В результате выполнения этих двух инструкций получается схема взаимоот-  
ношений, отраженная на рис. 6.2. Вторая инструкция вынуждает интерпре-  
татор создать переменную b и использовать для инициализации переменную a,   
при этом она замещается объектом, на который ссылается (3), и b превращает-  
ся в ссылку на этот объект. В результате переменные a и b ссылаются на один   
и тот же объект (то есть указывают на одну и ту же область в памяти). В языке   
Python это называется разделяемая ссылка – несколько имен ссылаются на   
один и тот же объект.

200   
Глава 6. Интерлюдия о динамической типизации   
Имена  
Ссылки  
Объекты  
a  
3  
b  
a=3  
b=a  
Рис. 6.2. Имена и объекты после выполнения инструкции присваивания b =   
a. Переменная b превращается в ссылку на объект 3. С технической точки   
зрения переменная в действительности является указателем на область   
памяти объекта, созданного в результате выполнения литерального выра-  
жения 3  
Далее добавим еще одну инструкцию:  
>>> a = 3  
>>> b = a  
>>> a = ‘spam’  
Как во всех случаях присваивания в языке Python, в результате выполнения   
этой инструкции создается новый объект, представляющий строку ‘spam’,   
а ссылка на него записывается в переменную a. Однако эти действия не оказы-  
вают влияния на переменную b – она по-прежнему ссылается на первый объ-  
ект, целое число 3. В результате схема взаимоотношений приобретает вид, по-  
казанный на рис. 6.3.  
a  
3  
b  
Имена  
Ссылки  
Объекты  
‘spam’  
a=3  
b=a  
a=‘spam’  
Рис. 6.3. Имена и объекты после выполнения инструкции присваивания a =   
‘spam’. Переменная a ссылается на новый объект (то есть на область памя-  
ти), созданный в результате выполнения литерального выражения ‘spam’,   
но переменная b по-прежнему ссылается на первый объект 3. Так как эта   
операция присваивания никак не изменяет объект 3, она изменяет только   
переменную a, но не b  
То же самое произошло бы, если бы ссылка на объект ‘spam’ вместо переменной   
a была присвоена переменной b – изменилась бы только переменная b, но не a.   
Аналогичная ситуация возникает, даже если тип объекта не изменяется. На-  
пример, рассмотрим следующие три инструкции:

Разделяемые ссылки   
201  
>>> a = 3  
>>> b = a  
>>> a = a + 2  
В этой последовательности происходят те же самые события: интерпретатор   
Python создает переменную a и записывает в нее ссылку на объект 3. После это-  
го он создает переменную b и записывает в нее ту же ссылку, что хранится в пе-  
ременной a, как показано на рис. 6.2. Наконец, последняя инструкция создает   
совершенно новый объект (в данном случае – целое число 5, которое является   
результатом выполнения операции сложения). Это не приводит к изменению   
переменной b. В действительности нет никакого способа перезаписать значение   
объекта 3 – как говорилось в главе 4, целые числа относятся к категории неиз-  
меняемых, и потому эти объекты невозможно изменить.  
Переменные в языке Python, в отличие от других языков программирования,   
всегда являются указателями на объекты, а не метками областей памяти, до-  
ступных для изменения: запись нового значения в переменную не приводит   
к изменению первоначального объекта, но приводит к тому, что переменная   
начинает ссылаться на совершенно другой объект. В результате инструкция   
присваивания может воздействовать только на одну переменную. Однако ког-  
да в уравнении появляются изменяемые объекты и операции, их изменяющие,   
картина несколько меняется. Чтобы узнать как, давайте двинемся дальше.  
Разделяемые ссылки и изменяемые объекты  
Как будет показано дальше в этой части книги, существуют такие объекты   
и операции, которые приводят к изменению самих объектов. Например, опе-  
рация присваивания значения элементу списка фактически изменяет сам спи-  
сок вместо того, чтобы создавать совершенно новый объект списка. При рабо-  
те с объектами, допускающими такие изменения, необходимо быть особенно   
внимательными при использовании разделяемых ссылок, так как изменение   
одного имени может отразиться на других именах.  
Чтобы проиллюстрировать сказанное, возьмем в качестве примера объекты   
списков, о которых рассказывалось в главе 4. Напомню, что списки, которые   
поддерживают возможность присваивания значений элементам, – это просто   
коллекции объектов, которые в программном коде записываются как литера-  
лы в квадратных скобках:  
>>> L1 = [2, 3, 4]  
>>> L2 = L1  
В данном случае L1 – это список, содержащий объекты 2, 3 и 4. Доступ к эле-  
ментам списка осуществляется по их индексам� так, L1[0] ссылается на объект   
2, то есть на первый элемент в списке L1. Безусловно, списки являются полно-  
ценными объектами, такими же, как целые числа и строки. После выполнения   
двух приведенных выше инструкций L1 и L2 будут ссылаться на один и тот же   
объект, так же, как переменные a и b в предыдущем примере (рис. 6.2). Точно   
так же, если теперь добавить еще одну инструкцию:  
>>> L1 = 24  
переменная L1 будет ссылаться на другой объект, а L2 по-прежнему будет ссы-  
латься на первоначальный список. Однако если синтаксис последней инструк-  
ции чуть-чуть изменить, эффект получится радикально другим:

202   
Глава 6. Интерлюдия о динамической типизации   
>>> L1 = [2, 3, 4] # Изменяемый объект  
>>> L2 = L1 # Создание второй ссылки на тот же самый объект  
>>> L1[0] = 24 # Изменение объекта  
   
>>> L1 # Переменная L1 изменилась  
[24, 3, 4]  
>>> L2 # Но так же изменилась и переменная L2!  
[24, 3, 4]  
Здесь мы не изменяем сам объект L1, изменяется компонент объекта, на кото-  
рый ссылается L1. Данное изменение затронуло часть самого объекта списка.   
Поскольку объект списка разделяется разными переменными (ссылки на него   
находятся в разных переменных), то изменения в самом списке затрагивают   
не только L1, то есть следует понимать, что такие изменения могут сказывать-  
ся в других частях программы. В этом примере изменения обнаруживаются   
также в переменной L2, потому что она ссылается на тот же самый объект, что   
и L1. Здесь мы фактически не изменяли L2, но значение этой переменной изме-  
нилось.  
Как правило, это именно то, что требовалось, но вы должны понимать, как это   
происходит. Это – поведение по умолчанию: если вас оно не устраивает, мож-  
но потребовать от интерпретатора, чтобы вместо создания ссылок он выполнял   
копирование объектов. Скопировать список можно несколькими способами,   
включая встроенную функцию list и модуль copy из стандартной библиотеки.   
Однако самым стандартным способом копирования является получение среза   
от начала и до конца списка (подробнее об этой операции рассказывается в гла-  
вах 4 и 7):  
>>> L1 = [2, 3, 4]  
>>> L2 = L1[:] # Создается копия списка L1  
>>> L1[0] = 24  
   
>>> L1  
[24, 3, 4]  
>>> L2 # L2 не изменилась  
[2, 3, 4]  
Здесь изменения в L1 никак не отражаются на L2, потому что L2 ссылается на   
копию объекта, на который ссылается переменная L1. То есть эти переменные   
указывают на различные области памяти.  
Обратите внимание, что способ, основанный на получении среза, неприменим   
в случае с другим изменяемым базовым типом – со словарями, потому что сло-  
вари не являются последовательностями. Чтобы скопировать словарь, необхо-  
димо воспользоваться методом X.copy(). Следует также отметить, что модуль   
copy из стандартной библиотеки имеет в своем составе универсальную функ-  
цию, позволяющую копировать объекты любых типов, включая вложенные   
структуры (например, словари с вложенными списками):  
import copy  
X = copy.copy(Y) # Создание “поверхностной” копии любого объекта Y  
X = copy.deepcopy(Y) # Создание полной копии: копируются все вложенные части  
В главах 8 и 9 мы будем рассматривать списки и словари во всей полноте и там   
же вернемся к концепции разделяемых ссылок и копирования. А пока просто   
держите в уме, что объекты, допускающие изменения в них самих (то есть из-

Разделяемые ссылки   
203  
меняемые объекты), всегда подвержены описанным эффектам. В число таких   
объектов в языке Python попадают списки, словари и некоторые объекты, объ-  
Python попадают списки, словари и некоторые объекты, объ-  
 попадают списки, словари и некоторые объекты, объ-  
явленные с помощью инструкции class. Если такое поведение является неже-  
лательным, вы можете просто копировать объекты.  
Разделяемые ссылки и равенство  
В интересах полноты обсуждения должен заметить, что возможность сборки   
мусора, описанная ранее в этой главе, может оказаться более принципиаль-  
ным понятием, чем литералы для объектов некоторых типов. Рассмотрим сле-  
дующие инструкции:  
>>> x = 42  
>>> x = ‘shrubbery’ # Объект 42 теперь уничтожен?  
Так как интерпретатор Python кэширует и повторно использует малые целые   
числа и небольшие строки, о чем уже упоминалось ранее, объект 42 скорее все-  
го не будет уничтожен. Он, вероятнее всего, останется в системной таблице для   
повторного использования, когда вы вновь сгенерируете число 42 в программ-  
ном коде. Однако большинство объектов уничтожаются немедленно, как толь-  
ко будет потеряна последняя ссылка, особенно те, к которым применение меха-  
низма кэширования не имеет смысла.  
Например, согласно модели ссылок в языке Python, существует два разных   
способа выполнить проверку равенства. Давайте создадим разделяемую ссыл-  
ку для демонстрации:  
>>> L = [1, 2, 3]  
>>> M = L # M и L – ссылки на один и тот же объект  
>>> L == M # Одно и то же значение  
True  
>>> L is M # Один и тот же объект  
True  
Первый способ, основанный на использовании оператора ==, проверяет, равны   
ли значения объектов. В языке Python практически всегда используется имен-  
Python практически всегда используется имен-  
практически всегда используется имен-  
но этот способ. Второй способ, основанный на использовании оператора is, про-  
веряет идентичность объектов. Он возвращает значение True, только если оба   
имени ссылаются на один и тот же объект, вследствие этого он является более   
строгой формой проверки равенства.  
На самом деле оператор is просто сравнивает указатели, которые реализуют   
ссылки, и тем самым может использоваться для выявления разделяемых ссы-  
лок в программном коде. Он возвращает значение False, даже если имена ссы-  
лаются на эквивалентные, но разные объекты, как, например, в следующем   
случае, когда выполняются два различных литеральных выражения:  
>>> L = [1, 2, 3]  
>>> M = [1, 2, 3] # M и L ссылаются на разные объекты  
>>> L == M # Одно и то же значение  
True  
>>> L is M # Но разные объекты  
False  
Посмотрите, что происходит, если те же самые действия выполняются над ма-  
лыми целыми числами:

204   
Глава 6. Интерлюдия о динамической типизации   
>>> X = 42  
>>> Y = 42 # Должно получиться два разных объекта  
>>> X == Y  
True  
>>> X is Y # Тот же самый объект: кэширование в действии!  
True  
В этом примере переменные X и Y должны быть равны (==, одно и то же значе-  
ние), но не эквивалентны (is, один и тот же объект), потому что было выполне-  
но два разных литеральных выражения. Однако из-за того, что малые целые   
числа и строки кэшируются и используются повторно, оператор is сообщает,   
что переменные ссылаются на один и тот же объект.  
Фактически если вы действительно хотите взглянуть на работу внутренних   
механизмов, вы всегда можете запросить у интерпретатора количество ссылок   
на объект: функция getrefcount из стандартного модуля sys возвращает значе-  
ние поля счетчика ссылок в объекте. Когда я, например, запросил количество   
ссылок на целочисленный объект 1 в среде разработки IDLE, я получил число   
837 (большая часть ссылок была создана системным программным кодом са-  
мой IDLE, а не мною):  
>>> import sys  
>>> sys.getrefcount(1) # 837 указателей на этот участок памяти  
837  
Такое кэширование объектов и повторное их использование не будет иметь   
большого значения для вашего программного кода (если вы не используете опе-  
ратор is!). Так как числа и строки не могут изменяться, совершенно неважно,   
сколько ссылок указывает на один и тот же объект. Однако такое поведение   
наглядно демонстрирует один из реализуемых Python способов оптимизации,   
направленной на повышение скорости выполнения.  
Динамическая типизация повсюду  
В действительности вам совсем не нужно рисовать схемы с именами, объек-  
тами, кружочками и стрелками, чтобы использовать Python. Однако в самом   
начале пути такие схемы иногда помогают разобраться в необычных случаях.   
Если после передачи изменяемого объекта в другую часть программы он воз-  
вращается измененным, значит, вы стали свидетелем того, о чем рассказыва-  
лось в этой главе.  
Более того, если к настоящему моменту динамическая типизация кажется   
вам немного непонятной, вы, вероятно, захотите устранить это недопонимание   
в будущем. Поскольку в языке Python все основано на присваивании и на ссыл-  
ках, понимание основ этой модели пригодится во многих ситуациях. Как будет   
показано позже, одна и та же модель используется в операторах присваивания,   
при передаче аргументов функциям, в переменных цикла for, при импорти-  
ровании модулей и так далее. Но к счастью, в Python реализована всего одна   
модель присваивания! Как только вы разберетесь в динамической типизации,   
вы обнаружите, что подобные принципы применяются повсюду в этом языке   
программирования.  
На практическом уровне наличие динамической типизации означает, что вам   
придется писать меньше программного кода. Однако не менее важно и то, что   
динамическая типизация составляет основу полиморфизма – концепции, ко-

В заключение   
205  
торая была введена в главе 4 и к которой мы еще вернемся далее в этой кни-  
ге, – в языке Python. Поскольку мы не ограничены применением типов в про-  
граммном коде на языке Python, он обладает очень высокой гибкостью. Как   
будет показано далее, при правильном использовании динамической типиза-  
ции и полиморфизма можно создавать такой программный код, который авто-  
матически будет адаптироваться под новые требования в ходе развития вашей   
системы.  
В заключение  
В этой главе мы подробно рассмотрели модель динамической типизации в язы-  
ке Python, то есть способ, который используется интерпретатором для автома-  
тического хранения информации о типах объектов и избавляет нас от необхо-  
димости вставлять код объявлений в наши сценарии. Попутно мы узнали, как   
реализована связь между переменными и объектами. Кроме того, мы исследо-  
вали понятие сборки мусора, узнали, как разделяемые ссылки на объекты мо-  
гут оказывать воздействие на несколько переменных и как ссылки влияют на   
понятие равенства в языке Python.  
Поскольку в языке Python имеется всего одна модель присваивания, и она ис-  
пользуется повсюду в этом языке, очень важно, чтобы вы разобрались в ней,   
прежде чем двигаться дальше. Контрольные вопросы к этой главе должны по-  
мочь вам повторить некоторые идеи этой главы. После этого в следующей главе   
мы возобновим наш обзор объектов, приступив к изучению строк.  
Закрепление пройденного  
Контрольные вопросы  
1. Взгляните на следующие три инструкции. Изменится ли значение перемен-  
ной A?  
A = “spam”  
B = A  
B = “shrubbery”  
2. Взгляните на следующие три инструкции. Изменится ли значение перемен-  
ной A?  
A = [“spam”]  
B = A  
B[0] = “shrubbery”  
3. А что можно сказать об этом случае? Изменится ли значение переменной A?  
A = [“spam”]  
B = A[:]  
B[0] = “shrubbery”  
Ответы  
1. Нет, значением переменной A по-прежнему будет строка ‘spam’. Когда пере-  
менной B присваивается строка ‘shrubbery’, она просто начинает указывать   
на другой строковый объект. Первоначально переменные A и B разделяют

206   
Глава 6. Интерлюдия о динамической типизации   
(то есть ссылаются, или указывают) один и тот же строковый объект со зна-  
чением ‘spam’, но в языке Python два имени никак не связаны между собой.   
Таким образом, назначение переменной B ссылки на другой объект не ока-  
зывает влияние на переменную A. То же самое было бы справедливо, если   
бы последняя инструкция имела вид B = B + ‘shrubbery’, так как операция   
конкатенации создает в результате новый объект, ссылка на который затем   
записывается в переменную B. Мы не можем изменять сами строковые объ-  
екты (или числа, или кортежи), потому что они относятся к категории не-  
изменяемых объектов.  
2. Да, теперь значением переменной A будет [“shrubbery”]. Формально мы не   
изменили ни одну из переменных, ни A, ни B, но мы изменили часть самого   
объекта, на который они ссылаются (указывают), посредством переменной   
B. Поскольку A ссылается на тот же самый объект, что и B, изменения можно   
обнаружить и с помощью переменной A.  
3. Нет, значением переменной A по-прежнему будет список [‘spam’]. Изменение   
самого объекта с помощью ссылки B не оказывает влияния на переменную   
A по той причине, что выражение получения среза создает копию объекта   
списка, прежде чем присвоить ссылку переменной B. После второго операто-  
ра присваивания в программе будет существовать два разных объекта спи-  
сков с одинаковыми значениями (в языке Python мы говорим: «равны, но   
не эквивалентны»). Третья инструкция изменяет значение объекта списка,   
на который ссылается переменная B, а объект, на который ссылается пере-  
менная A, остается неизменным.

Глава 7.  
   
Строки  
Следующий основной тип на пути нашего иcследования встроенных объектов   
языка Python – это строки, упорядоченные последовательности символов, ис-  
пользуемые для хранения и представления текстовой информации. Мы корот-  
ко познакомились со строками в главе 4. Здесь мы займемся более глубоким   
их исследованием и восполним некоторые подробности, которые ранее были   
опущены.  
С функциональной точки зрения строки могут использоваться для представ-  
ления всего, что только может быть выражено в текстовой форме: символы   
и слова (например, ваше имя), содержимое текстовых файлов, загруженных   
в память, адреса в Интернете, программы на языке Python и так далее. Они   
могут также использоваться для хранения двоичных значений байтов и сим-  
волов Юникода.  
Возможно, вам уже приходилось использовать строки в других языках про-  
граммирования. Строки в языке Python играют ту же роль, что и массивы   
символов в языке C, но они являются инструментом более высокого уровня,   
нежели простой массив символов. В отличие от C, строки в языке Python об-  
ладают мощным набором средств для их обработки. Кроме того, в отличие от   
таких языков, как C, в языке Python отсутствует специальный тип для пред-  
ставления единственного символа, поэтому в случае необходимости использу-  
ются односимвольные строки.  
Строго говоря, строки в языке Python относятся к категории неизменяемых   
последовательностей, в том смысле, что символы, которые они содержат, име-  
ют определенный порядок следования слева направо и сами строки невозмож-  
но изменить. Фактически строки – это первый представитель большого класса   
объектов, называемых последовательностями, которые мы будем здесь изу-  
чать. Обратите особое внимание на операции над последовательностями, пред-  
ставленные в этой главе, так как они похожим образом работают и с другими   
типами последовательностей, такими как списки и кортежи, которые мы бу-  
дем рассматривать позже.  
В табл. 7.1 приводятся наиболее типичные литералы строк и операций, ко-  
торые обсуждаются в этой главе. Пустые строки записываются как пара ка-

208   
Глава 7. Строки   
вычек (или апострофов), между которыми ничего нет� и существуют различ-  
ные способы записи строк в программном коде. Для работы со строками под-  
держиваются операции выражений, такие как конкатенация (объединение   
строк), выделение подстроки, выборка символов по индексам (по смещению от   
начала строки) и так далее. Помимо выражений язык Python предоставляет   
ряд строковых методов, которые реализуют обычные задачи работы со стро-  
ками, а также модули для решения более сложных задач обработки текста,   
таких как поиск по шаблону. Все эти инструменты будут рассматриваться да-  
лее в этой же главе.  
Таблица 7.1. Наиболее типичные литералы строк и операции   
Операция  
Интерпретация  
S = ‘’  
Пустая строка  
S = “spam’s”  
Строка в кавычках  
S = ‘s\np\ta\x00m’  
Экранированные последовательности  
block = “””...”””  
Блоки в тройных кавычках  
S = r’\temp\spam’  
Неформатированные строки  
S = b’spam’  
Строки байтов в версии 3.0 (глава 36)  
S = u’spam’  
Строки с символами Юникода.   
Только в версии 2.6 (глава 36)  
S1 + S2  
S \* 3  
Конкатенация, повторение  
S[i]  
S[i:j]  
len(S)  
Обращение к символу по индексу, извлечение   
подстроки (среза), длина  
“a %s parrot” % kind  
Выражение форматирования строки  
“a {0} parrot”.format(kind)  
Строковый метод форматирования в 2.6 и 3.0  
S.find(‘pa’)  
Вызов строкового метода: поиск  
S.rstrip()  
Удаление ведущих и конечных пробельных   
символов  
S.replace(‘pa’, ‘xx’)  
Замена  
S.split(‘,’)  
Разбиение по символу-разделитлю  
S.isdigit()  
Проверка содержимого  
S.lower()  
Преобразование регистра символов  
S.endswith(‘spam’)  
Проверка окончания строки  
‘spam’.join(strlist)  
Сборка строки из списка  
S.encode(‘latin-1’)  
Кодирование строк Юникода

Ответы   
209  
Операция  
Интерпретация  
for x in S: print(x)  
‘spam’ in S  
[c \* 2 for c in S]  
map(ord, S)  
Обход в цикле, проверка на вхождение  
Помимо базового набора инструментов для работы со строками Python также   
поддерживает более сложные приемы обработки строк, основанные на при-  
менении шаблонов, с помощью библиотечного модуля re (regular expression –   
регулярные выражения), о котором говорилось в главе 4, а кроме того, такие   
высокоуровневые инструменты обработки текста, как парсеры XML, которые   
будут коротко представлены в главе 36. Однако основное внимание в этой кни-  
ге будет уделено базовым операциям над строками, перечисленным в табл. 7.1.  
Данная глава начинается с краткого обзора форм строковых литералов и ба-  
зовых строковых выражений, затем мы перейдем к более сложным инстру-  
ментам, таким как методы строк и форматирование. В состав Python входит   
множество инструментов для работы со строками, но здесь мы не будем рас-  
сматривать все, что доступно, – полное описание приводится в справочном ру-  
ководстве по стандартной библиотеке Python. Наша цель – исследовать наи-  
Python. Наша цель – исследовать наи-  
. Наша цель – исследовать наи-  
более часто используемые инструменты, чтобы получить достаточно полное   
представление об имеющихся возможностях. Методы, которые мы не будем   
рассматривать здесь, по своему использованию очень напоминают те, что рас-  
сматриваются.  
Примечание по существу: Строго говоря, в этой главе рассма-  
тривается лишь часть возможностей Python для работы со стро-  
ками – тот набор, знание которого необходимо большинству   
программистов. Здесь будет представлен основной строковый   
тип str, который применяется для работы с текстовыми данны-  
ми ASCII и принцип действия которого не изменяется в разных   
версиях Python. То есть эта глава преднамеренно ограничивает-  
ся изучением основных способов обработки строк, которые при-  
меняются в большинстве сценариев на языке Python.  
С более формальной точки зрения, ASCII – это всего лишь разно-  
ASCII – это всего лишь разно-  
 – это всего лишь разно-  
видность текста Юникода. Различие между текстовыми и дво-  
ичными данными в языке Python обеспечивается за счет под-  
держки различных типов данных:  
 • В Python 3.0 существует три строковых типа: str – для пред-  
ставления текста Юникода (содержащего символы в кодировке   
ASCII и сим волы в других кодировках), bytes – для представле-  
ния двоичных данных (включая кодированный текст) и byte-  
array – изменяемый вариант типа bytes.  
 • В Python 2.6 для представления текста Юникода используется   
тип unicode, а для представления двоичных и текстовых дан-  
ных, состоящих из 8-битных символов, используется тип str.

210   
Глава 7. Строки   
Тип bytearray также доступен в версии 2.6, но не в более ранних версиях,   
и он не так тесно связан с двоичными данными, как в версии 3.0. Посколь-  
ку большинству программистов не требуется полное знание всех тонкостей   
кодирования Юникода или форматов представления двоичных данных,   
я поместил описание всех этих тонкостей в раздел «Расширенные возмож-  
ности» этой книги, в главу 36.  
Если вам действительно потребуется знание расширенных концепций рабо-  
ты со строками, таких как альтернативные наборы символов или упакован-  
ные двоичные данные и файлы, после знакомства с материалом, который   
приводится здесь, обращайтесь к главе 36. А теперь мы сосредоточимся на   
основном строковом типе и на его операциях. Основы, которые мы будем   
здесь рассматривать, в равной степени относятся и к дополнительным стро-  
ковым типам в языке Python.  
Литералы строк  
Вообще говоря, работать со строками в языке Python достаточно удобно. Са-  
мое сложное, пожалуй, – это наличие множества способов записи строк в про-  
граммном коде:  
 •  
Строки в апострофах: ‘spa”m’  
 •  
Строки в кавычках: “spa’m”  
 •  
Строки в тройных кавычках: ‘’’... spam ...’’’, “””... spam ...”””   
 •  
Экранированные последовательности: “s\tp\na\0m”  
 •  
Неформатированные строки: r”C:\new\test.spm”  
 •  
Строки байтов в версии 3.0 (глава 36): b’sp\x01am’  
 •  
Строки символов Юникода, только в версии 2.6 (глава 36): u’eggs spam’  
Варианты представления строк в апострофах и кавычках являются наиболее   
типичными, остальные играют особую роль, и мы отложим обсуждение по-  
следних двух форм до главы 36. Давайте коротко рассмотрим каждый из этих   
вариантов.  
Строки в апострофах и в кавычках – это одно и то же  
Кавычки и апострофы, окружающие строки, в языке Python являются взаи-  
Python являются взаи-  
 являются взаи-  
мозаменяемыми. То есть строковые литералы можно заключать как в апостро-  
фы, так и в кавычки – эти две формы представления строк ничем не отличают-  
ся, и обе они возвращают объект того же самого типа. Например, следующие   
две строки совершенно идентичны:  
>>> ‘shrubbery’, “shrubbery”  
(‘shrubbery’, ‘shrubbery’)  
Причина наличия двух вариантов состоит в том, чтобы позволить вставлять   
в литералы символы кавычек и апострофов, не используя для этого символ об-  
ратного слеша. Вы можете вставлять апострофы в строки, заключенные в ка-  
вычки, и наоборот:   
>>> ‘knight”s’, “knight’s”  
(‘knight”s’, “knight’s”)

Литералы строк   
211  
Между прочим, Python автоматически объединяет последовательности строко-  
вых литералов внутри выражения, хотя нет ничего сложного в том, чтобы до-  
бавить оператор + между литералами и вызвать операцию конкатенации явно:  
>>> title = “Meaning “ ‘of’ “ Life” # Неявная конкатенация  
>>> title  
‘Meaning of Life’  
Обратите внимание, если добавить запятые между этими строками, будет по-  
лучен кортеж, а не строка. Кроме того, заметьте, что во всех этих примерах   
интерпретатор предпочитает выводить строки в апострофах, если их нет вну-  
три строки. Кавычки и апострофы можно вставлять в строки, экранируя их   
символом обратного слеша:  
>>> ‘knight\\u2019s’, “knight\\u201ds”  
(“knight’s”, ‘knight”s’)  
Чтобы понять, зачем, вам необходимо узнать, как работает механизм экрани-  
рования вообще.  
Экранированные последовательности   
представляют служебные символы  
В последнем примере кавычка и апостроф внутри строк предваряются симво-  
лом обратного слеша. Это частный случай более общей формы: символы обрат-  
ного слеша используются для вставки специальных символов, известных как   
экранированные последовательности.  
Экранированные последовательности позволяют вставлять в строки символы,   
которые сложно ввести с клавиатуры. В конечном строковом объекте символ   
\ и один или более следующих за ним символов замещаются единственным   
символом, который имеет двоичное значение, определяемое экранированной   
последовательностью. Например, ниже приводится строка из пяти символов,   
в которую вставлены символ новой строки и табуляции:  
>>> s = ‘a\nb\tc’  
Последовательность \n образует единственный символ – байт, содержащий   
двоичное значение кода символа новой строки в используемом наборе симво-  
лов (обычно ASCII-код 10). Аналогично последовательность \t замещается сим-  
волом табуляции. Как будет выглядеть такая строка при печати, зависит от   
того, как она выводится. Функция автоматического вывода в интерактивной   
оболочке отобразит служебные символы как экранированные последователь-  
ности, а инструкция print будет интерпретировать их:  
>>> s  
‘a\nb\tc’  
>>> print(s)  
a  
b c  
Чтобы окончательно убедиться, сколько байтов входит в эту строку, можно   
воспользоваться встроенной функцией len – она возвращает фактическое чис-  
ло байтов в строке независимо от того, как строка отображается на экране:  
>>> len(s)  
5

212   
Глава 7. Строки   
Длина этой строки составляет пять байтов: она содержит байт ASCII-символа   
a, байт символа новой строки, байт ASCII-символа b и так далее. Обратите вни-  
мание, что символы обратного слеша не сохраняются в памяти строкового объ-  
екта – они используются лишь для того, чтобы вынудить интерпретатор сохра-  
нить значения байтов в строке. Для представления служебных символов язык   
Python обеспечивает полный набор экранированных последовательностей,   
перечисленных в табл. 7.2.  
Таблица 7.2. Экранированные последовательности  
Последовательность  
Назначение  
\newline  
Игнорируется (продолжение на новой строке)  
\\  
Сам символ обратного слеша (остается один символ \)  
\\u2019  
Апостроф (остается один символ ‘)  
\\u201d  
Кавычка (остается один символ “)  
\a  
Звонок  
\b  
Забой  
\f  
Перевод формата  
\n  
Новая строка (перевод строки)  
\r  
Возврат каретки  
\t  
Горизонтальная табуляция  
\v  
Вертикальная табуляция  
\xhh  
Символ с шестнадцатеричным кодом hh   
(не более 2 цифр)  
\ooo  
Символ с восьмеричным кодом ooo (не более 3 цифр)  
\0  
Символ Null (не признак конца строки)  
\N{id}  
Идентификатор ID базы данных Юникода   
�hhhh  
16-битный символ Юникода в шестнадцатеричном   
представлении  
�hhhhhhhh  
32-битный символ Юникода в шестнадцатеричном   
представленииa  
\\u0434ругое  
Не является экранированной последовательностью   
(символ обратного слеша сохраняется)  
Некоторые экранированные последовательности позволяют указывать абсо-  
лютные двоичные значения в байтах строк. Например, ниже приводится при-  
мер строки из пяти символов, содержащей два нулевых байта:  
a   
Экранированная последовательность �hhhhhhhh состоит ровно из восьми шестнадца-  
теричных цифр (h). Последовательности � и � могут использоваться только в лите-  
ралах строк символов Юникода.

Литералы строк   
213  
>>> s = ‘a\0b\0c’  
>>> s  
‘a\x00b\x00c’  
>>> len(s)  
5  
В языке Python нулевой байт (символ null) не является признаком завершения   
строки, как в языке C. Интерпретатор просто сохраняет в памяти как текст   
самой строки, так и ее длину. Фактически в языке Python вообще нет символа,   
который служил бы признаком завершения строки. Ниже приводится строка,   
полностью состоящая из экранированных кодов, – двоичные значения 1 и 2   
(записаны в восьмеричной форме), за которыми следует двоичное значение 3   
(записано в шестнадцатеричной форме):  
>>> s = ‘\001\002\x03’  
>>> s  
‘\x01\x02\x03’  
>>> len(s)  
3  
Обратите внимание, что интерпретатор Python отображает непечатаемые сим-  
Python отображает непечатаемые сим-  
 отображает непечатаемые сим-  
волы в шестнадцатеричном представлении, независимо от того, в каком виде   
они были указаны внутри литерала. Вы без ограничений можете комбиниро-  
вать абсолютные экранированные значения с другими типами экранирован-  
ных последовательностей, которые приводятся в табл. 7.2. Следующая строка   
содержит символы “spam”, символ табуляции и символ новой строки, а также   
нулевой символ, заданный в шестнадцатеричном представлении:  
>>> S = “s\tp\na\x00m”  
>>> S  
‘s\tp\na\x00m’  
>>> len(S)  
7  
>>> print(S)  
s p  
a m  
Это обстоятельство приобретает особую важность, когда возникает необхо-  
димость обрабатывать на языке Python файлы с двоичными данными. По-  
скольку содержимое таких файлов в сценариях на языке Python представлено   
строками, вы без труда сможете обрабатывать двоичные файлы, содержащие   
байты с любыми двоичными значениями (подробнее о файлах рассказывается   
в главе 9).1  
1   
Если вам требуется работать с файлами, содержащими двоичные данные, главное от-  
личие в работе с ними заключается в том, что открывать их нужно в режиме двоич-  
ного доступа (добавляя к флагу режима открытия флаг b, например “rb”, “wb” и так   
далее). В Python 3.0 содержимое двоичных файлов интерпретируется как коллек-  
ция строк типа bytes, которые по своим возможностям напоминают обычные строки.   
В Python 2.6 содержимое таких файлов интерпретируется как коллекция обычных   
строк типа str. Кроме того, обратите внимание на модуль struct, который будет опи-  
сан в главе 9, с помощью которого можно выполнять интерпретацию двоичных дан-  
ных, загруженных из файла. Расширенное описание принципов работы с двоичны-  
ми файлами и строками байтов приводится в главе 36.

214   
Глава 7. Строки   
Наконец, последняя строка в табл. 7.2 предполагает, что если интерпретатор   
не распознает символ после \ как корректный служебный символ, он просто   
оставляет символ обратного слеша в строке:  
>>> x = “C:\py\code” # Символ \ сохраняется в строке  
>>> x  
‘C:\\py\\code’  
>>> len(x)  
10  
Однако если вы не способны держать в памяти всю табл. 7.2, вам не следует   
полагаться на описанное поведение.1 Чтобы явно добавить символ обратного   
слеша в строку, нужно указать два символа обратного слеша, идущие подряд   
(\\ – экранированный вариант представления символа \), или использовать не-  
форматированные строки, которые описываются в следующем разделе.  
Неформатированные строки подавляют экранирование  
Как было показано, экранированные последовательности удобно использовать   
для вставки в строки служебных символов. Однако иногда зарезервированная   
экранированная последовательность может порождать неприятности. Очень   
часто, например, можно увидеть, как начинающие программисты пытаются   
открыть файл, передавая аргумент с именем файла, который имеет примерно   
следующий вид:  
myfile = open(‘C:\new\text.dat’, ‘w’)  
думая, что они открывают файл с именем text.dat в каталоге C:\new. Проблема   
здесь заключается в том, что последовательность \n интерпретируется как сим-  
вол новой строки, а последовательность \t замещается символом табуляции.   
В результате функция open будет пытаться открыть файл с именем C:(newline)  
ew(tab)ext.dat, причем обычно безуспешно.  
Именно в таких случаях удобно использовать неформатированные строки.   
Если перед кавычкой, открывающей строку, стоит символ r (в верхнем или   
в нижнем регистре), он отключает механизм экранирования. В результате ин-  
терпретатор Python будет воспринимать символы обратного слеша в строке как   
обычные символы. Таким образом, чтобы ликвидировать проблему, связанную   
с именами файлов в �indows, не забывайте добавлять символ r.  
myfile = open(r’C:\new\text.dat’, ‘w’)  
Как вариант, учитывая, что два идущих подряд символа обратного слеша ин-  
терпретируются как один символ, можно просто продублировать символы об-  
ратного слеша:  
myfile = open(‘C:\\new\\text.dat’, ‘w’)  
Сам интерпретатор Python в определенных случаях использует удваивание об-  
ратного слеша при выводе строк, содержащих обратный слеш:  
>>> path = r’C:\new\text.dat’  
>>> path # Показать, как интерпретатор представляет эту строку  
1   
Мне доводилось встречать людей, которые помнили всю таблицу или большую ее   
часть. Я мог бы посчитать их ненормальными, но тогда мне пришлось бы себя тоже   
включить в их число.

Литералы строк   
215  
‘C:\\new\\text.dat’  
>>> print(path) # Более дружественный формат представления  
C:\new\text.dat  
>>> len(path) # Длина строки  
15  
Так же как и в случае с числами, при выводе результатов в интерактивной обо-  
лочке по умолчанию используется такой формат представления, как если бы   
это был программный код, отсюда и экранирование символов обратного слеша.   
Инструкция print обеспечивает более дружественный формат, в котором каж-  
дая пара символов обратного слеша выводится как один символ. Чтобы про-  
верить, что дело обстоит именно так, можно проверить результат с помощью   
встроенной функции len, которая возвращает число байтов в строке незави-  
симо от формата отображения. Если посчитать символы в выводе инструкции   
print(path), можно увидеть, что каждому символу обратного слеша соответ-  
ствует один байт, а всего строка содержит 15 символов.  
Помимо хранения имен каталогов в �indows, неформатированные строки   
обычно используются для регулярных выражений (возможность поиска по   
шаблону, поддерживаемая модулем re, о котором говорилось в главе 4). Кроме   
того, следует отметить, что в сценариях на языке Python в строках с именами   
каталогов в системах �indows и UNIX можно использовать простые символы   
слеша, потому что Python старается поддерживать переносимость для путей   
к файлам (например, путь к файлу можно указать в виде строки ‘C:/new/text.  
dat’). И все же, когда для кодирования имен каталогов в �indows используется   
традиционная нотация с обратными слешами, удобно использовать неформа-  
тированные строки.  
Несмотря на свое предназначение, даже неформатированная   
строка не может заканчиваться единственным символом обрат-  
ного слеша, потому что обратный слеш в этом случае будет экра-  
нировать следующий за ним символ кавычки – вы по-прежнему   
должны экранировать кавычки внутри строки. То есть кон-  
струкция r”...\\u201d не является допустимым строковым литера-  
лом – неформатированная строка не может заканчиваться не-  
четным количеством символов обратного слеша. Если необходи-  
мо, чтобы неформатированная строка заканчивалась символом   
обратного слеша, можно добавить два символа и затем удалить   
второй из них (r’1\nb\tc\\’[:-1]), добавить один символ вручную   
(r’1\nb\tc’ + ‘\\’) или использовать обычный синтаксис строко-  
вых литералов и дублировать все символы обратного слеша (‘1\\  
nb\\tc\\’). Во всех трех случаях получается одна и та же строка из   
восьми символов, содержащая три обратных слеша.  
Тройные кавычки, многострочные блоки текста  
К настоящему моменту мы познакомились с кавычками, апострофами, экра-  
нированными последовательностями и неформатированными строками. Кро-  
ме этого в арсенале языка Python имеется формат представления строковых   
литералов, в котором используются тройные кавычки. Этот формат иногда   
называют блочной  строкой, который удобно использовать для определения   
многострочных блоков текста в программном коде. Литералы в этой форме на-

216   
Глава 7. Строки   
чинаются с трех идущих подряд кавычек (или апострофов), за которыми мо-  
жет следовать произвольное число строк текста, который закрывается такими   
же тремя кавычками. Внутри такой строки могут присутствовать и кавычки,   
и апострофы, но экранировать их не требуется – строка не считается завершен-  
ной, пока интерпретатор не встретит три неэкранированные кавычки того же   
типа, которые начинают литерал. Например:  
>>> mantra = “””Always look  
... on the bright  
... side of life.”””  
>>>  
>>> mantra  
‘Always look\n on the bright\nside of life.’  
Эта строка состоит из трех строк текста (в некоторых системах строка при-  
глашения к вводу изменяется на ..., когда ввод продолжается на следующей   
линии� среда IDLE просто переводит курсор на следующую линию). Интерпре-  
татор собирает блок текста, заключенный в тройные кавычки, в одну строку,   
добавляя символы новой строки (\n) там, где в программном коде выполнялся   
переход на новую строку. Обратите внимание, что в результате у второй строки   
имеется ведущий пробел, а у третьей – нет, то есть, что вы в действительности   
вводите, то и получаете. Чтобы увидеть, как в реальности интерпретируется   
строка с символами новой строки, можно воспользоваться инструкцией print:  
>>> print(mantra)  
Always look  
 on the bright  
side of life.  
Строки в тройных кавычках удобно использовать, когда в программе требу-  
ется ввести многострочный текст, например чтобы определить многостроч-  
ный текст сообщения об ошибке или код разметки на языке HTML или XML.   
Вы можете встраивать такие блоки текста прямо в свои сценарии, не исполь-  
зуя для этого внешние текстовые файлы или явную операцию конкатенации   
и символы новой строки.  
Часто строки в тройных кавычках используются для создания строк докумен-  
тирования, которые являются литералами строк, воспринимаемыми как ком-  
ментарии при появлении их в определенных местах сценария (подробнее о них   
будет рассказываться позже в этой книге). Комментарии не обязательно (но ча-  
сто!) представляют собой многострочный текст, и данный формат дает возмож-  
ность вводить многострочные комментарии.  
Наконец, иногда тройные кавычки являются ужасающим, хакерским спосо-  
бом временного отключения строк программного кода во время разработки   
(Хорошо, хорошо! На самом деле это совсем не так ужасно, а просто доволь-  
но распространенная практика). Если вам потребуется отключить несколько   
строк программного кода и запустить сценарий снова, просто вставьте по три   
кавычки до и после нужного блока кода, как показано ниже:  
X = 1  
“””  
import os  
print(os.getcwd())  
“””  
Y = 2

Строки в действии   
217  
Я назвал этот прием ужасным, потому что при работе интерпретатор вынуж-  
ден создавать строку из строк программного кода, отключенных таким спосо-  
бом, но, скорее всего, это слабо сказывается на производительности. В случае   
крупных блоков программного кода использовать этот прием гораздо удобнее,   
чем вставлять символы решетки в начале каждой строки, а затем убирать их.   
Это особенно верно, если используемый вами текстовый редактор не поддер-  
живает возможность редактирования исходных текстов на языке Python. При   
программировании на этом языке практичность часто берет верх над эстетич-  
ностью.  
Строки в действии  
Как только с помощью литеральных выражений, с которыми мы только что   
познакомились, строка будет создана, вам наверняка потребуется выполнять   
какие-либо операции с ее участием. В этом и в следующих двух разделах де-  
монстрируются основы работы со строками, форматирование и методы – пер-  
вая линия инструментальных средств обработки текста в языке Python.  
Базовые операции  
Давайте запустим интерактивный сеанс работы с интерпретатором Python,   
чтобы проиллюстрировать базовые операции над строками, которые были пе-  
речислены в табл. 7.1. Строки можно объединять с помощью оператора конка-  
тенации + и дублировать с помощью оператора повторения \*:  
% python  
>>> len(‘abc’) # Длина: число элементов  
3  
>>> ‘abc’ + ‘def’ # Конкатенация: новая строка  
‘abcdef’  
>>> ‘Ni!’ \* 4 # Повторение: подобно “Ni!” + “Ni!” + ...  
‘Ni!Ni!Ni!Ni!’  
Формально операция сложения двух строковых объектов создает новый стро-  
ковый объект, объединяющий содержимое операндов. Операция повторения   
напоминает многократное сложение строки с самой собой. В обоих случаях   
Python позволяет создавать строки произвольного размера – нет никакой не-  
обходимости предварительно объявлять что бы то ни было, включая размеры   
структур данных.1 Встроенная функция len возвращает длину строки (или лю-  
бого другого объекта, который имеет длину).  
1   
В отличие от массивов символов в языке C, при работе со строками в языке Python   
вам не нужно выделять или управлять памятью массивов – вы просто создаете объ-  
екты строк, когда в этом возникает необходимость, и позволяете интерпретатору са-  
мому управлять памятью, выделяемой для этих объектов. Как уже говорилось в гла-  
ве 6, Python автоматически освобождает память, занятую ненужными объектами,   
используя стратегию сборки мусора, основанную на подсчете количества ссылок.   
Каждый объект следит за количеством имен, структур данных и другими компонен-  
тами, которые ссылаются на него. Когда количество ссылок уменьшается до нуля,   
интерпретатор освобождает память, занятую объектом. Это означает, что интерпре-  
татору не требуется останавливаться и просматривать всю память в поисках неис-  
пользуемого пространства (дополнительный компонент сборки мусора собирает так-  
же циклические объекты).

218   
Глава 7. Строки   
Операция повторения на первый взгляд выглядит несколько странно, но она   
очень удобна в очень широком диапазоне применений. Например, чтобы вы-  
вести строку из 80 символов дефиса, можно самому сосчитать до 80, а можно   
возложить эту работу на плечи интерпретатора:  
>>> print(‘------- ...много дефисов... ---’) # 80 дефисов, сложный способ  
>>> print(‘-’ \* 80) # 80 дефисов, простой способ  
Обратите внимание: здесь работает механизм перегрузки операторов: мы ис-  
пользуем те же самые операторы + и \*, которые используются для выполнения   
операций сложения и умножения с числами. Интерпретатор выполняет тре-  
буемую операцию, потому что ему известны типы объектов, к которым приме-  
няются операции сложения и умножения. Но будьте внимательны: правила не   
так либеральны, как может показаться. Например, интерпретатор не позволя-  
ет смешивать строки и числа в выражениях сложения: ‘abc’ + 9 вызовет ошибку   
вместо того, чтобы автоматически преобразовать число 9 в строку.  
Как показано в последней строке табл. 7.1, допускается выполнять обход эле-  
ментов строки в цикле, используя инструкцию for, и проверять вхождение под-  
строки в строку с помощью оператора выражения in, который, по сути, выпол-  
няет операцию поиска. В случае поиска подстрок оператор in напоминает метод   
str.find(), который рассматривается ниже, в этой же главе, – с тем отличием,   
что он возвращает логический результат, а не позицию подстроки в строке:  
>>> myjob = “hacker”  
>>> for c in myjob: print(c, end=’ ‘), # Обход элементов строки в цикле  
...  
h a c k e r  
>>> “k” in myjob # Найдено  
True  
>>> “z” in myjob # Не найдено  
False  
>>> ‘spam’ in ‘abcspamdef’ # Поиск подстроки, позиция не возвращается  
True  
Оператор цикла for присваивает переменной очередной элемент последователь-  
ности (в данном случае – строки) и для каждого элемента выполняет одну или   
более инструкций. В результате переменная c превращается в своего рода кур-  
сор, который постепенно перемещается по строке. Далее в этой книге (особенно   
в главах 14 и 20) мы подробнее рассмотрим средства выполнения итераций, по-  
добных этой и другим, перечисленным в табл. 7.1.  
Доступ по индексам и извлечение подстроки  
Так как строки определены как упорядоченные коллекции символов, мы мо-  
жем обращаться к элементам строк по номерам позиций. В языке Python сим-  
волы извлекаются из строк с помощью операции индексирования – указанием   
числового смещения требуемого компонента в квадратных скобках после име-  
ни строки. В результате операции вы получаете строку, состоящую из одного   
символа, находящегося в указанной позиции.  
Как и в языке C, в языке Python смещения символов в строках начинают ис-  
числяться с 0, а последний символ в строке имеет смещение на единицу мень-  
ше длины строки. Однако, в отличие от C, Python также позволяет извлекать   
элементы последовательностей, таких как строки, используя отрицательные   
смещения. Технически отрицательное смещение складывается с длиной стро-

Строки в действии   
219  
ки, чтобы получить положительное смещение. Отрицательные смещения мож-  
но также представить себе как отсчет символов с конца строки. Например:  
>>> S = ‘spam’  
>>> S[0], S[-2] # Индексация от начала или от конца  
(‘s’, ‘a’)  
>>> S[1:3], S[1:], S[:-1] # Получение среза: извлечение подстроки  
(‘pa’, ‘pam’, ‘spa’)  
В первой строке определяется строка из четырех символов и связывается с име-  
нем S. В следующей строке выполняются две операции доступа к элементам по   
индексам: S[0] возвращает элемент со смещением 0 от начала (односимвольную   
строку ‘s’), а S[-2] возвращает элемент со смещением 2 от конца (что эквива-  
лентно смещению (4 + (-2)) от начала). Смещения и срезы отображаются на эле-  
менты последовательности, как показано на рис. 7.1.1  
[начало:конец]  
По умолчанию используются начало и конец последовательности  
Индексы указывают позиции, в которых будут выполнены «разрезы»  
2  
1  
0  
[ :  
: ]  
S  
L  
I  
С  
E  
O  
F  
S  
P  
A  
M  
Рис. 7.1. Смещения и срезы: положительные смещения отсчитываются   
от левого конца (первый элемент имеет смещение 0), а отрицательные    
отсчитываются от правого конца (последний элемент имеет смещение –1).   
При выполнении операций индексации и получения среза можно использо-  
вать любые виды смещений  
Последняя строка в предыдущем примере демонстрирует операцию извлече-  
ния среза (slicing). Получение среза можно считать некоторой формой синтак-  
сического анализа, особенно при работе со строками, – она позволяет извле-  
кать целые участки (подстроки) за одно действие. Операция получения среза   
может использоваться для извлечения столбцов данных, обрезания ведущих   
и завершающих блоков текста и тому подобного. Позже, в этой же главе, мы   
рассмотрим другой пример использования операции получения среза для син-  
таксического анализа строки.  
Как же выполняется операция получения среза? Когда производится индек-  
сирование объекта последовательности, такого как строка, парой смещений,   
разделенных двоеточием, интерпретатор Python возвращает новый объект,   
содержащий непрерывную область, определяемую парой смещений. Значе-  
ние смещения слева от двоеточия обозначает левую границу (включительно),   
а справа – верхнюю границу (она не входит в срез). Интерпретатор извлека-  
1   
Математически более подготовленные читатели (и студенты в моих классах) иногда   
обнаруживают здесь небольшую асимметрию: самый левый элемент имеет смеще-  
ние 0, а самый правый имеет смещение –1. Увы, в языке Python отсутствует такая   
вещь, как значение –0.

220   
Глава 7. Строки   
ет все элементы от нижней границы до верхней, но верхняя граница в срез не   
включается. Если левая и правая граница опущены, по умолчанию принима-  
ются значения, равные 0 и длине объекта соответственно, из которого извле-  
кается срез.  
Например, для только что рассмотренного примера выражение S[1:3] вернет эле-  
менты со смещениями 1 и 2. То есть будут извлечены второй и третий элементы,   
и операция остановится перед четвертым элементом со смещением, равным 3.   
Выражение S[1:] вернет все элементы, расположенные за первым, – за значение   
верхней границы, которая в выражении опущена, по умолчанию принимается   
длина строки. Наконец, выражение S[:-1] вернет все элементы, кроме послед-  
него, – за значение нижней границы по умолчанию принимается 0, а индекс –1   
соответствует последнему элементу, который в срез не включается.  
Все это может показаться на первый взгляд слишком замысловатым, но опера-  
ции извлечения отдельных элементов и срезов превратятся в простые и мощ-  
ные инструменты, как только вы ими овладеете. Если вы забыли, как выпол-  
няется срез, попробуйте получить его в интерактивном сеансе. В следующей   
главе вы увидите, что существует возможность изменить целый раздел опреде-  
ленного объекта одной инструкцией, достаточно лишь выполнить операцию   
присваивания срезу. Далее приводится краткий обзор для справки:  
 •  
Операция индексирования (S[i]) извлекает компоненты по их смещениям:  
 •  
Первый элемент имеет смещение 0.  
 •  
Отрицательные индексы определяют смещения в обратном порядке – от   
конца, или справа.  
 •  
Выражение S[0] извлекает первый элемент.  
 •  
Выражение S[-2] извлекает второй с конца элемент (так же, как и вы-  
ражение S[len(S)-2]).  
 •  
Операция извлечения подстроки (S[i:j]) извлекает непрерывный раздел   
последовательности:  
 •  
Элемент с индексом, равным верхней границе, не включается в срез.  
 •  
Если границы не указаны, по умолчанию они принимаются равными   
0 и длине последовательности.  
 •  
Выражение S[1:3] извлекает элементы со смещениями от 1 до 3 (не   
включая элемент со смещением 3).  
 •  
Выражение S[1:] извлекает элементы, начиная со смещения 1 и до кон-  
ца (длина последовательности).  
 •  
Выражение S[:3] извлекает элементы, начиная со смещения 0 и до 3 (не   
включая его).  
 •  
Выражение S[:-1] извлекает элементы, начиная со смещения 0 и до по-  
следнего (не включая его).  
 •  
Выражение S[:] извлекает элементы, начиная со смещения 0 и до кон-  
ца, – это эффективный способ создать поверхностную копию последо-  
вательности S.  
Последний пункт списка – это самый обычный трюк: с его помощью создается   
полная поверхностная копия объекта последовательности, то есть объект с тем   
же значением, но расположенный в другой области памяти (подробнее о ко-

Строки в действии   
221  
пировании объектов рассказывается в главе 9). Этот прием не очень полезен   
при работе с неизменяемыми объектами, такими как строки, но его ценность   
возрастает при работе с объектами, которые могут изменяться, – такими, как   
списки.   
В следующей главе вы узнаете, что синтаксис операции доступа к элементам   
последовательности по смещениям (квадратные скобки) также применим для   
доступа к элементам словарей по ключам – операции выглядят одинаково, но   
имеют разную интерпретацию.  
Расширенная операция извлечения подстроки:   
третий предел  
В версии Python 2.3 в операцию извлечения подстроки была добавлена под-  
держка необязательного третьего индекса, используемого как шаг (иногда на-  
зывается как шаг по индексу). Величина шага добавляется к индексу каждого   
извлекаемого элемента. Полная форма записи операции извлечения подстро-  
ки теперь выглядит так: X[I:J:K]. Она означает: «Извлечь все элементы после-  
довательности X, начиная со смещения I, вплоть до смещения J-1, с шагом K».   
Третий предел, K, по умолчанию имеет значение 1, именно по этой причине   
в обычной ситуации извлекаются все элементы среза, слева направо. Однако   
если явно указать значение третьего предела, его можно использовать, чтобы   
пропустить некоторые элементы или полностью изменить их порядок.  
Например, выражение X[1:10:2] вернет каждый второй элемент последователь-  
ности X в диапазоне смещений от 1 до 9 – то есть будут выбраны элементы со   
смещениями 1, 3, 5, 7 и 9. Как правило, по умолчанию первый и второй преде-  
лы принимают значения 0 и длину последовательности соответственно, поэто-  
му выражение X[::2] вернет каждый второй элемент от начала и до конца по-  
следовательности:  
>>> S = ‘abcdefghijklmnop’  
>>> S[1:10:2]  
‘bdfhj’  
>>> S[::2]  
‘acegikmo’  
Можно также использовать отрицательное значение шага. Например, выра-  
жение “hello”[::-1] вернет новую строку “olleh”. Здесь первые две границы по-  
лучают значения по умолчанию – 0 и длина последовательности, а величина   
шага, равная -1, указывает, что срез должен быть выбран в обратном поряд-  
ке – справа налево, а не слева направо. В результате получается перевернутая   
последовательность:  
>>> S = ‘hello’  
>>> S[::-1]  
‘olleh’  
При использовании отрицательного шага порядок применения первых двух   
границ меняется на противоположный. То есть выражение S[5:1:-1] извлечет   
элемент со 2 по 5 в обратном порядке (результат будет содержать элементы по-  
следовательности со смещениями 5, 4, 3 и 2):  
>>> S = ‘abcedfg’  
>>> S[5:1:-1]  
‘fdec’

222   
Глава 7. Строки   
Пропуск элементов и изменение порядка их следования – это наиболее типич-  
ные случаи использования операции получения среза с тремя пределами. За   
более подробной информацией вам следует обратиться к руководству по стан-  
дартной библиотеке языка Python (или поэкспериментировать в интерактив-  
ной оболочке). Мы еще вернемся к операции получения среза с тремя предела-  
ми далее в этой книге, когда будем рассматривать ее в соединении с оператором   
цикла for.  
Далее в этой книге мы также узнаем, что операция извлечения среза эквива-  
лентна операции индексирования, в которой в качестве индекса используется   
специальный объект среза, что очень важно для разработчиков классов, кото-  
рым требуется реализовать поддержку обеих операций:  
>>> ‘spam’[1:3] # Синтаксис извлечения среза  
‘pa’  
>>> ‘spam’[slice(1, 3)] # используется объект среза  
‘pa’  
>>> ‘spam’[::-1]  
‘maps’  
>>> ‘spam’[slice(None, None, -1)]  
‘maps’  
Придется держать в уме: срезы  
Повсюду в этой книге я буду включать аналогичные врезки с описанием   
наиболее типичных случаев использования рассматриваемых особен-  
ностей языка на практике. Поскольку у вас нет возможности осознать   
реальные возможности языка, пока вы не увидите большую часть кар-  
тины, эти врезки будут содержать множество упоминаний тем, незнако-  
мых для вас. Поэтому вам следует воспринимать эти сведения как пред-  
варительное знакомство со способами, которые связывают абстрактные   
концепции языка с решением наиболее распространенных задач про-  
граммирования.  
Например, далее вы увидите, что аргументы командной строки, пере-  
данные сценарию на языке Python при запуске, доступны в виде атрибу-  
та argv встроенного модуля sys:  
# File echo.py  
import sys  
print sys.argv  
   
% python echo.py -a -b -c  
[‘echo.py’, ‘-a’, ‘-b’, ‘-c’]  
Обычно вас будут интересовать только параметры, которые следуют за   
именем программы. Это приводит нас к типичному использованию опе-  
рации получения среза: мы можем с помощью единственной инструк-  
ции получить все элементы списка, за исключением первого. В данном   
случае выражение sys.argv[1:] вернет требуемый список [‘-a’, ‘-b’, ‘-c’].   
После этого список можно обрабатывать по своему усмотрению, не забо-  
тясь о присутствии имени программы в начале.

Строки в действии   
223  
Кроме того, операция получения среза часто используется для удаления   
лишних символов из строк, считываемых из файлов. Если известно,   
что строки всегда завершаются символом новой строки (символ \n), его   
можно удалить одним-единственным выражением – line[:-1], которое   
возвращает все символы строки, кроме последнего (нижняя граница по   
умолчанию принимается равной 0). В обоих случаях операция извлече-  
ния подстроки обеспечивает логику выполнения, которую в низкоуров-  
невых языках программирования пришлось бы реализовывать явно.  
Обратите внимание, что для удаления символа новой строки часто пред-  
почтительнее использовать метод line.rstrip, потому что он не поврежда-  
ет строки, в которых отсутствует символ новой строки в конце, – типич-  
ный случай при создании текстовых файлов некоторыми текстовыми   
редакторами. Операция извлечения подстроки применима, только если   
вы полностью уверены, что строки завершаются корректным образом.  
Инструменты преобразования строк  
Один из девизов языка Python – не поддаваться искушению делать предпо-  
ложения о том, что имелось в виду. Например, Python не позволит сложить   
строку и число, даже если строка выглядит как число (то есть содержит только   
цифры):  
>>> “42” + 1  
TypeError: cannot concatenate ‘str’ and ‘int’ objects  
В соответствии с архитектурой языка оператор + может означать и операцию   
сложения, и операцию конкатенации, вследствие чего выбор типа преобразо-  
вания становится неочевидным. Поэтому интерпретатор воспринимает такую   
инструкцию как ошибочную. Вообще в языке Python отвергается любая ма-  
гия, которая может осложнить жизнь программиста.  
Как же быть, если сценарий получает число в виде текстовой строки из файла   
или от пользовательского интерфейса? В этом случае следует использовать ин-  
струменты преобразования, чтобы можно было интерпретировать строку как   
число или наоборот. Например:  
>>> int(“42”), str(42) # Преобразование из/в строки  
(42, ‘42’)  
>>> repr(42), # Преобразование в строку, как если бы она была   
‘42’ # литералом в программном коде  
Функция int преобразует строку в число, а функция str преобразует число   
в строковое представление (по сути – в то, что выводится на экран). Функция   
repr (и прежний ее эквивалент, обратные апострофы, который был удален   
в Python 3.0) также преобразует объект в строковое представление, но возвра-  
щает объект в виде строки программного кода, который можно выполнить,   
чтобы воссоздать объект. Если объект – строка, то инструкция print выведет   
кавычки, окружающие строку:  
>>> print(str(‘spam’), repr(‘spam’))  
(‘spam’, “’spam’”)

224   
Глава 7. Строки   
Подробнее о различиях между функциями str и repr можно прочитать во врез-  
ке «Форматы представления repr и str», в главе 5. Кроме того, функции int   
и str изначально предназначены для выполнения преобразований.  
Вы не сможете смешивать строковые и числовые типы в таких операторах, как   
+, но вы можете вручную выполнить необходимые преобразования операндов   
перед выполнением операции:  
>>> S = “42”  
>>> I = 1  
>>> S + I  
TypeError: cannot concatenate ‘str’ and ‘int’ objects  
   
>>> int(S) + I # Операция сложения  
43  
   
>>> S + str(I) # Операция конкатенации  
‘421’  
Существуют похожие встроенные функции для преобразования вещественных   
чисел в/из строки:  
>>> str(3.1415), float(“1.5”)  
(‘3.1415’, 1.5)  
   
>>> text = “1.234E-10”  
>>> float(text)  
1.2340000000000001e-010  
Позднее мы познакомимся со встроенной функцией eval – она выполняет стро-  
ку, содержащую программный код на языке Python, и потому может выпол-  
нять преобразование строки в объект любого вида. Функции int и float преоб-  
разуют только числа, но это ограничение означает, что они обычно выполняют   
эту работу быстрее (и безопаснее, потому что они не принимают программный   
код произвольного выражения). Как было показано в главе 5, выражение фор-  
матирования строки также обеспечивает возможность преобразования чисел   
в строки. Форматирование будет обсуждаться ниже, в этой же главе.  
Преобразование кодов символов  
Имеется также возможность выполнить преобразование одиночного символа   
в его целочисленный код ASCII, для чего нужно передать этот символ функ-  
ции ord – она возвращает фактическое числовое значение соответствующего   
байта в памяти. Обратное преобразование выполняется с помощью функции   
chr, она получает целочисленный код ASCII и преобразует его в соответствую-  
щий символ:  
>>> ord(‘s’)  
115  
>>> chr(115)  
‘s’  
Эти функции можно применить ко всем символам строки в цикле. Они могут   
также использоваться для реализации своего рода строковой математики. На-  
пример, чтобы получить следующий по алфавиту символ, его можно преобра-  
зовать в число и выполнить математическое действие над ним:

Строки в действии   
225  
>>> S = ‘5’  
>>> S = chr(ord(S) + 1)  
>>> S  
‘6’  
>>> S = chr(ord(S) + 1)  
>>> S  
‘7’  
Следующий пример преобразования представляет собой альтернативу встро-  
енной функции int, по крайней мере для односимвольных строк, для преобра-  
зования строки в целое число:  
>>> int(‘5’)  
5  
>>> ord(‘5’) - ord(‘0’)  
5  
Такие преобразования могут использоваться в сочетании с операторами цикла,   
представленными в главе 4 и подробно рассматриваемыми в следующей части   
книги, для получения целочисленных значений из строковых представлений   
двоичных чисел. В каждой итерации текущее значение умножается на 2 и за-  
тем к нему прибавляется числовое значение следующей цифры:  
>>> B = ‘1101’ # Двоичные цифры преобразуются в числа с помощью функции ord  
>>> I = 0  
>>> while B != ‘’:  
... I = I \* 2 + (ord(B[0]) - ord(‘0’))  
... B = B[1:]  
...  
>>> I  
13  
Операция побитового сдвига влево (I << 1) могла бы дать тот же эффект, что   
и операция умножения на 2. Но так как мы еще не познакомились с циклами   
и уже видели встроенные функции int и bin в главе 5, которые могут исполь-  
зоваться в задачах преобразования двоичных чисел в Python 2.6 и 3.0, то же   
самое можно реализовать, как показано ниже:  
>>> int(‘1101’, 2) # Преобразовать двоичное представление в целое число  
13  
>>> bin(13) # Преобразовать целое число в двоичное представление  
‘0b1101’  
С течением времени язык Python стремится автоматизировать решение наи-  
Python стремится автоматизировать решение наи-  
 стремится автоматизировать решение наи-  
более типичных задач!  
Изменение строк  
Помните термин «неизменяемая последовательность»? Слово «неизменяемая»   
означает, что вы не можете изменить содержимое самой строки в памяти (то есть   
невозможно изменить элемент строки, выполнив присваивание по индексу):  
>>> S = ‘spam’  
>>> S[0] = “x”  
Возникает ошибка!

226   
Глава 7. Строки   
Тогда каким образом в языке Python производить изменение текстовой инфор-  
мации? Чтобы изменить строку, необходимо создать новую строку с помощью   
таких операций, как конкатенация и извлечение подстроки, и затем, если это   
необходимо, присвоить результат первоначальному имени:  
>>> S = S + ‘SPAM!’ # Чтобы изменить строку, нужно создать новую  
>>> S  
‘spamSPAM!’  
>>> S = S[:4] + ‘Burger’ + S[-1]  
>>> S  
‘spamBurger!’  
Первый пример добавляет подстроку в конец строки S с помощью операции   
конкатенации. В действительности здесь создается новая строка, которая за-  
тем присваивается имени S, но вы можете представить себе это действие как   
«изменение» первоначальной строки. Второй пример замещает четыре симво-  
ла шестью новыми – с помощью операций извлечения подстроки и конкатена-  
ции. Как будет показано далее в этой главе, похожего эффекта можно добиться   
с помощью строкового метода replace. Вот как это выглядит:  
>>> S = ‘splot’  
>>> S = S.replace(‘pl’, ‘pamal’)  
>>> S  
‘spamalot’  
Как и всякая операция, создающая новую строку, строковые методы создают   
новые строковые объекты. Если вам необходимо сохранить эти объекты, вы   
можете присвоить их переменной. Создание нового объекта строки для каж-  
дого изменения – операция не столь неэффективная, как может показаться, –   
вспомните, в предыдущей главе говорилось, что интерпретатор автоматически   
производит сборку мусора (освобождает память, занятую неиспользуемыми   
строковыми объектами), поэтому новые объекты повторно используют память,   
ранее занятую прежними значениями. Интерпретатор Python во многих слу-  
Python во многих слу-  
 во многих слу-  
чаях работает гораздо быстрее, чем можно было бы ожидать.  
Наконец, дополнительно существует возможность сборки текстовых значений   
с помощью выражений форматирования строк. Ниже приводятся два примера   
подстановки значений объектов в строку, при этом происходит преобразова-  
ние объектов в строки и изменение первоначальной строки в соответствии со   
специ фикаторами формата:  
>>> ‘That is %d %s bird!’ % (1, ‘dead’) # Выражение форматирования  
That is 1 dead bird!  
>>> ‘That is {0} {1} bird!’.format(1, ‘dead’) # Метод форматирования   
‘That is 1 dead bird!’ # в 2.6 и 3.0  
Несмотря на впечатление, что происходит замена символов в строке, в действи-  
тельности в результате форматирования получаются новые строковые объек-  
ты, а оригинальные строки не изменяются. Мы будем рассматривать приемы   
форматирования ниже, в этой же главе, где вы увидите, что операции форма-  
тирования могут иметь большую практическую пользу, чем в этом примере.   
Второй пример представляет строковый метод� и мы в первую очередь позна-  
комимся с методами строк, а потом перейдем к изучению приемов форматиро-  
вания.

Строковые методы   
227  
Как будет показано в главе 36, в Python 3.0 и 2.6 появился новый   
строковый тип bytearray, который относится к категории изме-  
няемых, в том смысле, что объекты этого типа могут изменяться   
непосредственно. В действительности объекты типа bytearray не   
являются строками – это последовательности 8-битных целых   
чисел. Однако они поддерживают большую часть строковых   
операций и при отображении выводятся как строки символов   
ASCII. Объекты этого типа обеспечивают возможность хране-  
ния больших объемов текста, который требуется изменять до-  
статочно часто. В главе 36 мы также увидим, что функции ord   
и chr способны работать с символами Юникода, которые могут   
храниться в нескольких байтах.  
Строковые методы  
В дополнение к операторам выражений строки предоставляют набор методов,   
реализующих более сложные операции обработки текста. Методы – это про-  
сто функции, которые связаны с определенными объектами. Формально они   
являются атрибутами, присоединенными к объектам, которые ссылаются на   
функции. В языке Python выражения и встроенные функции могут работать   
с некоторым набором типов, но методы являются специфичными  для  типов   
объектов: строковые методы, например, работают только со строковыми объ-  
ектами. В Python 3.0 некоторые методы могут относиться к разным типам (на-  
Python 3.0 некоторые методы могут относиться к разным типам (на-  
 3.0 некоторые методы могут относиться к разным типам (на-  
пример, многие типы имеют метод count), но при этом по своей функционально-  
сти они в большей степени зависят от типа объекта, чем другие инструменты.  
Если говорить более точно, функции – это пакеты программного кода, а вы-  
зовы методов объединяют в себе выполнение двух операций (извлечение атри-  
бута и вызов функции).  
Извлечение атрибута  
Выражение вида object.attribute означает: «извлечь значение атрибута at-  
tribute из объекта object».  
Вызов функции  
Выражение вида function(arguments) означает: «вызвать программный код   
функции function, передав ему ноль или более объектов-аргументов argu-  
ments, разделенных запятыми, и вернуть значение функции».   
Объединение этих двух действий позволяет вызвать метод объекта. Выраже-  
ние вызова метода object.method(arguments) вычисляется слева направо, то есть   
интерпретатор сначала извлекает метод объекта, а затем вызывает его, переда-  
вая ему входные аргументы. Если метод возвращает какой-либо результат, он   
становится результатом всего выражения вызова метода.  
Как будет много раз показано в этой части книги, большинство объектов обла-  
дает методами, которые можно вызвать, и все они доступны с использованием   
одного и того же синтаксиса вызова метода. Чтобы вызвать метод объекта, вам   
потребуется существующий объект. Давайте перейдем к рассмотрению неко-  
торых примеров.

228   
Глава 7. Строки   
В табл. 7.3 приводятся шаблоны вызова встроенных строковых методов в вер-  
сии Python 3.0 (более полный и обновленный перечень методов вы найдете   
в руководстве по стандартной библиотеке языка Python или воспользовавшись   
функцией help в интерактивной оболочке, передав ей любую строку). В Py-  
Py-  
thon 2.6 набор строковых методов немного отличается – в их число, например,   
входит метод decode, потому что работа со строками Юникода в этой версии реа-  
лизована несколько иначе (подробнее об этом рассказывается в главе 36). В та-  
блице имя S представляет объект строки, а необязательные аргументы заклю-  
чены в квадратные скобки. Строковые методы, представленные в этой таблице,   
реализуют высокоуровневые операции, такие как разбиение и слияние, преоб-  
разование регистра символов, проверка типа содержимого и поиск подстроки.  
Таблица 7.3. Строковые методы   
S.capitalize()  
S.ljust(width [, fill])  
S.center(width [, fill]))  
S.lower()  
S.count(sub [, start [, end]])   
S.lstrip([chars])  
S.encode([encoding [,errors]])   
S.maketrans(x[, y[, z]])  
S.endswith(suffix [, start [,   
end]])   
S.partition(sep)  
S.expandtabs([tabsize])   
S.replace(old, new [, count])  
S.find(sub [, start [, end]])   
S.rfind(sub [,start [,end]])  
S.format(fmtstr, \*args, \*\*kwargs) S.rindex(sub [, start [, end]])  
S.index(sub [, start [, end]])   
S.rjust(width [, fill])  
S.isalnum()   
S.rpartition(sep)  
S.isalpha()   
S.rsplit([sep[, maxsplit]])  
S.isdecimal()  
S.rstrip([chars])  
S.isdigit()  
S.split([sep [,maxsplit]])  
S.isidentifier()  
S.splitlines([keepends])  
S.islower()  
S.startswith(prefix [, start [, end]])  
S.isnumeric()  
S.strip([chars])  
S.isprintable()  
S.swapcase()  
S.isspace()  
S.title()  
S.istitle()  
S.translate(map)  
S.isupper()  
S.upper()  
S.join(iterable)  
S.zfill(width)  
Как видите, строковых методов достаточно много, а в книге не так много места,   
чтобы иметь возможность описать их все, поэтому за подробной информацией   
обращайтесь к руководству по стандартной библиотеке Python или к справоч-  
 к руководству по стандартной библиотеке Python или к справоч-  
к руководству по стандартной библиотеке Python или к справоч-  
 руководству по стандартной библиотеке Python или к справоч-  
руководству по стандартной библиотеке Python или к справоч-  
 по стандартной библиотеке Python или к справоч-  
по стандартной библиотеке Python или к справоч-  
 стандартной библиотеке Python или к справоч-  
стандартной библиотеке Python или к справоч-  
 библиотеке Python или к справоч-  
библиотеке Python или к справоч-  
 Python или к справоч-  
или к справоч-  
 к справоч-  
к справоч-  
 справоч-  
справоч-

Строковые методы   
229  
ным руководствам. А теперь давайте поработаем над программным кодом, ко-  
 руководствам. А теперь давайте поработаем над программным кодом, ко-  
руководствам. А теперь давайте поработаем над программным кодом, ко-  
. А теперь давайте поработаем над программным кодом, ко-  
А теперь давайте поработаем над программным кодом, ко-  
торый демонстрирует некоторые наиболее часто используемые методы в дей-  
ствии и попутно иллюстрирует основные приемы обработки текста, применяе-  
мые в языке Python.   
Примеры использования строковых методов:   
изменение строк   
Как уже говорилось ранее, строки являются неизменяемыми объектами,   
поэтому их невозможно изменить непосредственно. Чтобы из существующей   
строки сконструировать новое текстовое значение, необходимо создать новую   
строку с помощью таких операций, как извлечение подстроки и конкатенация.   
Например, чтобы изменить два символа в середине строки, можно использо-  
вать такой способ:  
>>> S = ‘spammy’  
>>> S = S[:3] + ‘xx’ + S[5:]  
>>> S  
‘spaxxy’  
При этом, если требуется только заменить подстроку, можно воспользоваться   
методом replace:  
>>> S = ‘spammy’  
>>> S = S.replace(‘mm’, ‘xx’)  
>>> S  
‘spaxxy’  
Метод replace является более универсальным, чем предполагает этот про-  
граммный код. Он принимает в качестве аргумента оригинальную подстроку   
(любой длины) и строку (любой длины) замены, и выполняет глобальный поиск   
с заменой:   
>>> ‘aa$bb$cc$dd’.replace(‘$’, ‘SPAM’)  
‘aaSPAMbbSPAMccSPAMdd’  
В этой роли метод replace может использоваться как инструмент реализации   
поиска с заменой по шаблону (например, замены символов формата). Обратите   
внимание, что на этот раз мы просто выводим результат на экран, а не присваи-  
ваем его переменной – присваивать результат переменной необходимо только   
в том случае, если потребуется сохранить результат дальнейшего использова-  
ния.  
Если необходимо заменить одну подстроку фиксированного размера, которая   
может появиться в любом месте, можно также выполнить операцию замены   
или отыскать подстроку с помощью метода find и затем воспользоваться опера-  
циями извлечения подстроки:  
>>> S = ‘xxxxSPAMxxxxSPAMxxxx’  
>>> where = S.find(‘SPAM’) # Поиск позиции  
>>> where # Подстрока найдена со смещением 4  
4  
>>> S = S[:where] + ‘EGGS’ + S[(where+4):]  
>>> S  
‘xxxxEGGSxxxxSPAMxxxx’

230   
Глава 7. Строки   
Метод find возвращает смещение, по которому найдена подстрока (по умолча-  
нию поиск начинается с начала строки), или значение -1, если искомая под-  
строка не найдена. Другой вариант использования метода replace заключается   
в передаче третьего аргумента, который определяет число производимых за-  
мен:  
>>> S = ‘xxxxSPAMxxxxSPAMxxxx’  
>>> S.replace(‘SPAM’, ‘EGGS’) # Заменить все найденные подстроки  
‘xxxxEGGSxxxxEGGSxxxx’  
   
>>> S.replace(‘SPAM’, ‘EGGS’, 1) # Заменить одну подстроку  
‘xxxxEGGSxxxxSPAMxxxx’  
Обратите внимание: метод replace возвращает новую строку. Так как строки   
являются неизменяемыми, методы никогда в действительности не изменяют   
оригинальную строку, даже если они называются «replace» (заменить)!  
Тот факт, что операция конкатенации и метод replace всякий раз создают но-  
вые строковые объекты, может оказаться недостатком их использования для   
изменения строк. Если в сценарии производится множество изменений длин-  
ных строк, вы можете повысить производительность сценария, преобразовав   
строку в объект, который допускает внесение изменений:  
>>> S = ‘spammy’  
>>> L = list(S)  
>>> L  
[‘s’, ‘p’, ‘a’, ‘m’, ‘m’, ‘y’]  
Встроенная функция list (или функция-конструктор объекта) создает новый   
список из элементов любой последовательности – в данном случае «разрывая»   
строку на символы и формируя из них список. Обладая строкой в таком пред-  
ставлении, можно производить необходимые изменения, не вызывая создание   
новой копии строки при каждом изменении:  
>>> L[3] = ‘x’ # Этот прием допустим для списков, но не для строк  
>>> L[4] = ‘x’  
>>> L  
[‘s’, ‘p’, ‘a’, ‘x’, ‘x’, ‘y’]  
Если после внесения изменений необходимо выполнить обратное преобразова-  
ние (чтобы, например, записать результат в файл), можно использовать метод   
join, который «собирает» список обратно в строку:  
>>> S = ‘’.join(L)  
>>> S  
‘spaxxy’  
Метод join на первый взгляд может показаться немного странным. Так как он   
является строковым методом (а не методом списка), он вызывается через ука-  
зание желаемой строки-разделителя. Метод join объединяет строки из списка,   
вставляя строку-разделитель между элементами списка. В данном случае при   
получении строки из списка используется пустая строка-разделитель. В более   
общем случае можно использовать произвольную строку-разделитель:  
>>> ‘SPAM’.join([‘eggs’, ‘sausage’, ‘ham’, ‘toast’])  
‘eggsSPAMsausageSPAMhamSPAMtoast’

Строковые методы   
231  
На практике объединение подстрок в одну строку часто выполняется намного   
быстрее, чем последовательность операций конкатенации отдельных элемен-  
тов списка. Обратите также внимание на упоминавшийся выше строковый тип   
bytearray, появившийся в версиях Python 3.0 и 2.6, который полностью описы-  
Python 3.0 и 2.6, который полностью описы-  
 3.0 и 2.6, который полностью описы-  
вается в главе 36, – благодаря тому, что объекты этого типа могут изменяться   
непосредственно, в некоторых случаях он представляет отличную альтернати-  
ву комбинации функций list/join, особенно когда текст приходится изменять   
достаточно часто.  
Примеры методов строк: разбор текста  
Еще одна распространенная роль, которую играют методы строк, – это про-  
стейший разбор текста, то есть анализ структуры и извлечение подстрок. Для   
извлечения подстрок из фиксированных смещений можно использовать прием   
извлечения срезов:  
>>> line = ‘aaa bbb ccc’  
>>> col1 = line[0:3]  
>>> col3 = line[8:]  
>>> col1  
‘aaa’  
>>> col3  
‘ccc’  
В этом примере поля данных располагаются в фиксированных позициях и по-  
тому могут быть легко извлечены из оригинальной строки. Этот прием может   
использоваться, только если анализируемые компоненты располагаются в из-  
вестных фиксированных позициях. Если поля отделяются друг от друга не-  
которым разделителем, можно воспользоваться методом разбиения строки на   
компоненты. Этот прием используется, когда искомые данные могут распола-  
гаться в произвольных позициях внутри строки:  
>>> line = ‘aaa bbb ccc’  
>>> cols = line.split()  
>>> cols  
[‘aaa’, ‘bbb’, ‘ccc’]  
Строковый метод split преобразует строку в список подстрок, окружаю-  
щих строки-разделители. В предыдущем примере мы не указали строку-  
разделитель, поэтому по умолчанию в качестве разделителей используются   
пробельные символы – строка разбивается на группы по символам пробела,   
табуляции или перевода строки, и в результате мы получили список подстрок.   
В других случаях данные могут отделяться другими разделителями. В следу-  
ющем примере производится разбиение (и, следовательно, разбор) строки по   
символу запятой, который обычно используется для отделения данных, извле-  
ченных из баз данных:  
>>> line = ‘bob,hacker,40’  
>>> line.split(‘,’)  
[‘bob’, ‘hacker’, ‘40’]  
Разделители могут содержать более одного символа:  
>>> line = “i’mSPAMaSPAMlumberjack”  
>>> line.split(“SPAM”)  
[“i’m”, ‘a’, ‘lumberjack’]

232   
Глава 7. Строки   
Хотя оба способа, основанные на извлечении подстрок и разбиении строк, име-  
ют определенные ограничения, они работают достаточно быстро и могут ис-  
пользоваться для разбора текстовой информации в простых случаях.  
Другие часто используемые методы строк в действии  
Другие строковые методы имеют более специфическое предназначение, на-  
пример удаляют пробельные символы в конце текстовой строки, выполняют   
преобразование регистра символов, проверяют характер содержимого строки   
и проверяют наличие подстроки в конце строки или в начале:  
>>> line = “The knights who say Ni!\n”  
>>> line.rstrip()  
‘The knights who say Ni!’  
>>> line.upper()  
‘THE KNIGHTS WHO SAY NI!\n’  
>>> line.isalpha()  
False  
>>> line.endswith(‘Ni!\n’)  
True  
>>> line.startswith(‘The’)  
True  
Для достижения тех же результатов в некоторых случаях могут использовать-  
ся альтернативные приемы – с использованием оператора проверки вхожде-  
ния in можно проверить присутствие подстроки, а функция получения длины   
строки и операция извлечения подстроки могут использоваться для имитации   
действия функции endswith:  
>>> line  
‘The knights who say Ni!\n’  
   
>>> line.find(‘Ni’) != -1 # Поиск с использованием вызова метода или выражения  
True  
>>> ‘Ni’ in line  
True  
   
>>> sub = ‘Ni!\n’  
>>> line.endswith(sub) # Проверка наличия подстроки в конце строки  
True # с помощью метода или операции извлечения подстроки  
>>> line[-len(sub):] == sub  
True  
Обратите также внимание на строковый метод format, который описывается   
ниже в этой главе, – он позволяет реализовать более сложные способы подста-  
новки в одной инструкции, чем комбинирование нескольких операций.  
Для работы со строками существует достаточно много методов, поэтому мы не   
будем рассматривать их все. Некоторые методы вы увидите далее в этой книге,   
а за дополнительной информацией вы можете обратиться к руководству по би-  
блиотеке языка Python и другим источникам информации или просто поэкспе-  
риментировать с ними в интерактивном режиме. Кроме того, можно изучить   
результат вызова функции help(S.method), чтобы познакомиться с описанием   
метода method для любого объекта строки S.  
Обратите внимание, что ни один из строковых методов не поддерживает шабло-  
ны, – для обработки текста с использованием шаблонов необходимо использо-

Строковые методы   
233  
вать модуль re, входящий в состав стандартной библиотеки языка Python, – до-  
полнительный инструмент, начальные сведения о котором приводились в гла-  
ве 4, а полное его обсуждение выходит далеко за рамки этой книги (один при-  
мер приводится в конце главы 36). Тем не менее, несмотря на это ограничение,   
строковые методы иногда оказываются эффективнее, чем функции модуля re.  
Оригинальный модуль string (был убран в версии 3.0)  
История развития строковых методов достаточно запутана. В течение перво-  
го десятилетия существования Python в состав стандартной библиотеки вхо-  
дил модуль string, который содержал функции, во многом напоминающие со-  
временные строковые методы. В ответ на требования пользователей в версии   
Python 2.0 эти функции были преобразованы в методы строковых объектов.   
Однако, из-за большого объема программного кода, уже написанного к тому   
времени, оригинальный модуль string был сохранен для обеспечения обратной   
совместимости.  
Ныне вы должны использовать не оригинальный модуль string, а применять   
только строковые методы. Фактически, как предполагается, оригинальные   
строковые функции, соответствующие методам, были полностью убраны из со-  
става стандартной библиотеки языка Python в версии 3.0. Однако, поскольку   
использование модуля string еще можно встретить в старом программном коде,   
мы коротко рассмотрим его.  
В версии Python 2.6 по-прежнему существует два способа использования рас-  
ширенных операций над строками: посредством вызова методов и вызовом   
функций модуля string, которым в качестве аргумента передается объект стро-  
ки. Например, допустим, что переменная X ссылается на объект строки, тогда   
вызов метода объекта будет выглядеть следующим образом:  
X.method(arguments)  
что эквивалентно вызову аналогичной функции из модуля string (представим,   
что этот модуль уже был импортирован):  
string.method(X, arguments)  
В следующем примере демонстрируется порядок использования метода:  
>>> S = ‘a+b+c+’  
>>> x = S.replace(‘+’, ‘spam’)  
>>> x  
‘aspambspamcspam’  
Чтобы выполнить ту же самую операцию с помощью модуля string, необходимо   
импортировать модуль (по крайней мере, один раз) и передать функции объект:  
>>> import string  
>>> y = string.replace(S, ‘+’, ‘spam’)  
>>> y  
‘aspambspamcspam’  
Поскольку подход, основанный на применении модуля, был стандартом на про-  
тяжении многих лет, а строки являются одним из центральных компонентов   
большинства программ, при анализе существующих программ вам наверняка   
встретятся оба варианта работы со строками.

234   
Глава 7. Строки   
И тем не менее, при создании новых программ вместо функций устаревшего   
модуля вы должны использовать строковые методы. Для этого есть и другие   
серьезные основания, помимо того, что многие функции модуля string были   
исключены из состава стандартной библиотеки в версии Python 3.0. Одна из та-  
Python 3.0. Одна из та-  
 3.0. Одна из та-  
ких причин – схема вызова модуля требует от вас импортировать модуль string   
(при использовании строковых методов ничего импортировать не нужно).   
Другая причина заключается в том, что при использовании функций модуля   
с клавиатуры приходится вводить на несколько символов больше (когда мо-  
дуль загружается с использованием инструкции import, а не from). И наконец,   
функции модуля выполняются несколько медленнее, чем методы (в настоящее   
время функции модуля отображаются на вызовы методов, в результате чего   
производятся дополнительные вызовы).  
Оригинальный модуль string без функций, эквивалентных строковым мето-  
дам, остался в составе Python 3.0, потому что в нем присутствуют дополни-  
тельные средства работы со строками, включая предопределенные строковые   
константы и систему шаблонных объектов (которая не описывается в этой кни-  
ге из-за большой сложности – за дополнительной информацией обращайтесь   
к руководству по библиотеке языка Python). Если вы хотите предусмотреть   
возможность переноса своих программ, написанных для версии 2.6, на версию   
Python 3.0, вы должны смотреть на базовые строковые функции как на при-  
зраки прошлого.  
Выражения форматирования строк   
Уже с помощью представленных к этому моменту строковых методов и опера-  
ций над последовательностями можно сделать немало, но, кроме того, в арсе-  
нале языка Python имеются дополнительные возможности обработки строк –   
операции форматирования строк позволяют выполнять подстановку в строки   
значений различных типов за одно действие. Про операции форматирования   
нельзя сказать, что они в каких-то случаях предоставляют единственно воз-  
можный способ решения каких-либо задач, но их удобно использовать, напри-  
мер когда необходимо отформатировать текст для вывода на экран. Благодаря   
богатству новых идей, появляющихся в мире языка Python, операции форма-  
Python, операции форма-  
, операции форма-  
тирования строк могут выполняться двумя способами:  
Выражения форматирования строк  
Первоначально существовавший способ – он основан на модели функции   
printf из языка C и широко используется в существующих программах.  
Метод форматирования строк   
Более новый способ, появившийся в версиях Python 2.6 и 3.0, – более уни-  
Python 2.6 и 3.0, – более уни-  
 2.6 и 3.0, – более уни-  
кальный для языка Python, возможности которого в значительной степени   
пересекаются с возможностями выражений форматирования.  
Из-за новизны метода форматирования существует вероятность, что некоторые   
из его особенностей со временем будут переведены в разряд нерекомендуемых.   
Выражения форматирования, скорее всего, станут нерекомендуемыми в более   
поздних выпусках Python, хотя это во многом будет зависеть от предпочтений   
активных программистов на языке Python. Но так как оба способа являются   
лишь вариациями на одну и ту же тему, то в настоящее время можно смело ис-  
пользовать любой из них. Выражения форматирования строк появились в язы-  
ке значительно раньше, поэтому мы начнем с них.

Выражения форматирования строк   
235  
В языке Python имеется двухместный оператор %, предназначенный для рабо-  
ты со строками (вы можете вспомнить, что для чисел он является оператором   
деления по модулю, или получения остатка от деления). Когда этот оператор   
применяется к строкам, он обеспечивает простой способ форматирования зна-  
чений, согласно заданной строке формата. Проще говоря, оператор % обеспе-  
чивает возможность компактной записи программного кода, выполняющего   
множественную подстановку строк, позволяя избавиться от необходимости   
конструирования отдельных фрагментов строки по отдельности с последую-  
щим их объединением.  
Чтобы отформатировать строку, требуется:  
1. Слева от оператора % указать строку формата, содержащую один или более   
спецификаторов формата, каждый из которых начинается с символа % (на-  
пример, %d).  
2. Справа от оператора % указать объект (или объекты, в виде кортежа), значе-  
ние которого должно быть подставлено на место спецификатора (или специ-  
фикаторов) в левой части выражения.  
Например, в примере форматирования строки, который приводился ранее   
в этой главе, мы видели, что целое число 1 замещает спецификатор %d в строке   
формата, расположенной в левой части выражения, а строка “dead” замещает   
спецификатор %s. В результате получается новая строка, которая содержит эти   
две подстановки.  
>>> ‘That is %d %s bird!’ % (1, ‘dead’) # Выражение форматирования  
That is 1 dead bird!  
Строго говоря, выражения форматирования строк не являются абсолютно   
необходимыми – все, что можно сделать с их помощью, точно так же можно   
сделать с помощью серии преобразований и операций конкатенации. Однако   
операция форматирования позволяет объединить множество шагов в одной   
инструкции. Мощные возможности этой операции заслуживают того, чтобы   
рассмотреть еще несколько примеров:  
>>> exclamation = “Ni”  
>>> “The knights who say %s!” % exclamation  
‘The knights who say Ni!’  
   
>>> “%d %s %d you” % (1, ‘spam’, 4)  
‘1 spam 4 you’  
   
>>> “%s -- %s -- %s” % (42, 3.14159, [1, 2, 3])  
‘42 -- 3.14159 -- [1, 2, 3]’  
В первом примере строка “Ni” внедряется в целевую строку слева, замещая   
спецификатор %s. Во втором примере в целевую строку вставляются три зна-  
чения. Обратите внимание: когда вставляется более одного значения, в правой   
части выражения их необходимо сгруппировать с помощью круглых скобок   
(то есть создать из них кортеж). Оператор форматирования % ожидает получить   
справа либо один объект, либо кортеж объектов.  
В третьем примере также вставляются три значения – целое число, веществен-  
ное число и объект списка, но обратите внимание, что в левой части выражения   
всем значениям соответствует спецификатор %s, который соответствует опера-  
ции преобразования в строку. Объекты любого типа могут быть преобразованы   
в строку (это происходит, например, при выводе на экран), поэтому для любого

236   
Глава 7. Строки   
объекта может быть указан спецификатор %s. Вследствие этого вам не придет-  
ся выполнять специальное форматирование, в большинстве случаев вам доста-  
точно будет знать только о существовании спецификатора %s.   
Имейте в виду, что выражение форматирования всегда создает новую строку,   
а не изменяет строку, расположенную в левой части. Поскольку строки явля-  
ются неизменяемыми, этот оператор вынужден работать именно таким спо-  
собом. Как уже говорилось ранее, если вам требуется сохранить полученный   
результат, присвойте его переменной.  
Дополнительные возможности форматирования строки  
Для реализации более сложного форматирования в выражениях форматиро-  
вания можно использовать любые спецификаторы формата, представленные   
в табл. 7.4. Большинство из них окажутся знакомы программистам, исполь-  
зовавшим язык C, потому что операция форматирования строк в языке Python   
поддерживает все наиболее типичные спецификаторы формата, которые допу-  
скается использовать в функции printf языка C (в отличие от которой выраже-  
ние в языке Python возвращает результат, а не выводит его на экран). Некото-  
рые спецификаторы из табл. 7.4 предоставляют альтернативные способы фор-  
матирования данных одного и того же типа, например %e, %f и %g обеспечивают   
альтернативные способы форматирования вещественных чисел.  
Таблица 7.4. Спецификаторы формата  
Спецификатор  
Назначение  
s  
Строка (для объекта любого другого типа будет выполнен   
вызов функции str(X), чтобы получить строковое представ-  
ление объекта)  
r  
s, но использует функцию repr, а не str  
c  
Символ  
d  
Десятичное (целое) число  
i  
Целое число  
u  
То же, что и d (устарел: больше не является представлени-  
ем целого без знака)  
o  
Восьмеричное целое число  
x  
Шестнадцатеричное целое число  
X  
x, но шестнадцатеричные цифры возвращаются в верхнем   
регистре  
e  
Вещественное число в экспоненциальной форме  
E  
e, но алфавитные символы возвращаются в верхнем   
регистре  
f  
Вещественное число в десятичном представлении  
F  
Вещественное число в десятичном представлении  
g  
Вещественное число e или f

Выражения форматирования строк   
237  
Спецификатор  
Назначение  
G  
Вещественное число E или а  
%  
Символ %  
Фактически спецификаторы формата в левой части выражения поддержива-  
ют целый набор операций преобразования с достаточно сложным собственным   
синтаксисом. В общем виде синтаксис использования спецификатора формата   
выглядит следующим образом:  
%[(name)][flags][width][.precision]code  
Символ спецификатора формата (code) из табл. 7.4 располагается в самом кон-  
це. Между символом % и символом спецификатора можно добавлять следую-  
щую информацию: ключ в словаре (name)� список флагов (flags), которые могут   
определять, например, признак выравнивания (-), знак числа (+), наличие ве-  
дущих нулей (0)� общую ширину поля и число знаков после десятичной точки   
и многое другое. Параметры формата width и precision могут также принимать   
значение \*, чтобы показать, что фактические значения этих параметров долж-  
ны извлекаться из следующего элемента в списке входных значений.  
Полное описание синтаксиса спецификаторов формата вы найдете в стандарт-  
ном руководстве по языку Python, а сейчас для демонстрации наиболее типич-  
ных случаев их использования приведем несколько примеров. В первом при-  
мере сначала применяется форматирование целого числа с параметрами по   
умолчанию, а затем целое число выводится в поле шириной в шесть символов,   
с выравниванием по левому краю и с дополнением ведущими нулями:  
>>> x = 1234  
>>> res = “integers: ...%d...%-6d...%06d” % (x, x, x)  
>>> res  
‘integers: ...1234...1234 ...001234’  
Спецификаторы %e, %f и %g отображают вещественные числа разными способа-  
ми, как демонстрируется в следующем примере:  
>>> x = 1.23456789  
>>> x  
1.2345678899999999  
   
>>> ‘%e | %f | %g’ % (x, x, x)  
‘1.234568e+00 | 1.234568 | 1.23457’  
   
>>> ‘%E’ % x  
‘1.234568E+00’   
Для вещественных чисел можно реализовать дополнительные эффекты форма-  
тирования, указав необходимость выравнивания по левому краю, дополнение   
ведущими нулями, знак числа, ширину поля и число знаков после десятичной   
точки. Для простых задач можно было бы использовать простые функции пре-  
образования чисел в строки с применением выражения форматирования или   
встроенной функции str, продемонстрированной ранее:  
>>> ‘%-6.2f | %05.2f | %+06.1f’ % (x, x, x)  
‘1.23 | 01.23 | +001.2’

238   
Глава 7. Строки   
>>> “%s” % x, str(x)  
(‘1.23456789’, ‘1.23456789’)  
Если ширина поля и количество знаков после десятичной точки заранее не из-  
вестны, их можно вычислять во время выполнения, а в строке формата вместо   
фактических значений использовать символ \*, чтобы указать интерпретатору,   
что эти значения должны извлекаться из очередного элемента в списке вход-  
ных значений, справа от оператора %. Число 4 в кортеже определяет количество   
знаков после десятичной точки:  
>>> ‘%f, %.2f, %.\*f’ % (1/3.0, 1/3.0, 4, 1/3.0)  
‘0.333333, 0.33, 0.3333’  
Если вас заинтересовала эта особенность, поэкспериментируйте самостоятель-  
но с этими примерами, чтобы получить более полное представление.  
Форматирование строк из словаря  
Операция форматирования позволяет также использовать в спецификаторах   
формата ссылки на ключи словаря, который указывается в правой части вы-  
ражения, для извлечения соответствующих значений. Мы пока немного гово-  
рили о словарях, поэтому следующий пример демонстрирует самый простой   
случай:  
>>> “%(n)d %(x)s” % {“n”:1, “x”:”spam”}  
‘1 spam’  
В данном случае (n) и (x) в строке формата ссылаются на ключи в словаре в пра-  
вой части выражения и служат для извлечения соответствующих им значе-  
ний. Этот прием часто используется в программах, создающих код разметки   
HTML или XML, – вы можете построить словарь значений и затем подставить   
их все одним выражением форматирования, которое использует ключи:  
>>> reply = “”” # Шаблон с замещаемыми спецификаторами формата  
Greetings...  
Hello %(name)s!  
Your age squared is %(age)s  
“””  
>>> values = {‘name’: ‘Bob’, ‘age’: 40} # Подготовка фактических значений  
>>> print reply % values # Подстановка значений  
   
Greetings...  
Hello Bob!  
Your age squared is 40  
Этот способ также часто используется в комбинации со встроенной функцией   
vars, которая возвращает словарь, содержащий все переменные, существую-  
щие на момент ее вызова:  
>>> food = ‘spam’  
>>> age = 40  
>>> vars()  
{‘food’: ‘spam’, ‘age’: 40, ...и еще множество других... }  
Если задействовать эту функцию в правой части оператора форматирования,   
можно отформатировать значения, обращаясь к ним по именам переменных   
(то есть по ключам словаря):

Метод форматирования строк   
239  
>>> “%(age)d %(food)s” % vars()  
‘40 spam’  
Словари во всех подробностях мы будем изучать в главе 8. А в главе 5 вы найде-  
те примеры использования спецификаторов %x и %o для преобразования значе-  
ний в шестнадцатеричное и восьмеричное строковое представление.  
Метод форматирования строк  
Как уже упоминалось выше, в Python 2.6 и 3.0 появился новый способ форма-  
Python 2.6 и 3.0 появился новый способ форма-  
 2.6 и 3.0 появился новый способ форма-  
тирования строк, более близкий по духу к идеям языка Python. В отличие от   
выражений форматирования, метод форматирования не так сильно опирается   
на модель функции «printf» языка C и имеет более подробный и явный синтак-  
printf» языка C и имеет более подробный и явный синтак-  
» языка C и имеет более подробный и явный синтак-  
сис. С другой стороны, новый способ по-прежнему опирается на некоторые кон-  
цепции функции «printf», такие как символы и параметры спецификаторов   
формата. Кроме того, возможности метода форматирования в значительной   
степени пересекаются с возможностями выражений форматирования (иногда,   
правда, за счет большего объема программного кода) и могут выполнять столь   
же сложные операции форматирования. Вследствие этого сложно сказать, ка-  
кой из двух способов лучше, поэтому для большинства программистов совсем   
не лишним будет знакомство с обоими.  
Основы  
В двух словах, новый метод format объектов строк, появившийся в версиях 2.6   
и 3.0 (и выше), использует строку, относительно которой он вызывается, как   
шаблон и принимает произвольное количество аргументов, представляющих   
значения для подстановки. Фигурные скобки внутри строки шаблона ис-  
пользуются для обозначения замещаемых спецификаторов и их параметров,   
которые могут определять порядковые номера позиционных аргументов (на-  
пример, {1}) или имена именованных аргументов (например, {food}). В главе 18   
вы узнаете, что аргументы могут передаваться функциям и методам в виде по-  
зиционных и именованных аргументов, а язык Python предоставляет возмож-  
Python предоставляет возмож-  
 предоставляет возмож-  
ность создавать функции и методы, способные принимать произвольное коли-  
чество позиционных и именованных аргументов. Ниже приводится несколько   
примеров использования метода format в Python 2.6 и 3.0:  
>>> template = ‘{0}, {1} and {2}’ # Порядковые номера позиционных аргументов  
>>> template.format(‘spam’, ‘ham’, ‘eggs’)  
‘spam, ham and eggs’  
   
>>> template = ‘{motto}, {pork} and {food}’ # Имена именованных аргументов  
>>> template.format(motto=’spam’, pork=’ham’, food=’eggs’)  
‘spam, ham and eggs’  
   
>>> template = ‘{motto}, {0} and {food}’ # Оба варианта  
>>> template.format(‘ham’, motto=’spam’, food=’eggs’)  
‘spam, ham and eggs’  
Естественно, строка шаблона также может быть литералом, кроме того, сохра-  
няется возможность подстановки значений объектов любых типов:  
>>> ‘{motto}, {0} and {food}’.format(42, motto=3.14, food=[1, 2])  
‘3.14, 42 and [1, 2]’

240   
Глава 7. Строки   
Как и при использовании оператора форматирования % и других строковых ме-  
тодов, метод format создает и возвращает новый объект строки, который мож-  
но тут же вывести на экран или сохранить для последующего использования   
(не забывайте, что строки являются неизменяемыми объектами, поэтому ме-  
тод format вынужден создавать новый объект). Возможность форматирования   
строк может использоваться не только для их отображения:  
>>> X = ‘{motto}, {0} and {food}’.format(42, motto=3.14, food=[1, 2])  
>>> X  
‘3.14, 42 and [1, 2]’  
>>> X.split(‘ and ‘)  
[‘3.14, 42’, ‘[1, 2]’]  
>>> Y = X.replace(‘and’, ‘but under no circumstances’)  
>>> Y  
‘3.14, 42 but under no circumstances [1, 2]’  
Использование ключей, атрибутов и смещений  
Как и оператор форматирования %, метод format обладает дополнительными   
возможностями. Например, в строках формата допускается ссылаться на име-  
на атрибутов объектов и ключи словарей, – как и в привычном синтаксисе   
языка Python, квадратные скобки обозначают ключи словаря, а точка приме-  
Python, квадратные скобки обозначают ключи словаря, а точка приме-  
, квадратные скобки обозначают ключи словаря, а точка приме-  
няется для организации доступа к атрибутам объектов, на которые ссылаются   
позиционные или именованные спецификаторы. В первом примере ниже де-  
монстрируется подстановка значения ключа «spam» словаря и значения атри-  
spam» словаря и значения атри-  
» словаря и значения атри-  
бута «platform» объекта импортированного модуля sys, которые передаются   
как позиционные аргументы. Во втором примере делается то же самое, но на   
этот раз объекты для подстановки передаются в виде именованных аргумен-  
тов:  
>>> import sys  
   
>>> ‘My {1[spam]} runs {0.platform}’.format(sys, {‘spam’: ‘laptop’})  
‘My laptop runs win32’  
   
>>> ‘My {config[spam]} runs {sys.platform}’.format(sys=sys,  
 config={‘spam’: ‘laptop’})  
‘My laptop runs win32’  
В квадратных скобках в строках формата можно также указывать смещение   
от начала списка (или любой другой последовательности), но при этом допуска-  
ется использовать только положительные смещения, поэтому данная возмож-  
ность не является достаточно универсальной, как можно было бы подумать.   
Как и в операторе %, чтобы обратиться по отрицательному смещению, получить   
срез или вставить результат произвольного выражения, данные для подстанов-  
ки должны быть подготовлены отдельными выражениями, за пределами стро-  
ки формата:  
>>> somelist = list(‘SPAM’)  
>>> somelist  
[‘S’, ‘P’, ‘A’, ‘M’]  
   
>>> ‘first={0[0]}, third={0[2]}’.format(somelist)  
‘first=S, third=A’

Метод форматирования строк   
241  
>>> ‘first={0}, last={1}’.format(somelist[0], somelist[-1]) # Если [-1]  
‘first=S, last=M’ # использовать внутри строки формата, это приведет к ошибке  
   
>>> parts = somelist[0], somelist[-1], somelist[1:3] # Если [1:3]  
>>> ‘first={0}, last={1}, middle={2}’.format(\*parts) # использовать внутри   
“first=S, last=M, middle=[‘P’, ‘A’]” # строки формата, это приведет к ошибке  
Специальные приемы форматирования  
Еще одно сходство с оператором форматирования % состоит в том, что за счет ис-  
пользования дополнительных синтаксических конструкций форматирования   
имеется возможность выполнять более точное форматирование. Вслед за иден-  
тификатором символа подстановки, через двоеточие, можно указать специфи-  
катор формата, определяющий ширину поля вывода, выравнивание и код типа   
значения. Ниже приводится формальный синтаксис спецификатора формата:  
{fieldname!conversionflag:formatspec}  
Поля спецификатора имеют следующий смысл:  
 •  
fieldname – порядковый номер или имя именованного аргумента, за кото-  
рым может следовать необязательное имя «.name» атрибута или индекс   
«[index]» элемента.  
 •  
conversionflag – может быть r, s или a, которые определяют применение   
к значению встроенной функции repr, str или ascii соответственно.  
 •  
formatspec – определяет способ представления значения, описывает такие   
характеристики представления, как ширина поля вывода, выравнивание,   
дополнение, количество знаков после десятичной точки и так далее, и за-  
вершается необязательным кодом типа значения.  
Поле formatspec, следующее за двоеточием, в общем виде имеет следующий   
синтаксис (квадратные скобки окружают необязательные компоненты и не   
имеют отношения к синтаксису поля):  
[[fill]align][sign][#][0][width][.precision][typecode]  
В поле align может указываться символ <, >, = или ^, обозначающий выравни-  
вание по левому или по правому краю, дополнение после символа знака числа   
или выравнивание по центру соответственно. Спецификатор формата format-  
spec может также содержать вложенные {} строки форматирования с именами   
полей, чтобы извлекать значения из списка аргументов динамически (практи-  
чески так же, как символ \* в операторе форматирования).  
Подробное описание синтаксиса и полный список допустимых кодов типа вы   
найдете в руководстве по стандартной библиотеке языка Python – этот список   
практически полностью совпадает со списком спецификаторов, используемых   
в операторе форматирования % и перечисленных в табл. 7.4. Дополнительно ме-  
тод format позволяет использовать код типа «b» для отображения целых чисел   
в двоичном формате (эквивалент вызову встроенной функции bin). Кроме того,   
код типа «%» используется для отображения символа процента, а для ото-  
бражения десятичных целых чисел допускается использовать только код «d»   
(коды «i» и «u» считаются недопустимыми).  
В следующем примере спецификатор {0:10} предписывает вывести значение   
первого позиционного аргумента в поле шириной 10 символов. Спецификатор

242   
Глава 7. Строки   
{1:<10} предписывает вывести значение второго позиционного аргумента в поле   
шириной 10 символов, с выравниванием по левому краю, а спецификатор   
{0.platform:>10} предписывает вывести значение атрибута platform первого пози-  
ционного аргумента в поле шириной 10 символов, с выравниванием по правому   
краю:  
>>> ‘{0:10} = {1:10}’.format(‘spam’, 123.4567)  
‘spam = 123.457’  
   
>>> ‘{0:>10} = {1:<10}’.format(‘spam’, 123.4567)  
‘ spam = 123.457 ‘  
   
>>> ‘{0.platform:>10} = {1[item]:<10}’.format(sys, dict(item=’laptop’))  
‘ win32 = laptop ‘  
Для вывода вещественных чисел метод format поддерживает те же самые коды   
типов и параметры форматирования, что и оператор %. Так, в следующем при-  
мере спецификатор {2:g} предписывает вывести третий аргумент, отформати-  
рованный в соответствии с представлением вещественных чисел по умолча-  
нию, предусмотренным кодом «g». Спецификатор {1:.2f} предписывает исполь-  
зовать формат «f» представления вещественных чисел с двумя знаками после   
десятичной точки, а спецификатор {2:06.2f} дополнительно ограничивает ши-  
рину поля вывода 6 символами и предписывает дополнить число нулями слева:  
>>> ‘{0:e}, {1:.3e}, {2:g}’.format(3.14159, 3.14159, 3.14159)  
‘3.141590e+00, 3.142e+00, 3.14159’  
   
>>> ‘{0:f}, {1:.2f}, {2:06.2f}’.format(3.14159, 3.14159, 3.14159)  
‘3.141590, 3.14, 003.14’  
Метод format поддерживает также возможность вывода чисел в шестнадцате-  
ричном, восьмеричном и двоичном представлениях. Фактически строка фор-  
мата может служить альтернативой использованию некоторых встроенных   
функций:  
>>> ‘{0:X}, {1:o}, {2:b}’.format(255, 255, 255) # Шестнадцатеричное,   
‘FF, 377, 11111111’ # восьмеричное и двоичное представление  
   
>>> bin(255), int(‘11111111’, 2), 0b11111111 # Другие способы работы с   
(‘0b11111111’, 255, 255) # двоичным представлением  
   
>>> hex(255), int(‘FF’, 16), 0xFF # Другие способы работы с  
(‘0xff’, 255, 255) # шестнадцатеричным представлением  
   
>>> oct(255), int(‘377’, 8), 0o377, 0377 # Другие способы работы с  
(‘0377’, 255, 255, 255) # восьмеричным представлением  
 # 0377 допустимо только в 2.6, но не в 3.0!  
Параметры форматирования можно указывать непосредственно в строке фор-  
мата или динамически извлекать из списка аргументов, с помощью синтаксиса   
вложенных конструкций, практически так же, как с помощью символа звез-  
дочки в операторе форматирования %:  
>>> ‘{0:.2f}’.format(1 / 3.0) # Параметры определены непосредственно   
‘0.33’ # в строке формата  
>>> ‘%.2f’ % (1 / 3.0)  
‘0.33’  
   
>>> ‘{0:.{1}f}’.format(1 / 3.0, 4) # Значение извлекается из списка аргументов

Метод форматирования строк   
243  
‘0.3333’  
>>> ‘%.\*f’ % (4, 1 / 3.0) # Как то же самое делается в выражениях  
‘0.3333’  
Наконец, в Python 2.6 и 3.0 появилась новая встроенная функция format, ко-  
торая может использоваться для форматирования одиночных значений. Она   
может рассматриваться как более компактная альтернатива методу format   
и напоминает способ форматирования единственного значения с помощью опе-  
ратора %:   
>>> ‘{0:.2f}’.format(1.2345) # Строковый метод  
‘1.23’  
>>> format(1.2345, ‘.2f’) # Встроенная функция  
‘1.23’  
>>> ‘%.2f’ % 1.2345 # Выражение форматирования  
‘1.23’  
С технической точки зрения, встроенная функция format вызывает метод \_\_  
format\_\_ объекта, который в свою очередь вызывает метод str.format каждого   
форматируемого элемента. Однако эта функция имеет не такой компактный   
синтаксис, как оригинальный оператор %, что ведет нас к следующему разделу.  
Сравнение с оператором форматирования %   
Если вы внимательно прочитали предыдущие разделы, вы могли заметить,   
что, по крайней мере, ссылки на позиционные аргументы и ключи словарей   
в строковом методе format выглядят почти так же, как в операторе форматиро-  
вания %, особенно когда используются дополнительные параметры форматиро-  
вания и коды типов значений. В действительности, в общем случае применение   
оператора форматирования выглядит проще, чем вызов метода format, особенно   
при использовании универсального спецификатора формата %s:  
print(‘%s=%s’ % (‘spam’, 42)) # 2.X+ выражение форматирования  
print(‘{0}={1}’.format(‘spam’, 42)) # 3.0 (и 2.6) метод format   
Как вы увидите чуть ниже, в более сложных случаях оба способа имеют почти   
одинаковую сложность (сложные задачи обычно сложны сами по себе, неза-  
висимо от используемого подхода), поэтому некоторые считают format в значи-  
тельной степени избыточным.  
С другой стороны, метод format предлагает дополнительные потенциальные   
преимущества. Например, оператор % не позволяет использовать именованные   
аргументы, ссылки на атрибуты и выводить числа в двоичном представлении,   
хотя возможность использования словарей в операторе % помогает добиться тех   
же целей. Чтобы увидеть сходные черты между этими двумя приемами, срав-  
ните следующие примеры использования оператора % с аналогичными приме-  
рами использования метода format, представленными выше:  
# Основы: с оператором % вместо метода format()  
>>> template = ‘%s, %s, %s’  
>>> template % (‘spam’, ‘ham’, ‘eggs’) # Позиционные параметры  
‘spam, ham, eggs’  
   
>>> template = ‘%(motto)s, %(pork)s and %(food)s’  
>>> template % dict(motto=’spam’, pork=’ham’, food=’eggs’) # Ключи словаря  
‘spam, ham and eggs’

244   
Глава 7. Строки   
>>> ‘%s, %s and %s’ % (3.14, 42, [1, 2]) # Произвольные типы  
‘3.14, 42 and [1, 2]’  
   
# Использование ключей, атрибутов и смещений  
   
>>> ‘My %(spam)s runs %(platform)s’ % {‘spam’: ‘laptop’, ‘platform’: sys.platform}  
‘My laptop runs win32’  
   
>>> ‘My %(spam)s runs %(platform)s’ % dict(spam=’laptop’, platform=sys.platform)  
‘My laptop runs win32’  
   
>>> somelist = list(‘SPAM’)  
>>> parts = somelist[0], somelist[-1], somelist[1:3]  
>>> ‘first=%s, last=%s, middle=%s’ % parts  
“first=S, last=M, middle=[‘P’, ‘A’]”  
Когда требуется добиться более сложного форматирования, эти два подхода   
становятся почти равными в смысле сложности, хотя, если сравнить следую-  
щие примеры использования оператора % с аналогичными примерами исполь-  
зования метода format, представленными выше, опять можно заметить, что   
оператор % выглядит несколько проще и компактнее:  
# Специальные приемы форматирования  
>>> ‘%-10s = %10s’ % (‘spam’, 123.4567)  
‘spam = 123.4567’  
   
>>> ‘%10s = %-10s’ % (‘spam’, 123.4567)  
‘ spam = 123.4567 ‘  
   
>>> ‘%(plat)10s = %(item)-10s’ % dict(plat=sys.platform, item=’laptop’)  
‘ win32 = laptop ‘  
   
# Вещественные числа  
   
>>> ‘%e, %.3e, %g’ % (3.14159, 3.14159, 3.14159)  
‘3.141590e+00, 3.142e+00, 3.14159’  
   
>>> ‘%f, %.2f, %06.2f’ % (3.14159, 3.14159, 3.14159)  
‘3.141590, 3.14, 003.14’  
   
# Числа в шестнадцатеричном и восьмеричном представлениях, но не в двоичном  
>>> ‘%x, %o’ % (255, 255)  
‘ff, 377’  
Метод format имеет ряд дополнительных особенностей, которые не поддержи-  
ваются оператором %, но даже когда требуется реализовать еще более сложное   
форматирование, оба подхода выглядят примерно одинаковыми в смысле слож-  
ности. Например, ниже демонстрируются два подхода к достижению одина-  
ковых результатов – при использовании параметров, определяющих ширину   
полей и выравнивание, а также различных способов обращения к атрибутам:   
# В обоих случаях фактические значения определяются непосредственно в операции  
   
>>> import sys  
   
>>> ‘My {1[spam]:<8} runs {0.platform:>8}’.format(sys, {‘spam’: ‘laptop’})  
‘My laptop runs win32’

Метод форматирования строк   
245  
>>> ‘My %(spam)-8s runs %(plat)8s’ % dict(spam=’laptop’, plat=sys.platform)  
‘My laptop runs win32’  
На практике фактические значения редко определяются непосредственно   
в операции форматирования, как здесь. Чаще эти значения вычисляются за-  
ранее (например, чтобы затем подставить их все сразу в шаблон разметки   
HTML). Когда мы начинаем учитывать общепринятую практику в своих при-  
). Когда мы начинаем учитывать общепринятую практику в своих при-  
мерах, сравнение метода format и оператора % становится еще более наглядным   
(как вы узнаете в главе 18, аргумент \*\*data в вызове метода – это специальный   
синтаксис распаковывания ключей и значений словарей в отдельные пары   
«name�value» именованных аргументов, благодаря чему появляется возмож-  
name�value» именованных аргументов, благодаря чему появляется возмож-  
�value» именованных аргументов, благодаря чему появляется возмож-  
value» именованных аргументов, благодаря чему появляется возмож-  
» именованных аргументов, благодаря чему появляется возмож-  
ность обращаться к ним по именам в строке формата):  
# В обоих случаях используются данные, собранные предварительно  
   
>>> data = dict(platform=sys.platform, spam=’laptop’)  
   
>>> ‘My {spam:<8} runs {platform:>8}’.format(\*\*data)  
‘My laptop runs win32’  
   
>>> ‘My %(spam)-8s runs %(platform)8s’ % data  
‘My laptop runs win32’  
Как обычно, сообществу пользователей языка Python еще предстоит решить,   
что использовать лучше, – оператор %, метод format или их комбинацию. По-  
экспериментируйте с обоими способами самостоятельно, чтобы получить пред-  
ставление об имеющихся возможностях, а за дополнительными подробностя-  
ми обращайтесь к руководству по стандартной библиотеке Python 2.6 и 3.0.  
Улучшения метода format в версии Python 3.1: В будущей вер-  
сии 3.1 (когда писалась эта глава, она находилась еще в состоя-  
нии альфа-версии) будет добавлена возможность указывать   
символ-разделитель разрядов в десятичных числах, что позво-  
лит вставлять запятые между трехзначными группами цифр.   
Для этого достаточно просто добавить запятую перед кодом типа   
значения, как показано ниже:  
>>> ‘{0:d}’.format(999999999999)  
‘999999999999’  
   
>>> ‘{0:,d}’.format(999999999999)  
‘999,999,999,999’  
Кроме того, в Python 3.1 имеется возможность не указывать явно   
порядковые номера позиционных аргументов. В этом случае ар-  
гументы из списка будут выбираться последовательно, однако   
использование этой особенности может свести к нулю основное   
преимущество метода format, как это описывается в следующем   
разделе:  
>>> ‘{:,d}’.format(999999999999)  
‘999,999,999,999’  
   
>>> ‘{:,d} {:,d}’.format(9999999, 8888888)  
‘9,999,999 8,888,888’

246   
Глава 7. Строки   
>>> ‘{:,.2f}’.format(296999.2567)  
‘296,999.26’  
Официально в этой книге не рассматривается версия 3.1, поэто-  
му данное примечание следует рассматривать только как пред-  
варительное. Дополнительно в версии Python 3.1 будут ликви-  
Python 3.1 будут ликви-  
 3.1 будут ликви-  
дированы основные проблемы производительности, наблюдаю-  
щиеся в версии 3.0, связанные с файловыми операциями ввода/  
вывода, и делающие версию 3.0 непривлекательной для боль-  
3.0 непривлекательной для боль-  
непривлекательной для боль-  
 для боль-  
для боль-  
 боль-  
боль-  
шинства применений. Дополнительные подробности вы найдете   
в примечаниях к выпуску 3.1. Ознакомьтесь также со сценарием   
formats.py в главе 24, где вы найдете приемы добавления запя-  
тых и форматирования денежных сумм вручную, которые мож-  
но использовать, пока не вышла в свет версия Python 3.1.  
Когда может пригодиться новый метод format?  
Теперь, когда я потратил столько слов, чтобы сравнить и противопоставить два   
способа форматирования, я должен объяснить, когда может пригодиться ме-  
тод format. В двух словах: несмотря на то, что использование метода формати-  
рования требует вводить больше программного кода, тем не менее:  
 •  
Он обладает некоторыми особенностями, отсутствующими в операторе фор-  
матирования %  
 •  
Позволяет более явно ссылаться на аргументы  
 •  
Вместо символа оператора используется более говорящее имя метода  
 •  
Не различает случаи, когда выполняется подстановка одного или несколь-  
ких значений  
На сегодняшний день доступны оба приема, и оператор форматирования по-  
прежнему находит широкое применение, тем не менее метод format в конечном   
итоге может вытеснить его. Но пока у вас есть выбор, поэтому, прежде чем дви-  
нуться дальше, подробнее остановимся на некоторых отличиях.  
Дополнительные возможности  
Метод format поддерживает дополнительные возможности, недоступные при   
использовании оператора %, такие как отображение чисел в двоичном формате   
и (появившаяся в Python 3.1) возможность разделения групп разрядов. Кроме   
того, метод format позволяет напрямую обращаться к ключам словарей и атри-  
бутам объектов в строке формата. Однако, как мы уже видели, при использова-  
нии оператора % того же эффекта можно добиться другими способами:  
>>> ‘{0:b}’.format((2 \*\* 16) -1)  
‘1111111111111111’  
   
>>> ‘%b’ % ((2 \*\* 16) -1)  
ValueError: unsupported format character ‘b’ (0x62) at index 1  
   
>>> bin((2 \*\* 16) -1)  
‘0b1111111111111111’  
   
>>> ‘%s’ % bin((2 \*\* 16) -1)[2:]  
‘1111111111111111’

Метод форматирования строк   
247  
Смотрите также примеры в предыдущем разделе, где проводилось сравнение   
способов, основанных на использовании ключей словаря в операторе % и ссы-  
лок на атрибуты объектов в методе format, которые в значительной степени вы-  
глядят, как вариации на одну и ту же тему.  
Неявные ссылки на значения  
Один из случаев, когда реализация на основе метода format выглядит более по-  
нятной, – подстановка большого количества значений в строку формата. Так,   
в примере lister.py, который приводится в главе 30, выполняется подстановка   
шести значений в одну строку формата� в этом случае метки {i} в вызове метода   
format читаются проще, чем спецификаторы %s в выражении форматирования:  
‘\n%s<Class %s, address %s:\n%s%s%s>\n’ % (...) # Выражение  
   
‘\n{0}<Class {1}, address {2}:\n{3}{4}{5}>\n’.format(...) # Метод  
С другой стороны, использование ключей словаря в операторе % позволяет   
в значительной степени ликвидировать эту разницу. Кроме того, этот случай   
можно отнести к наиболее сложным случаям форматирования, которые ред-  
ко встречаются на практике, – в более типичных случаях выбор между дву-  
мя способами форматирования больше напоминает жеребьевку. Кроме того,   
в версии Python 3.1 (когда писалась эта глава, она находилась еще в состоянии   
альфа-версии) допускается не указывать порядковые номера позиционных ар-  
гументов, что несколько снижает преимущества метода format:  
C:\misc> C:\Python31\python  
>>> ‘The {0} side {1} {2}’.format(‘bright’, ‘of’, ‘life’)  
‘The bright side of life’  
>>>  
>>> ‘The {} side {} {}’.format(‘bright’, ‘of’, ‘life’) # Python 3.1+  
‘The bright side of life’  
>>>  
>>> ‘The %s side %s %s’ % (‘bright’, ‘of’, ‘life’)  
‘The bright side of life’  
Использование возможности автоматической нумерации в версии 3.1, как   
в данном примере, снижает преимущества метода format. А если сравнить опе-  
рации форматирования, например, вещественных чисел, можно заметить, что   
выражение с оператором % по-прежнему получается более компактным и вы-  
глядит менее запутанным:  
C:\misc> C:\Python31\python  
>>> ‘{0:f}, {1:.2f}, {2:05.2f}’.format(3.14159, 3.14159, 3.14159)  
‘3.141590, 3.14, 03.14’  
>>>  
>>> ‘{:f}, {:.2f}, {:06.2f}’.format(3.14159, 3.14159, 3.14159)  
‘3.141590, 3.14, 003.14’  
>>>  
>>> ‘%f, %.2f, %06.2f’ % (3.14159, 3.14159, 3.14159)  
‘3.141590, 3.14, 003.14’  
Имя метода и универсальные аргументы  
Учитывая появление в версии 3.1 возможности автоматической нумерации   
параметров, единственными потенциальными преимуществами метода format

248   
Глава 7. Строки   
остаются замена оператора % более говорящим названием метода и отсутствие   
различий между операциями с заменой единственного или нескольких значе-  
ний. Первое из этих преимуществ может заинтересовать начинающих програм-  
мистов (слово «format» в тексте программы распознается проще, чем множество   
символов «%»), хотя это слишком субъективно.  
Последнее преимущество может оказаться более существенным – при ис-  
пользовании оператора форматирования % единственное значение можно ука-  
зать непосредственно, но когда необходимо передать несколько значений, они   
должны быть заключены в кортеж:  
>>> ‘%.2f’ % 1.2345  
‘1.23’  
>>> ‘%.2f %s’ % (1.2345, 99)  
‘1.23 99’  
С технической точки зрения, оператор форматирования принимает один объ-  
ект – либо само значение для подстановки, либо кортеж с одним или более   
элементами. Фактически из-за того, что единственный элемент может переда-  
ваться либо сам по себе, либо внутри кортежа, когда возникает необходимость   
отформатировать кортеж, он должен быть оформлен, как вложенный кортеж:  
>>> ‘%s’ % 1.23  
‘1.23’  
>>> ‘%s’ % (1.23,)  
‘1.23’  
>>> ‘%s’ % ((1.23,),)  
‘(1.23,)’  
Метод format, напротив, ликвидирует различия между этими двумя случаями,   
принимая одни и те же аргументы:  
>>> ‘{0:.2f}’.format(1.2345)  
‘1.23’  
>>> ‘{0:.2f} {1}’.format(1.2345, 99)  
‘1.23 99’  
>>> ‘{0}’.format(1.23)  
‘1.23’  
>>> ‘{0}’.format((1.23,))  
‘(1.23,)’  
Из всего вышесказанного можно сделать вывод, что метод format удобнее для   
начинающих программистов и меньше способствует появлению ошибок про-  
граммирования. Однако эту проблему нельзя признать существенной – если   
всегда заключать значения в кортеж и не пользоваться возможностью прямой   
передачи единственного значения, оператор % по сути становится похожим на   
метод format. Кроме того, из-за ограниченной гибкости метода его использова-  
ние приводит к увеличению объема программного кода, и учитывая, что опе-  
ратор форматирования широко использовался на протяжении всей истории   
развития Python, пока не ясно, насколько оправданно переводить существую-  
Python, пока не ясно, насколько оправданно переводить существую-  
, пока не ясно, насколько оправданно переводить существую-  
щий программный код на использование нового инструмента, о чем мы погово-  
рим в следующем разделе.  
Возможный отказ в будущем?  
Как уже упоминалось выше, существует определенный риск, что в будущем   
разработчики языка Python могут объявить оператор % нерекомендуемым,

Общие категории типов   
249  
в пользу метода format. Фактически о такой возможности явно говорится   
в справочном руководстве для версии Python 3.0.  
Конечно, этого пока не произошло, и оба способа форматирования доступны без   
каких-либо ограничений в версиях Python 2.6 и 3.0 (версии, которые рассма-  
Python 2.6 и 3.0 (версии, которые рассма-  
 2.6 и 3.0 (версии, которые рассма-  
триваются в данной книге). Поддержка обоих способов сохранится и в прибли-  
жающемся выпуске Python 3.1, поэтому отказ от какого-либо из них выглядит   
достаточно маловероятным в обозримом будущем. Кроме того, благодаря тому,   
что оператор форматирования широко используется практически во всех су-  
ществующих программах на языке Python, большинству программистов будет   
полезно знать оба способа.  
Однако если в будущем оператор форматирования будет объявлен нерекомен-  
дуемым, вам может потребоваться переписать все выражения с оператором %,   
заменив их вызовами метода format, и преобразовать выражения, которые при-  
водятся в этой книге, чтобы перейти на использование более новой версии Py-  
Py-  
thon. Рискуя показаться тенденциозным, я, тем не менее, полагаю, что такое   
изменение будет основано на предпочтениях практикующих программистов,   
а не по прихоти ограниченной группы основных разработчиков – особенно   
если учесть, что теперь окно для проникновения новых несовместимостей в Py-  
Py-  
thon 3.0 уже закрыто. Откровенно говоря, такой отказ от одного из способов   
выглядел бы как замена одной сложности другой, которая в значительной сте-  
пени эквивалентна заменяемой! Тем не менее если вас волнует вопрос мигра-  
ции на будущие версии Python, обязательно старайтесь следить за событиями   
в разработке.   
Общие категории типов  
Теперь, когда мы исследовали строки, первый объект из коллекции языка   
Python, сделаем паузу, чтобы дать определение некоторым общим концепци-  
ям, применимым к большинству типов, которые будут рассматриваться даль-  
ше. Оказывается, что операции над встроенными типами работают одинаково   
в случае применения их к типам одной категории, поэтому нам необходимо   
лишь определить эти категории. К настоящему моменту мы исследовали толь-  
ко числа и строки, но так как они относятся к двум из трех основных категорий   
в языке Python, то вы знаете о других типах гораздо больше, чем могли бы по-  
думать.  
Типы одной категории имеют общий набор операций  
Как вы уже знаете, строки представляют собой неизменяемые последователь-  
ности: они не могут быть изменены непосредственно (неизменяемые) и явля-  
ются упорядоченными коллекциями элементов, доступ к которым может осу-  
ществляться по величине смещения (последовательности). Оказывается, над   
последовательностями, которые мы будем рассматривать в этой части книги,   
могут выполняться те же операции, которые были продемонстрированы в этой   
главе, – конкатенация, доступ к элементам по индексам, обход элементов в ци-  
кле и так далее. Формально в языке Python существует три категории типов   
(и операций):  
Числа (целые, вещественные, с фиксированной точностью, рациональные и др.)  
Поддерживают операции сложения, умножения и так далее.

250   
Глава 7. Строки   
Последовательности (строки, списки, кортежи)  
Поддерживают операции индексации, извлечения среза, конкатенации   
и так далее.  
Отображения (словари)  
Поддерживают операцию индексации по ключу и так далее.  
Множества образуют отдельную категорию типов (они не отображают ключи   
в значения и не являются упорядоченными последовательностями), и мы еще   
не рассматривали подробно отображения (словари обсуждаются в следующей   
главе), но к другим типам, которые нам встретятся, в основном будут приме-  
нимы одни и те же операции. Например, для любых объектов последователь-  
ностей X и Y:  
 •  
Выражение X + Y создает новый объект последовательности, включающий   
содержимое обоих операндов.  
 •  
Выражение X \* N создает новый объект последовательности, включающий N   
копий операнда X.  
Другими словами, эти операции действуют одинаково на любые виды последо-  
вательностей, включая строки, списки, кортежи и некоторые типы, определяе-  
мые пользователем. Единственное отличие состоит в том, что результат, воз-  
вращаемый выражением, имеет тот же тип, что и операнды X и Y, то есть, если   
выполняется операция конкатенации списков, то возвращается новый список,   
а не строка. Операции индексации и извлечения среза одинаково работают для   
любых последовательностей – тип объекта определяет, какая задача должна   
быть решена.  
Изменяемые типы   
допускают непосредственное изменение  
Классификация по возможности изменения – это слишком существенное огра-  
ничение, чтобы не помнить о нем, и все же она часто сбивает с толку начинаю-  
щих программистов. Если объект является значением неизменяемого типа, вы   
не сможете изменить его непосредственно – в этом случае интерпретатор будет   
выдавать сообщение об ошибке. Вместо этого необходимо, чтобы программный   
код создавал новый объект, содержащий новое значение. Основные базовые   
типы данных в языке Python делятся на следующие категории:  
Неизменяемые (числа, строки, кортежи, фиксированные множества)  
Объекты неизменяемых типов не поддерживают возможность непосред-  
ственного изменения значения объекта, однако вы всегда сможете созда-  
вать новые объекты с помощью выражений и присваивать их требуемым   
переменным.  
Изменяемые (списки, словари, множества)  
Объекты изменяемых типов, наоборот, всегда могут изменяться непосред-  
ственно, с помощью операций, которые не создают новые объекты. Изме-  
няемые объекты могут быть скопированы, но они поддерживают и возмож-  
ность непосредственного изменения.  
Вообще неизменяемые типы обеспечивают определенный уровень поддержки   
целостности, гарантируя, что объект не подвергнется изменениям в другой ча-  
сти программы. Чтобы вспомнить, почему это имеет такое большое значение,

В заключение   
251  
вернитесь к дискуссии о разделяемых ссылках на объекты в главе 6. Чтобы   
увидеть, как списки, словари и кортежи соотносятся с категориями типов, нам   
следует перейти к следующей главе.  
В заключение  
В этой главе мы подробно рассмотрели строковый тип объектов. Мы узнали   
о строковых литералах и исследовали операции над строками, включая вы-  
ражения с последовательностями, вызовы методов и форматирование с при-  
менением оператора форматирования и метода format. Попутно мы подробно   
изучили различные концепции, такие как извлечение среза, синтаксис вызова   
методов и блоки строк в тройных кавычках. Кроме того, мы определили неко-  
торые идеи, общие для различных типов: последовательности, к примеру, под-  
держивают общий набор операций.   
В следующей главе мы продолжим наши исследования и рассмотрим наибо-  
лее типичные коллекции объектов в языке Python – списки и словари. Там вы   
увидите, что многое из того, что говорилось здесь, применимо и к этим двум   
типам. Как уже упоминалось выше, в заключительной части этой книги мы   
вернемся к модели реализации строк в языке Python и подробнее рассмотрим   
особенности поддержки символов Юникода и двоичных данных в строках,   
которые представляют интерес для некоторых программистов, хотя и не для   
всех. Но перед этим ответьте на контрольные вопросы главы, которые помогут   
вам закрепить сведения, полученные здесь.  
Закрепление пройденного  
Контрольные вопросы  
1. Можно ли использовать строковый метод find для поиска в списках?  
2. Можно ли применить выражение извлечения среза к спискам?  
3. Как бы вы преобразовали символы в соответствующие им целочисленные   
коды ASCII? Как бы вы выполнили обратное преобразование – из кодов   
в символы?  
4. Как бы вы реализовали изменение строки на языке Python?  
5. Допустим, что имеется строка S со значением “s,pa,m”. Укажите два способа   
извлечения двух символов в середине строки.  
6. Сколько символов в строке “a\nb\x1f\000d”?  
7. По каким причинам вы могли бы использовать модуль string вместо стро-  
ковых методов?  
Ответы  
1.   
Нет, потому что методы всегда зависят от типа объекта. То есть они мо-  
гут применяться только к одному определенному типу данных. Но выра-  
жения, такие как X+Y, и встроенные функции, такие как len(X), являются   
более универсальными и могут использоваться для выполнения операций   
над различными типами. В данном случае, к примеру, оператор проверки   
вхождения in имеет похожий эффект и может использоваться для поиска

252   
Глава 7. Строки   
как в строках, так и в списках. В Python 3.0 была предпринята попытка   
сгруппировать методы по категориям (например, типы изменяемых после-  
довательностей, list и bytearray, обладают похожим множеством методов),   
и тем не менее методы все равно зависят от типа объекта в большей степени,   
чем другие разновидности операций.  
2. Да. В отличие от методов, выражения универсальны и могут использовать-  
ся для выполнения операций над различными типами. В данном случае   
операция извлечения среза в действительности является операцией над   
последовательностями – она может применяться к любому типу последо-  
вательностей, включая строки, списки и кортежи. Различие состоит лишь   
в том, что при выполнении операции над списком в результате будет полу-  
чен новый список.  
3. Встроенная функция ord(S) преобразует односимвольную строку в целочис-  
ленный код. Функция chr(I) преобразует целочисленный код в символ.  
4. Строки невозможно изменить – они являются неизменяемыми. Однако по-  
добного эффекта можно добиться, создав новую строку, – выполнив опера-  
цию конкатенации, извлечения среза, форматирования или вызвав метод,   
такой как replace, – и затем присвоив результат первоначальной перемен-  
ной.  
5. Можно извлечь подстроку с помощью выражения S[2:4] или, разбив строку   
по запятым, извлечь строку с помощью выражения S.split(‘,’)[1]. Попро-  
буйте выполнить эти выражения в интерактивном сеансе, чтобы получить   
представление о том, как они действуют.  
6. Шесть. Строка “a\nb\x1f\000d” содержит следующие символы: a, символ но-  
вой строки (\n), b, символ с десятичным кодом 31 (\x1f – в шестнадцатерич-  
ном представлении), символ с кодом 0 (\000 – в восьмеричном представле-  
нии) и d. Передайте эту строку встроенной функции len, чтобы проверить   
ответ и выведите результаты преобразования каждого символа с помощью   
функции ord, чтобы увидеть фактические значения кодов символов. За бо-  
лее подробной информацией обращайтесь к табл. 7.2.  
7. Вам никогда не следует использовать модуль string – он полностью ликви-  
дирован в версии Python 3.0. Единственная причина использования модуля   
string – это применение других средств из этого модуля, таких как предо-  
пределенные константы и шаблонные объекты. Кроме того, вы можете за-  
метить, что этот модуль используется только в «старом и пыльном» про-  
граммном коде на языке Python.

Глава 8.  
   
Списки и словари  
В этой главе вашему вниманию будут представлены такие типы объектов,   
как списки и словари, каждый из которых является коллекцией других объ-  
ектов. Эти два типа являются основными рабочими лошадками практически   
во всех сценариях на языке Python. Как вы увидите далее, оба типа облада-  
Python. Как вы увидите далее, оба типа облада-  
. Как вы увидите далее, оба типа облада-  
ют исключительной гибкостью: они могут изменяться непосредственно, могут   
увеличиваться и уменьшаться в размерах по мере необходимости и могут быть   
вложены в объекты любых других типов. Благодаря этим типам вы сможете   
создавать и обрабатывать в своих сценариях структуры данных любой степени   
сложности.  
Списки  
Следующая остановка в нашем путешествии по встроенным объектам языка   
Python называется список. Списки – это самые гибкий тип упорядоченных   
коллекций в языке Python. В отличие от строк, списки могут содержать объ-  
Python. В отличие от строк, списки могут содержать объ-  
. В отличие от строк, списки могут содержать объ-  
екты любых типов: числа, строки и даже другие списки. Кроме того, в отличие   
от строк, списки могут изменяться непосредственно, с помощью операции при-  
сваивания по смещениям и срезам, с помощью методов списков, с использова-  
нием инструкций удаления и другими способами – списки являются изменяе-  
мыми объектами.  
Списки в языке Python реализуют практически все необходимое для работы   
с коллекциями данных, что вам пришлось бы писать вручную при использо-  
вании низкоуровневого языка программирования, такого как язык C. Ниже   
приводятся основные свойства списков. Списки в языке Python – это:  
Упорядоченные коллекции объектов произвольных типов  
С функциональной точки зрения, списки – это лишь место, в котором со-  
браны другие объекты, поэтому их можно также рассматривать как груп-  
пы. Кроме того, списки обеспечивают позиционное упорядочение элементов   
слева направо (то есть они являются последовательностями).

254   
Глава 8. Списки и словари   
Доступ к элементам по смещению  
Так же как и в случае со строками, вы можете использовать операцию ин-  
дексирования для извлечения отдельных объектов из списка по их смеще-  
ниям. Поскольку элементы в списках упорядочены по их местоположению,   
можно также выполнять такие действия, как извлечение срезов и конкате-  
нация.  
Переменная длина, гетерогенность и произвольное число уровней вложенности  
В отличие от строк, списки могут увеличиваться и уменьшаться непосред-  
ственно (их длина может изменяться) и могут содержать не только односим-  
вольные строки, но и любые другие объекты (списки гетерогенны). Списки   
могут содержать другие сложные объекты и поддерживают возможность   
создания произвольного числа уровней вложенности, поэтому имеется воз-  
можность создавать списки списков из списков и так далее.  
Относятся к категории изменяемых объектов  
В терминах категорий типов списки могут изменяться непосредственно   
(являются изменяемыми объектами) и поддерживают все операции над по-  
следовательностями, которые поддерживаются и строками, такие как ин-  
дексирование, извлечение срезов и конкатенация. Операции над последо-  
вательностями одинаковым образом работают как в случае списков, так и в   
случае строк, единственное отличие – при применении операций над после-  
довательностями (таких как конкатенация и извлечение среза) к спискам   
возвращается новый список, а не новая строка. Кроме того, т. к. списки яв-  
ляются изменяемыми объектами, они поддерживают также операции, ко-  
торые не поддерживаются строками (такие как операции удаления и при-  
сваивания по индексам, изменяющие список непосредственно).  
Массивы ссылок на объекты  
Формально списки в языке Python могут содержать ноль или более ссылок   
на другие объекты. Списки чем-то напоминают массивы указателей (адре-  
сов). Извлечение элемента из списка в языке Python выполняется так же   
быстро, как извлечение элемента массива в языке C. В действительности   
списки – это самые настоящие массивы языка C, реализованные в интер-  
C, реализованные в интер-  
, реализованные в интер-  
претаторе Python, а не связанные структуры данных. Как мы узнали в гла-  
Python, а не связанные структуры данных. Как мы узнали в гла-  
, а не связанные структуры данных. Как мы узнали в гла-  
ве 6, всякий раз, когда используется ссылка на объект, интерпретатор разы-  
меновывает ее, поэтому ваши программы всегда будут иметь дело только   
с объектами. Всякий раз, когда выполняется присваивание объекта элемен-  
ту какой-либо структуры или имени переменной, интерпретатор Python со-  
Python со-  
 со-  
храняет ссылку на этот объект, а не его копию (за исключением, когда явно   
запрашивается выполнение операции копирования).  
В табл. 8.1 приводятся наиболее типичные операции, применяемые к спискам.   
Как обычно, за дополнительной информацией о списках вам следует обра-  
титься к руководству по стандартной библиотеке языка Python или запустить   
help(list) или dir(list) в интерактивной оболочке, чтобы получить полный   
перечень методов списков – этим функциям можно передать существующий   
список или слово list, которое является именем типа данных «список».

Списки   
255  
Таблица 8.1. Литералы списков и операции  
Операция  
Интерпретация  
L = []  
Пустой список  
L = [0, 1, 2, 3]  
Четыре элемента с индексами 0..3  
L = [‘abc’, [‘def’, ghi’]]  
Вложенные списки  
L = list(‘spam’)  
L = list(range(-4, 4))  
Создание списка из итерируемого объекта.  
Создание списка из непрерывной последова-  
тельности целых чисел  
L[i]  
L[i][j]  
L[i:j]  
len(L)  
Индекс, индекс индекса, срез, длина  
L1 + L2  
L \* 3  
Конкатенация, дублирование  
for x in L: print(x)  
3 in L  
Обход в цикле, проверка вхождения  
L.append(4)  
L.extend([5,6,7])  
L.insert(I, X)  
Методы: добавление элементов в список  
L.index(1)  
L.count()  
Методы: поиск  
L.sort()  
L.reverse()  
Методы: сортировка, изменение порядка сле-  
дования элементов на обратный  
del L[k]  
del L[i:j]  
L.pop()  
L.remove(2)  
L[i:j] = []  
Уменьшение списка  
L[i] = 1  
L[i:j] = [4,5,6]  
Присваивание по индексу, присваивание срезу  
L = [x\*\*2 for x in range(5)]  
list(map(ord, ‘spam’))  
Генераторы списков и отображение (главы 14   
и 20)  
Когда список определяется литеральным выражением, он записывается как   
последовательность объектов (точнее, как последовательность выражений, соз-  
дающих объекты) в квадратных скобках, разделенных запятыми. Например,

256   
Глава 8. Списки и словари   
во второй строке в табл. 8.1 выполняется присваивание переменной L списка из   
четырех элементов. Вложенные списки описываются как вложенные последо-  
вательности квадратных скобок (строка 3), а пустые списки определяются как   
пустая пара квадратных скобок (строка 1).1  
Многие операции из табл. 8.1 должны выглядеть для вас знакомыми, так как   
они являются теми же самыми операциями над последовательностями, кото-  
рые мы применяли к строкам, – индексирование, конкатенация, обход элемен-  
тов в цикле и так далее. Кроме того, списки поддерживают специфические для   
своего типа методы (такие как сортировка, перестановка элементов в обратном   
порядке, добавление элементов в конец списка и так далее), а также операции   
непосредственного изменения списка (удаление элементов, присваивание по   
индексам и срезам и так далее). Списки получили эти операции потому, что   
они относятся к категории объектов изменяемых типов.  
Списки в действии  
Возможно, самый лучший способ понять принцип действия списков – это по-  
смотреть на них в действии. Давайте еще раз вернемся к интерактивному сеан-  
су работы с интерпретатором и проиллюстрируем операции из табл. 8.1.  
Базовые операции над списками  
Поскольку списки являются последовательностями, они, как и строки, под-  
держивают операторы + и \* – для списков они так же соответствуют операции   
конкатенации и повторения, но в результате получается новый список, а не   
строка:  
% python  
>>> len([1, 2, 3]) # Длина  
3  
>>> [1, 2, 3] + [4, 5, 6] # Конкатенация  
[1, 2, 3, 4, 5, 6]  
>>> [‘Ni!’] \* 4 # Повторение  
[‘Ni!’, ‘Ni!’, ‘Ni!’, ‘Ni!’]  
Несмотря на то что оператор + со списками работает точно так же, как и со стро-  
ками, очень важно знать, что с обеих сторон оператора должны находиться по-  
следовательности одного и того же типа, в противном случае во время работы   
программы вы получите сообщение об ошибке.   
Например, нельзя выполнить операцию конкатенации для списка и строки,   
если предварительно не преобразовать список в строку (используя, например,   
функцию str или оператор форматирования %) или строку в список (с помощью   
встроенной функцией list):  
1   
На практике в текстах программ обработки списков вы нечасто встретите списки, за-  
данные подобным образом. Чаще вам будет встречаться программный код, обрабаты-  
вающий списки, которые создаются динамически (во время выполнения). Несмотря   
на всю важность владения синтаксисом литералов, следует иметь в виду, что боль-  
шая часть структур данных в языке Python конструируется во время выполнения   
программы.

Списки в действии   
257  
>>> str([1, 2]) + “34” # То же, что и “[1, 2]” + “34”  
‘[1, 2]34’  
>>> [1, 2] + list(“34”) # То же, что и [1, 2] + [“3”, “4”]  
[1, 2, ‘3’, ‘4’]  
Итерации по спискам и генераторы списков  
В целом списки могут участвовать во всех операциях над последовательностя-  
ми, которые мы применяли к строкам в предыдущей главе, включая инстру-  
менты выполнения итераций:  
>>> 3 in [1, 2, 3] # Проверка на вхождение  
True  
>>> for x in [1, 2, 3]:  
... print(x, end=’ ‘) # Итерации  
...  
1 2 3  
Подробнее об операции обхода элементов списка в цикле for и о встроенной   
функции range мы поговорим в главе 13, потому что они имеют отношению   
к синтаксису инструкций. Говоря коротко, оператор цикла for выбирает эле-  
менты последовательности в порядке слева направо и выполняет одну или бо-  
лее инструкций для каждого из них.   
Последняя строка в табл. 8.1 представляет генератор списков и вызов встроен-  
ной функции map, которые подробно описываются в главе 14 и расширенное об-  
суждение которых приводится в главе 20. Как говорилось в главе 4, генераторы   
списков – это способ построить новый список, применяя выражение к каждо-  
му элементу последовательности� они являются близкими родственниками ин-  
струкции цикла for.  
>>> res = [c \* 4 for c in ‘SPAM’] # Генератор списков  
>>> res  
[‘SSSS’, ‘PPPP’, ‘AAAA’, ‘MMMM’]  
Функционально это выражение эквивалентно циклу for, создающему список   
результатов вручную, но, как мы узнаем в следующих главах, генераторы спи-  
сков не только имеют более простой синтаксис, но и выполняются быстрее:  
>>> res = []  
>>> for c in ‘SPAM’: # Эквивалент генератора списков  
... res.append(c \* 4)  
...  
>>> res  
[‘SSSS’, ‘PPPP’, ‘AAAA’, ‘MMMM’]  
Как уже говорилось в главе 4, встроенная функция map выполняет похожие дей-  
ствия, но применяет к элементам последовательности не выражение, а функ-  
цию, и из полученных результатов создает новый список:  
>>> list(map(abs, [-1, -2, 0, 1, 2])) # Функция map применяется к   
[1, 2, 0, 1, 2] # последовательности  
Пока мы еще не готовы к детальному обсуждению механизма итераций, поэто-  
му отложим пока рассмотрение подробностей на будущее, но далее в этой главе   
мы еще встретимся с похожими выражениями-генераторами для словарей.

258   
Глава 8. Списки и словари   
Индексы, срезы и матрицы  
Так как списки являются последовательностями, операции доступа к элемен-  
там по индексам и извлечения срезов работают точно так же, как и в случае со   
строками. Однако в результате обращения к элементу по индексу возвращает-  
ся объект, который расположен по указанному смещению, а в результате опе-  
рации извлечения среза всегда возвращается новый список:  
>>> L = [‘spam’, ‘Spam’, ‘SPAM!’]  
>>> L[2] # Отсчет смещений начинается с нуля  
‘SPAM!’  
>>> L[-2] # Отрицательное смещение: отсчитывается справа  
‘Spam’  
>>> L[1:] # Операция извлечения среза возвращает разделы списка  
[‘Spam’, ‘SPAM!’]  
Здесь следует отметить следующее: поскольку списки (и объекты других ти-  
пов) могут быть вложены в другие списки, иногда бывает необходимо объеди-  
нять в цепочку несколько индексов, чтобы получить доступ к элементам на   
более глубоком уровне вложенности в структуре данных. Например, один из   
простейших способов представления матриц (многомерных массивов) в языке   
Python заключается в использовании вложенных списков. Ниже приводится   
пример двухмерного массива размером 3x3, построенного на базе списков:  
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
С помощью первого индекса извлекается целая строка (в действительности –   
вложенный список), а с помощью второго извлекается элемент этой строки:  
>>> matrix[1]  
[4, 5, 6]  
>>> matrix[1][1]  
5  
>>> matrix[2][0]  
7  
>>> matrix = [[1, 2, 3],  
... [4, 5, 6],  
... [7, 8, 9]]  
>>> matrix[1][1]  
5  
Предыдущий пример демонстрирует, в частности, что определение списка мо-  
жет при необходимости располагаться в нескольких строках, так как список   
ограничен квадратными скобками (подробнее о синтаксисе будет говориться   
в следующей части книги). Далее в этой главе будут также представлены ма-  
трицы, реализованные на базе словарей. Кроме того, для высокопроизводи-  
тельной работы с числовыми данными упомянутый в главе 5 модуль NumPy   
предоставляет другие способы организации матриц.  
Изменение списка  
Списки относятся к категории изменяемых объектов, поэтому они поддержи-  
вают операции, которые изменяют сам список непосредственно. То есть все   
операции, представленные в этом разделе, изменяют сам список объектов и не   
приводят к необходимости создавать новую копию, как это было в случае со

Списки в действии   
259  
строками. В языке Python приходится иметь дело только со ссылками на объ-  
екты, что обусловливает существенные различия между непосредственным   
изменением объекта и созданием нового объекта – как обсуждалось в главе 6,   
непосредственное изменение объекта может отражаться более чем на одной   
ссылке.  
Присваивание по индексам и срезам  
При использовании списков существует возможность изменять их содержи-  
мое, выполняя присваивание значений элементам (по смещению) или целым   
разделам (срезам) списка:  
>>> L = [‘spam’, ‘Spam’, ‘SPAM!’]  
>>> L[1] = ‘eggs’ # Присваивание по индексу элемента  
>>> L  
[‘spam’, ‘eggs’, ‘SPAM!’]  
>>> L[0:2] = [‘eat’, ‘more’] # Присваивание срезу: удаление+вставка  
>>> L # Элементы 0 и 1 были заменены  
[‘eat’, ‘more’, ‘SPAM!’]  
Обе операции присваивания – и отдельному элементу, и срезу – производят-  
ся непосредственно в списке – они изменяют сам список, а не создают новый   
список объектов. Операция присваивания по индексу в языке Python работает   
практически так же, как в языке C и во многих других языках программиро-  
вания: интерпретатор замещает старую ссылку на объект в указанном смеще-  
нии на новую.  
Присваивание  срезу, последняя операция в предыдущем примере, замещает   
целый раздел списка за один прием. Поскольку это довольно сложная опера-  
ция, проще будет представить ее, как последовательное выполнение двух дей-  
ствий:  
1. Удаление. Раздел списка, определяемый слева от оператора =, удаляется.  
2. Вставка. Новые элементы, содержащиеся в объекте, расположенном спра-  
ва от оператора =, вставляются в список, начиная с левого края, где нахо-  
дился прежний удаленный срез.1  
В действительности это не совсем то, что происходит на самом деле, но это до-  
статочно точно объясняет, почему число вставляемых элементов не должно со-  
ответствовать числу удаляемых элементов. Например, представим, что список   
L имеет значение [1,2,3], тогда в результате операции присваивания L[1:2]=[4,5]   
будет получен список [1,4,5,3]. Интерпретатор сначала удалит 2 (срез, состоя-  
щий из одного элемента), а затем, начиная с позиции удаленного элемента 2,   
вставит элементы 4 и 5. Это также объясняет, почему операция L[1:2]=[] в дей-  
ствительности является операцией удаления – интерпретатор удалит срез (эле-  
мент со смещением 1) и затем вставит пустой список.  
В результате операция присваивания срезу замещает целый раздел списка,   
или «столбец», за одно действие. Поскольку длина последовательности справа   
от оператора = не должна обязательно соответствовать длине среза, которому   
1   
Здесь требуется дополнительное уточнение, описывающее случай, когда при присва-  
ивании происходит перекрытие срезов, например: выражение L[2:5]=L[3:6], будет   
выполнено безошибочно, потому что перед тем, как срез слева будет удален, сначала   
будет произведено извлечение среза справа.

260   
Глава 8. Списки и словари   
выполняется присваивание, эта операция может использоваться для замены   
(посредством перезаписи), расширения (посредством вставки) или сжатия (по-  
средством удаления) требуемого списка. Это довольно мощная операция, но,   
честно говоря, она достаточно редко используется на практике. Обычно ис-  
пользуются более простые способы замены, вставки и удаления (например,   
операция конкатенация и методы списков insert, pop и remove), которые про-  
граммисты предпочитают использовать на практике.  
Методы списков  
Как и строки, объекты списков в языке Python поддерживают специфичные   
методы, многие из которых изменяют сам список непосредственно:  
>>> L.append(‘please’) # Вызов метода добавления элемента в конец списка  
>>> L  
[‘eat’, ‘more’, ‘SPAM!’, ‘please’]  
>>> L.sort() # Сортировка элементов списка (‘S’ < ‘e’)  
>>> L  
[‘SPAM!’, ‘eat’, ‘more’, ‘please’]  
Методы были представлены в главе 7. Коротко напомню, что методы – это   
функции (в действительности – атрибуты, ссылающиеся на функции), которые   
связаны с определенным типом объектов. Методы обеспечивают выполнение   
специфических операций, например методы списков, представленные здесь,   
доступны только для списков.  
Наиболее часто используемым методом, пожалуй, является метод append, кото-  
рый просто добавляет единственный элемент (ссылку на объект) в конец спи-  
ска. В отличие от операции конкатенации, метод append принимает единствен-  
ный объект, а не список. По своему действию выражение L.append(X) похоже на   
выражение L+[X], но в первом случае изменяется сам список, а во втором – соз-  
дается новый список.1   
Другой часто используемый метод – метод sort, выполняет переупорядочива-  
ние элементов в самом списке. По умолчанию он использует стандартные опе-  
раторы сравнения языка Python (в данном случае выполняется сравнивание   
строк) и выполняет сортировку в порядке возрастания значений.   
Однако существует возможность изменить порядок сортировки с помощью   
именованных аргументов – специальных синтаксических конструкций вида   
«name�value», которые используются в вызовах функций для передачи пара-  
name�value», которые используются в вызовах функций для передачи пара-  
�value», которые используются в вызовах функций для передачи пара-  
value», которые используются в вызовах функций для передачи пара-  
», которые используются в вызовах функций для передачи пара-  
метров настройки по их именам. Именованный аргумент key в вызове метода   
sort позволяет определить собственную функцию сравнения, принимающую   
единственный аргумент и возвращающую значение, которое будет использо-  
вано в операции сравнения, а именованный аргумент reverse позволяет выпол-  
нить сортировку не в порядке возрастания, а в порядке убывания:  
1   
В отличие от операции конкатенации (+), метод append не создает новый объект, по-  
этому обычно он выполняется быстрее. Существует возможность имитировать рабо-  
ту метода append с помощью операции присваивания срезу: выражение L[len(L):]=[X]   
соответствует вызову L.append(X), а выражение L[:0]=[X] соответствует операции до-  
бавления в начало списка. В обоих случаях удаляется пустой сегмент списка и встав-  
ляется элемент X, при этом изменяется сам список L, так же быстро, как при исполь-  
зовании метода append.

Списки в действии   
261  
>>> L = [‘abc’, ‘ABD’, ‘aBe’]  
>>> L.sort() # Сортировка с учетом регистра символов  
>>> L  
[‘ABD’, ‘aBe’, ‘abc’]  
>>> L = [‘abc’, ‘ABD’, ‘aBe’]  
>>> L.sort(key=str.lower) # Приведение символов к нижнему регистру  
>>> L  
[‘abc’, ‘ABD’, ‘aBe’]  
>>>  
>>> L = [‘abc’, ‘ABD’, ‘aBe’]  
>>> L.sort(key=str.lower, reverse=True) # Изменяет направление сортировки  
>>> L  
[‘aBe’, ‘ABD’, ‘abc’]  
Аргумент key может также пригодиться при сортировке списков словарей, ког-  
да с его помощью можно указать ключ, по которому будет определяться поло-  
жение каждого словаря в отсортированном списке. Словари мы будем изучать   
ниже, в этой главе, а более подробные сведения об именованных аргументах вы   
получите в четвертой части книги.  
Сравнивание и сортировка в Python 3.0: В Python 2.6 и в более   
ранних версиях сравнивание выполняется по-разному для объ-  
ектов разных типов (например, списков и строк) – язык задает   
способ упорядочения различных типов, который можно при-  
знать скорее детерминистским, чем эстетичным. Этот способ   
упорядочения основан на именах типов, вовлеченных в опера-  
цию сравнения, например любые целые числа всегда меньше   
любых строк, потому что строка “int” меньше, чем строка “str”.   
При выполнении операции сравнения никогда не выполняется   
преобразование типов объектов, за исключением сравнивания   
объектов числовых типов.  
В Python 3.0 такой порядок был изменен: попытки сравнивания   
объектов различных типов возбуждают исключение – вместо   
сравнивания по названиям типов. Так как метод сортировки ис-  
пользует операцию сравнения, это означает, что инструкция [1,   
2, ‘spam’].sort() будет успешно выполнена в Python 2.X, но воз-  
X, но воз-  
, но воз-  
будит исключение в версии Python 3.0 и выше.   
Кроме того, в версии Python 3.0 больше не поддерживается воз-  
Python 3.0 больше не поддерживается воз-  
 3.0 больше не поддерживается воз-  
можность передачи методу sort произвольной функции сравне-  
ния, для реализации иного способа упорядочения. Чтобы обой-  
ти это ограничение, предлагается использовать именованный   
аргумент key=func, в котором предусматривать возможность   
трансформации значений в процессе сортировки, и применять   
именованный аргумент reverse=True для сортировки по убыва-  
нию. То есть фактически выполнять те же действия, которые   
раньше выполнялись внутри функции сравнения.  
Важно заметить, что методы append и sort изменяют сам объект списка и не   
возвращают список в виде результата (точнее говоря, оба метода возвращают   
значение None). Если вы написали инструкцию вроде L=L.append(X), вы не полу-  
чите измененное значение L (в действительности вы вообще потеряете ссылку

262   
Глава 8. Списки и словари   
на список) – использование таких атрибутов, как append и sort, приводит к из-  
менению самого объекта, поэтому нет никаких причин выполнять повторное   
присваивание.  
Отчасти из-за этих особенностей методов в последних версиях Python сорти-  
Python сорти-  
 сорти-  
ровку можно также выполнить с помощью встроенной функции, которая спо-  
собна сортировать не только списки, но и любые другие последовательности   
и возвращает новый список с результатом сортировки (оригинальный список   
при этом не изменяется):  
>>> L = [‘abc’, ‘ABD’, ‘aBe’]  
>>> sorted(L, key=str.lower, reverse=True) # Встроенная функция сортировки  
[‘aBe’, ‘ABD’, ‘abc’]  
   
>>> L = [‘abc’, ‘ABD’, ‘aBe’]  
>>> sorted([x.lower() for x in L], reverse=True) # Элементы предварительно  
[‘abe’, ‘abd’, ‘abc’] # изменяются!  
Обратите внимание, что в последнем примере перед сортировкой с помощью   
генератора списков выполняется приведение символов к нижнему регистру,   
и значения элементов в получившемся списке отличаются от значений элемен-  
тов в оригинальном списке – в противоположность примеру с использованием   
именованных аргументов. В последнем примере выполняется сортировка не   
оригинального, а временного списка, созданного в процессе сортировки. По   
мере продвижения дальше мы познакомимся с ситуациями, когда встроенная   
функция sorted может оказаться более удобной, чем метод sort.  
Как и строки, списки обладают рядом других методов, выполняющих специа-  
лизированные операции. Например, метод reverse изменяет порядок следова-  
ния элементов в списке на обратный, а методы extend и pop вставляют несколько   
элементов в конец списка и удаляют элементы из конца списка соответственно.   
Кроме того, существует встроенная функция reversed, которая во многом на-  
поминает встроенную функцию sorted, но ее необходимо обертывать в вызов   
функции list, потому что она возвращает итератор (подробнее об итераторах   
мы поговорим позднее):  
>>> L = [1, 2]  
>>> L.extend([3,4,5]) # Добавление нескольких элементов в конец списка  
>>> L  
[1, 2, 3, 4, 5]  
>>> L.pop() # Удаляет и возвращает последний элемент списка  
5  
>>> L  
[1, 2, 3, 4]  
>>> L.reverse() # Изменяет порядок следования элементов на обратный  
>>> L  
[4, 3, 2, 1]  
>>> list(reversed(L)) # Встроенная функция сортировки в обратном порядке  
[1, 2, 3, 4]   
В некоторых типах программ метод pop, показанный здесь, часто используется   
в паре с методом append для реализации структур данных типа стек – «послед-  
ний пришел, первый ушел» (Last-In-First-Out, LIFO). Конец списка служит   
вершиной стека:

Списки в действии   
263  
>>> L = []  
>>> L.append(1) # Втолкнуть на стек  
>>> L.append(2)  
>>> L  
[1, 2]  
>>> L.pop() # Вытолкнуть со стека  
2  
>>> L  
[1]  
Хотя это здесь и не показано, тем не менее метод pop может принимать необя-  
зательное смещение элемента, который удаляется из списка и возвращается   
(по умолчанию это последний элемент). Другие методы списков позволяют   
удалять элементы с определенными значениями (remove), вставлять элементы   
в определенную позицию (insert), отыскивать смещение элемента по заданно-  
му значению (index) и так далее:  
>>> L = [‘spam’, ‘eggs’, ‘ham’]  
>>> L.index(‘eggs’) # Индекс объекта  
1  
>>> L.insert(1, ‘toast’) # Вставка в требуемую позицию  
>>> L  
[‘spam’, ‘toast’, ‘eggs’, ‘ham’]  
>>> L.remove(‘eggs’) # Удаление элемента с определенным значением  
>>> L  
[‘spam’, ‘toast’, ‘ham’]  
>>> L.pop(1) # Удаление элемента в указанной позиции  
‘toast’  
>>> L  
[‘spam’, ‘ham’]  
Чтобы поближе познакомиться с этими методами, обратитесь к имеющимся   
источникам документации или поэкспериментируйте с этими методами в ин-  
терактивной оболочке интерпретатора.  
Прочие часто используемые операции над списками  
Так как списки относятся к категории изменяемых объектов, вы можете ис-  
пользовать инструкцию del для удаления элемента или среза непосредственно   
из списка:  
>>> L  
[‘SPAM!’, ‘eat’, ‘more’, ‘please’]  
>>> del L[0] # Удаление одного элемента списка  
>>> L  
[‘eat’, ‘more’, ‘please’]  
>>> del L[1:] # Удаление целого сегмента списка  
>>> L # То же, что и L[1:] = []  
[‘eat’]  
Так как операция присваивания срезу выполняется как удаление и вставка,   
можно удалять срезы списка, присваивая им пустой список (L[i:j]=[]) – интер-  
претатор сначала удалит срез, определяемый слева от оператора =, а затем вста-  
вит пустой список. С другой стороны, присваивание пустого списка по индексу   
элемента приведет к сохранению ссылки на пустой список в этом элементе, а не   
к его удалению:

264   
Глава 8. Списки и словари   
>>> L = [‘Already’, ‘got’, ‘one’]  
>>> L[1:] = []  
>>> L  
[‘Already’]  
>>> L[0] = []  
>>> L  
[[]]  
Все только что рассмотренные операции используются достаточно часто, од-  
нако существуют и другие дополнительные методы и операции для списков,   
которые не были показаны здесь (включая методы вставки и поиска). Чтобы   
получить полный перечень имеющихся дополнительных операций, всегда сле-  
дует обращаться к руководствам по языку Python, к функциям dir и help (с ко-  
торыми мы впервые познакомились в главе 4) или к книгам, упоминавшимся   
в предисловии.  
Кроме того, я хотел бы напомнить, что все операции непосредственного изме-  
нения объектов, обсуждавшиеся здесь, применимы только к изменяемым объ-  
ектам: они не будут работать со строками (или с кортежами, которые рассма-  
триваются в главе 9), независимо от прикладываемых вами усилий. Изменяе-  
мость или неизменяемость – это исходное свойство, присущее каждому типу   
объектов.  
Словари  
После списков словари являются, пожалуй, самым гибким из встроенных ти-  
пов данных в языке Python. Если списки являются упорядоченными коллек-  
циями объектов, то в отличие от них элементы в словарях сохраняются и из-  
влекаются с помощью ключа, а не с помощью смещения, определяющего их   
позицию.  
Будучи встроенным типом данных, словари могут заменить множество алго-  
ритмов поиска и структур данных, которые приходится реализовывать вруч-  
ную в низкоуровневых языках программирования, – доступ к элементам сло-  
варя по их индексам представляет собой быструю операцию поиска. Кроме   
того, иногда словари могут играть роль записей и таблиц символов, используе-  
мых в других языках, и способны служить для представления разреженных   
(по большей части пустых) структур данных. Ниже приводятся основные ха-  
рактеристики словарей в языке Python:  
Доступ к элементам по ключу, а не по индексу  
Иногда словари называют ассоциативными массивами, или хешами. Они   
определяют взаимосвязь между значениями и ключами, поэтому для из-  
влечения элементов словаря можно использовать ключи, под которыми эти   
элементы были сохранены в словаре. Для получения элементов словаря ис-  
пользуется та же самая операция доступа по индексу, как и в списке, толь-  
ко индекс приобретает форму ключа, а не смещения относительно начала.   
Неупорядоченные коллекции произвольных объектов  
В отличие от списков, элементы словарей хранятся в неопределенном по-  
рядке. В действительности, интерпретатор вносит элемент случайности

Словари   
265  
в порядок следования элементов для обеспечения более быстрого поиска.   
Ключи описывают символическое (не физическое) местоположение элемен-  
тов в словаре.  
Переменная длина, гетерогенность и произвольное число уровней вложенности  
Подобно спискам словари могут увеличиваться и уменьшаться непосред-  
ственно (то есть без создания новых копий). Они могут содержать объекты   
любых типов и поддерживают возможность создания произвольного числа   
уровней вложенности (они могут содержать списки, другие словари и так   
далее).  
Относятся к категории «изменяемых отображений»  
Словари могут изменяться непосредственно с использованием операции   
индексирования (они являются изменяемыми), но они не поддерживают   
операции над последовательностями, которые поддерживаются строками   
и списками. Словари представляют собой неупорядоченные коллекции,   
поэтому операции, которые основаны на использовании фиксированного   
порядка следования элементов (например, конкатенация, извлечение сре-  
за), не имеют смысла для словарей. Словари – это единственный встроен-  
ный представитель объектов-отображений (объекты, которые отображают   
ключи на значения).  
Таблицы ссылок на объекты (хеш-таблицы)  
Если списки – это массивы ссылок на объекты, которые поддерживают   
возможность доступа к элементам по их позициям, то словари – это неупо-  
рядоченные таблицы ссылок на объекты, которые поддерживают доступ   
к элементам по ключу. Внутри словари реализованы как хеш-таблицы   
(структуры данных, которые обеспечивают очень высокую скорость поис-  
ка), изначально небольшого размера и увеличивающиеся по мере необхо-  
димости. Более того, интерпретатор Python использует оптимизированные   
алгоритмы хеширования для обеспечения максимально высокой скорости   
поиска ключей. Подобно спискам, словари хранят ссылки на объекты (а не   
их копии).  
В табл. 8.2 приводятся некоторые наиболее часто используемые операции над   
словарями (опять же, чтобы получить полный перечень операций, обращай-  
тесь к руководству или воспользуйтесь функцией dir(dict) или help(dict), где   
dict – это имя типа). При определении в виде литералов словари записываются   
как последовательность пар key:value, разделенных запятыми, заключенных   
в фигурные скобки.1 Пустой словарь в литеральном представлении – это пу-  
стая пара скобок. Словари могут вкладываться в литеральном представлении   
в виде значений внутри других словарей, списков или кортежей.  
1   
Как и в случае со списками, вам нечасто придется встречать словари, сконструиро-  
ванные с использованием литералов. Однако списки и словари увеличиваются в раз-  
мерах по-разному. Как будет показано в следующем разделе, словари обычно допол-  
няются с помощью операции присваивания по новым ключам во время выполнения   
программы – такой подход совершенно не годится для списков (списки обычно рас-  
ширяются с помощью метода append).

266   
Глава 8. Списки и словари   
Таблица 8.2. Литералы словарей и операции  
Операция  
Интерпретация  
D = {}  
Пустой словарь  
D = {‘spam’: 2, ‘eggs’: 3}  
Словарь из двух элементов  
D = {‘food’: {‘ham’: 1, ‘egg’: 2}}  
Вложение  
D = dict(name=’Bob’, age=40)  
D = dict(zip(keyslist, valslist))  
D = dict.fromkeys([‘a’, ‘b’])  
Альтернативные способы создания   
словарей:  
именованные аргументы, применение   
функции zip, списки ключей  
D[‘eggs’]  
D[‘food’][‘ham’]  
Доступ к элементу по ключу  
‘eggs’ in D  
Проверка на вхождение: проверка нали-  
чия ключа  
D.keys()  
D.values()  
D.items()  
D.copy()  
D.get(key, default)  
D.update(D2)  
D.pop(key)  
Методы: список ключей,   
список значений,   
список ключей и значений,  
копирование,   
получение значения по умолчанию,   
слияние,   
удаление и так далее  
len(D)  
Длина (количество элементов)  
D[key] = 42  
del D[key]  
Добавление/изменение ключей,   
удаление ключей  
list(D.keys())  
D1.keys() & D2.keys()  
Представления словарей (в Python 3.0)  
D = {x: x\*2 for x in range(10)}  
Генераторы словарей (в Python 3.0)  
Словари в действии  
Согласно табл. 8.2, доступ к элементам словарей осуществляется по ключу,   
а для доступа к элементам вложенных словарей осуществляется путем объ-  
единения серии индексов (ключей в квадратных скобках) в цепочку. Когда   
интерпретатор создает словарь, он сохраняет элементы в произвольном поряд-  
ке – чтобы получить значение обратно, необходимо указать ключ, с которым   
это значение было ассоциировано. Давайте вернемся в интерактивный сеанс   
работы с интерпретатором и исследуем некоторые операции над словарями из   
табл. 8.2.

Словари в действии   
267  
Базовые операции над словарями  
В обычном случае сначала создается словарь, а затем выполняются операции   
сохранения новых ключей и обращения к элементам по ключу:  
% python  
>>> D = {‘spam’: 2, ‘ham’: 1, ‘eggs’: 3} # Создание словаря  
>>> D[‘spam’] # Извлечение значения по ключу  
2  
>>> D # Случайный порядок следования   
{‘eggs’: 3, ‘ham’: 1, ‘spam’: 2}  
Здесь переменной D присваивается словарь, в котором ключу ‘spam’ соответ-  
ствует целочисленное значение 2, и так далее. Для доступа к элементам слова-  
ря используется тот же самый синтаксис с квадратными скобками, что и при   
извлечении элементов списков, но в данном случае доступ осуществляется по   
ключу, а не по позиции элемента.  
Обратите внимание на последнюю инструкцию в этом примере: порядок сле-  
дования ключей в словаре практически всегда отличается от первоначально-  
го. Дело в том, что для обеспечения максимально высокой скорости поиска   
по ключу (для хеширования) ключи должны располагаться в памяти в ином   
порядке. Именно поэтому операции, которые предполагают наличие установ-  
ленного порядка следования элементов слева направо (например, извлечение   
среза, конкатенация), неприменимы к словарям – они позволяют извлекать   
значения только по ключам, а не по индексам.  
Встроенная функция len может работать и со словарями – она возвращает чис-  
ло элементов в словаре или, что то же самое, длину списка ключей. Оператор   
in проверки вхождения позволяет проверить наличие ключа, а метод keys воз-  
вращает все ключи, имеющиеся в словаре, в виде списка. Последний удобно ис-  
пользовать для последовательной обработки словарей, но при этом вы не долж-  
ны делать какие-либо предположения о порядке следования ключей в списке.   
Поскольку результатом вызова метода keys является список, он всегда может   
быть отсортирован (ниже мы подробнее коснемся проблемы сортировки сло-  
варей):  
>>> len(D) # Число элементов словаря  
3  
>>> ‘ham’ in D # Проверка на вхождение  
True  
>>> list(D.keys()) # Создает новый список ключей  
[‘eggs’, ‘ham’, ‘spam’]  
Обратите внимание на второе выражение в этом листинге. Как уже упомина-  
лось ранее, оператор проверки на вхождение in может использоваться для ра-  
боты со строками и списками, но точно так же он может использоваться и для   
работы со словарями. Это возможно благодаря тому, что словари определяют   
итераторы, которые обеспечивают пошаговый обход списков ключей. Суще-  
ствуют и другие типы, которые поддерживают итераторы, отвечающие обыч-  
ному использованию типа� например, файлы имеют итераторы, которые позво-  
ляют выполнять построчное чтение данных. Итераторы будут рассматриваться   
в главах 14 и 20.

268   
Глава 8. Списки и словари   
Обратите также внимание на синтаксис последнего примера в листинге. В вер-  
сии Python 3.0 мы были вынуждены заключить вызов метода в вызов функции   
list по уже встречавшейся ранее причине – в версии 3.0 метод keys возвраща-  
ет итератор, а не список. Вызов функции list принудительно выполняет обход   
всех значений итератора, что позволяет вывести их все сразу. В версии 2.6 ме-  
тод keys конструирует и возвращает обычный список, поэтому для отображе-  
ния результатов вызов функции list в этой версии интерпретатора не требует-  
ся. Подробнее об этом рассказывается ниже, в этой главе.  
Ключи в словарях следуют в произвольном порядке, который   
может изменяться от версии к версии, поэтому не нужно трево-  
житься, если у вас ключи будут выведены в порядке, отличном   
от того, что приводится здесь. В действительности порядок сле-  
дования ключей изменился и у меня – я выполнял все эти при-  
меры под управлением Python 3.0, и порядок следования клю-  
чей изменился по сравнению с тем, что приводился в примерах   
в предыдущем издании. Вы не должны полагаться на какой-то   
определенный порядок следования ключей ни в своих програм-  
мах, ни при чтении книг!  
Изменение словарей  
Давайте продолжим работу в интерактивном сеансе. Словари, как и списки,   
относятся к категории изменяемых объектов, поэтому их можно изменять,   
увеличивать, уменьшать непосредственно, не создавая новые словари: чтобы   
изменить или создать новую запись в словаре, достаточно выполнить операцию   
присваивания по ключу. Инструкция del также может применяться к слова-  
рям – она удаляет значение, связанное с ключом, который играет роль индек-  
са. Кроме того, обратите внимание на наличие вложенного списка в следую-  
щем примере (значение для ключа ‘ham’). Все типы-коллекции в языке Python   
могут вкладываться друг в друга в произвольном порядке:  
>>> D  
{‘eggs’: 3, ‘ham’: 1, ‘spam’: 2}  
   
>>> D[‘ham’] = [‘grill’, ‘bake’, ‘fry’] # Изменение элемента  
>>> D  
{‘eggs’: 3, ‘ham’: [‘grill’, ‘bake’, ‘fry’], ‘spam’: 2}  
   
>>> del D[‘eggs’] # Удаление элемента  
>>> D  
{‘ham’: [‘grill’, ‘bake’, ‘fry’], ‘spam’: 2}  
   
>>> D[‘brunch’] = ‘Bacon’ # Добавление нового элемента  
>>> D  
{‘brunch’: ‘Bacon’, ‘ham’: [‘grill’, ‘bake’, ‘fry’] , ‘spam’: 2}  
Как и в случае со списками, операция присваивания по существующему клю-  
чу словаря приводит к изменению ассоциированного с ним значения. Однако   
в отличие от списков, словари допускают выполнение присваивания по ново-  
му ключу (который ранее отсутствовал), в результате создается новый элемент   
словаря, как показано в предыдущем примере для ключа ‘brunch’. Этот при-

Словари в действии   
269  
ем не может применяться к спискам, потому что в этом случае интерпретатор   
обнаруживает выход за пределы списка и генерирует сообщение об ошибке.   
Чтобы увеличить размер списка, необходимо использовать такие инструменты   
списков, как метод append или присваивание срезу.  
Дополнительные методы словарей  
Методы словарей обеспечивают выполнение различных операций. Например,   
методы словарей values и items возвращают список значений элементов словаря   
и кортежи пар (key, value) соответственно:  
>>> D = {‘spam’: 2, ‘ham’: 1, ‘eggs’: 3}  
>>> list(D.values())  
[3, 1, 2]  
>>> list(D.items())  
[(‘eggs’, 3), (‘ham’, 1), (‘spam’, 2)]  
Такие списки удобно использовать в циклах, когда необходимо выполнить об-  
ход элементов словаря. Попытка извлечения несуществующего элемента сло-  
варя обычно приводит к появлению ошибки, однако метод get в таких случаях   
возвращает значение по умолчанию (None или указанное значение). С помощью   
этого метода легко можно реализовать получение значений по умолчанию и из-  
бежать появления ошибки обращения к несуществующему ключу:  
>>> D.get(‘spam’) # Ключ присутствует в словаре  
2  
>>> print(D.get(‘toast’)) # Ключ отсутствует в словаре  
None  
>>> D.get(‘toast’, 88)  
88  
Метод update реализует своего рода операцию конкатенации для словарей, при   
этом он не имеет никакого отношения к упорядочению элементов слева напра-  
во (для словарей такое упорядочение не имеет смысла). Он объединяет ключи   
и значения одного словаря с ключами и значениями другого, просто перезапи-  
сывая значения с одинаковыми ключами:  
>>> D  
{‘eggs’: 3, ‘ham’: 1, ‘spam’: 2}  
>>> D2 = {‘toast’:4, ‘muffin’:5}  
>>> D.update(D2)  
>>> D  
{‘toast’: 4, ‘muffin’: 5, ‘eggs’: 3, ‘ham’: 1, ‘spam’: 2}  
Наконец, метод pop удаляет ключ из словаря и возвращает его значение. Он на-  
поминает метод pop списков, только вместо необязательного индекса элемента   
принимает ключ:  
# удаление элементов словаря по ключу  
>>> D  
{‘toast’: 4, ‘muffin’: 5, ‘eggs’: 3, ‘ham’: 1, ‘spam’: 2}  
>>> D.pop(‘muffin’)  
5  
>>> D.pop(‘toast’) # Удаляет и возвращает значение заданного ключа  
4

270   
Глава 8. Списки и словари   
>>> D  
{‘eggs’: 3, ‘ham’: 1, ‘spam’: 2}  
   
# удаление элементов списка по номеру позиции  
>>> L = [‘aa’, ‘bb’, ‘cc’, ‘dd’]  
>>> L.pop() # Удаляет и возвращает последний элемент списка  
‘dd’  
>>> L  
[‘aa’, ‘bb’, ‘cc’]  
>>> L.pop(1) # Удаляет и возвращает элемент из заданной позиции  
‘bb’  
>>> L  
[‘aa’, ‘cc’]  
Кроме того, словари имеют метод copy, который мы рассмотрим в главе 9, как   
один из способов избежать побочных эффектов, связанных с наличием не-  
скольких ссылок на один и тот же словарь. В действительности словари обла-  
дают гораздо большим числом методов, чем перечислено в табл. 8.2. Чтобы по-  
лучить полный список, обращайтесь к руководствам по языку Python  
Таблица языков  
Давайте рассмотрим более жизненный пример словаря. В следующем примере   
создается таблица, которая отображает названия языков программирования   
(ключи) на имена их создателей (значения). С помощью этой таблицы можно по   
названию языка программирования определить имя его создателя:  
>>> table = {‘Python’: ‘Guido van Rossum’,  
... ‘Perl’: ‘Larry Wall’,  
... ‘Tcl’: ‘John Ousterhout’ }  
>>>  
>>> language = ‘Python’  
>>> creator = table[language]  
>>> creator  
‘Guido van Rossum’  
   
>>> for lang in table: # То же, что и: for lang in table.keys()  
... print(lang, ‘\t’, table[lang])  
...  
Tcl John Ousterhout  
Python Guido van Rossum  
Perl Larry Wall  
В последней команде использован оператор цикла for, который мы еще подроб-  
но не рассматривали. Для тех, кто не знаком с циклами for, замечу, что при-  
веденная команда просто выполняет обход всех ключей в таблице и выводит   
список ключей и их значений, разделенных символом табуляции. Подробно   
о циклах for будет рассказываться в главе 13.  
Словари не являются последовательностями, как списки и строки, но если не-  
обходимо выполнить обход элементов словаря, в этом нет ничего сложного –   
это легко сделать с помощью метода keys, возвращающего все ключи словаря,   
которые можно обойти в цикле for. В случае необходимости внутри цикла   
можно получать значение элемента по его ключу, как это реализовано в дан-  
ном примере.

Словари в действии   
271  
Кроме того, Python в действительности позволяет выполнять обход ключей   
словаря и без вызова метода keys в операторе цикла for. Для любого словаря D   
цикл можно оформить как for key in D:, что равносильно полной форме записи   
for key in D.keys():. Это всего лишь еще одна разновидность итераторов, упоми-  
навшихся ранее, которая позволяет использовать оператор проверки вхожде-  
ния in со словарями (подробнее об итераторах далее в книге).  
Замечания по использованию словарей  
Словари окажутся достаточно просты в использовании, когда вы освоите ра-  
боту с ними, но я хочу привести несколько соображений, которые вам следует   
твердо знать:  
 •  
Операции над последовательностями неприменимы к словарям. Словари –   
это отображения, а не последовательности. Вследствие того, что словари не   
предусматривают никакого упорядочения элементов, такие операции, как   
конкатенация (упорядоченное объединение) и извлечение среза (извлечение   
непрерывного блока элементов), просто неприменимы. В действительности,   
когда в программном коде во время выполнения производится попытка сде-  
лать нечто подобное, интерпретатор выдает сообщение об ошибке.  
 •  
Присваивание по несуществующему индексу приводит к созданию нового   
элемента. Ключи можно создавать при определении словаря в виде лите-  
рала (в этом случае они встраиваются непосредственно в литерал) или при   
присваивании значения новому ключу существующего объекта словаря.   
Результат получается тот же самый.  
 •  
Ключи не обязательно должны быть строками. В наших примерах в каче-  
стве ключей использовались строки, но могут использоваться любые другие   
неизменяемые объекты (то есть не списки). Например, в качестве ключей   
допустимо использовать целые числа, что превращает словарь в подобие   
списка (как минимум, в смысле индексирования). В качестве ключей мож-  
но также использовать кортежи, что позволяет создавать составные ключи.   
Экземпляры классов (обсуждаются в четвертой части книги) также могут   
играть роль ключей при условии, что они поддерживают определенные ме-  
тоды, которые сообщат интерпретатору, что он имеет дело с неизменяемым   
объектом, в противном случае они будут бесполезны, если рассматривать   
их как фиксированные ключи.  
Использование словарей для имитации гибких списков  
Последнее замечание в предыдущем списке имеет настолько важное значение,   
что имеет смысл продемонстрировать его применение на нескольких приме-  
рах. Списки не допускают возможность присваивания по индексам, находя-  
щимся за пределами списков:  
>>> L = []  
>>> L[99] = ‘spam’  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in ?  
IndexError: list assignment index out of range  
Можно, конечно, с помощью операции повторения создать список достаточно   
большой длины (например, [0]\*100), но можно создать нечто похоже, задейство-

272   
Глава 8. Списки и словари   
вав словарь, который не требует такого выделения пространства. При исполь-  
зовании целочисленных ключей словари могут имитировать списки, которые   
увеличиваются при выполнении операции присваивания по смещению:  
>>> D = {}  
>>> D[99] = ‘spam’  
>>> D[99]  
‘spam’  
>>> D  
{99: ‘spam’}  
Результат выглядит так, как если бы D был списком из 100 элементов, но на   
самом деле это словарь с единственным элементом – значением ключа 99 явля-  
ется строка ‘spam’. В такой структуре можно обращаться по смещениям, как   
в списке, но при этом не требуется выделять пространство для всех позиций,   
которые могут когда-либо потребоваться при выполнении программы. При ис-  
пользовании подобным образом словари представляют собой более гибкие эк-  
виваленты списков.  
Использование словарей для структур разреженных данных  
Похожим образом словари могут использоваться для реализации структур раз-  
реженных данных, таких как многомерные массивы, где всего несколько эле-  
ментов имеют определенные значения:  
>>> Matrix = {}  
>>> Matrix[(2, 3, 4)] = 88  
>>> Matrix[(7, 8, 9)] = 99  
>>>  
>>> X = 2; Y = 3; Z = 4 # символ ; отделяет инструкции  
>>> Matrix[(X, Y, Z)]  
88  
>>> Matrix  
{(2, 3, 4): 88, (7, 8, 9): 99}  
Здесь словарь использован для представления трехмерного массива, в котором   
только два элемента, (2,3,4) и (7,8,9), имеют определенные значения. Ключами   
словаря являются кортежи, определяющие координаты непустых элементов.   
Благодаря этому вместо трехмерной матрицы, объемной и по большей части   
пустой, оказалось достаточно использовать словарь из двух элементов. В такой   
ситуации попытка доступа к пустым элементам будет приводить к возбужде-  
нию исключения, так как эти элементы физически отсутствуют:  
>>> Matrix[(2,3,6)]  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in ?  
KeyError: (2, 3, 6)  
Как избежать появления ошибок обращения   
к несуществующему ключу  
Ошибки обращения к несуществующему ключу являются обычными при ра-  
боте с разреженными матрицами, но едва ли кто-то захочет, чтобы они при-  
водили к преждевременному завершению программы. Существует по край-  
ней мере три способа получить значение по умолчанию вместо возбуждения   
исключения – можно предварительно проверить наличие ключа с помощью

Словари в действии   
273  
условного оператора if, воспользоваться конструкцией try, чтобы перехватить   
и явно обработать исключение, или просто использовать представленный ра-  
нее метод словаря get, способный возвращать значение по умолчанию для не-  
существующих ключей:  
>>> if (2,3,6) in Matrix: # Проверить наличие ключа перед обращением  
... print(Matrix[(2,3,6)]) # конструкция if/else описывается в главе 12   
... else:  
... print(0)  
...  
0  
>>> try:  
... print(Matrix[(2,3,6)]) # Попытаться обратиться по индексу  
... except KeyError: # Перехватить исключение и обработать  
... print(0)  
...  
0  
>>> Matrix.get((2,3,4), 0) # Существует; извлекается и возвращается  
88  
>>> Matrix.get((2,3,6), 0) # Отсутствует; используется аргумент default   
0  
Способ, основанный на применении метода get, является самым простым из   
приведенных, если оценивать объем программного кода, – инструкции if и try   
подробно будут рассматриваться далее в этой книге.  
Использование словарей в качестве «записей»  
Как видите, словари в языке Python способны играть множество ролей. Вооб-  
Python способны играть множество ролей. Вооб-  
 способны играть множество ролей. Вооб-  
ще говоря, они способны заменить реализацию алгоритмов поиска в структу-  
рах (потому что операция индексирования по ключу уже является операцией   
поиска) и могут представлять самые разные типы структурированной инфор-  
мации. Например, словари представляют один из многих способов описания   
свойств элементов в программах, то есть они могут играть ту же роль, какую   
играют «структуры» и «записи» в других языках программирования.  
В следующем примере выполняется заполнение словаря путем присваивания   
значений новым ключам в виде нескольких инструкций:  
>>> rec = {}  
>>> rec[‘name’] = ‘mel’  
>>> rec[‘age’] = 45  
>>> rec[‘job’] = ‘trainer/writer’  
>>>  
>>> print(rec[‘name’])  
mel  
Встроенные типы языка Python позволяют легко представлять структуриро-  
Python позволяют легко представлять структуриро-  
 позволяют легко представлять структуриро-  
ванную информацию� это особенно заметно, когда появляются уровни вложен-  
ности. В следующем примере снова используется словарь для хранения свойств   
объекта, но на этот раз заполнение производится в единственной инструкции   
(вместо того, чтобы выполнять присваивание каждому ключу в отдельности),   
причем здесь присутствуют вложенные список и словарь, чтобы обеспечить   
представление структурированных свойств объекта:  
>>> mel = {‘name’: ‘Mark’,  
... ‘jobs’: [‘trainer’, ‘writer’],

274   
Глава 8. Списки и словари   
... ‘web’: ‘www.rmi.net/~lutz’,  
... ‘home’: {‘state’: ‘CO’, ‘zip’:80513}}  
Чтобы извлечь компоненты вложенных объектов, достаточно просто объеди-  
 извлечь компоненты вложенных объектов, достаточно просто объеди-  
извлечь компоненты вложенных объектов, достаточно просто объеди-  
 компоненты вложенных объектов, достаточно просто объеди-  
компоненты вложенных объектов, достаточно просто объеди-  
 вложенных объектов, достаточно просто объеди-  
вложенных объектов, достаточно просто объеди-  
 объектов, достаточно просто объеди-  
объектов, достаточно просто объеди-  
, достаточно просто объеди-  
достаточно просто объеди-  
нить в цепочку операции индексирования:  
>>> mel[‘name’]  
‘Mark’  
>>> mel[‘jobs’]  
[‘trainer’, ‘writer’]  
>>> mel[‘jobs’][1]  
‘writer’  
>>> mel[‘home’][‘zip’]  
80513  
Хотя в четвертой части книги будет показано, что классы (объединяющие   
в себе данные и логику их обработки) еще лучше справляются с ролью запи-  
сей, в простых случаях словари являются простым и удобным инструментом.  
Придется держать в уме: интерфейсы словарей  
Помимо удобного способа хранения информации по ключам непосред-  
ственно в программе, некоторые расширения для Python также предо-  
ставляют интерфейсы, которые выглядят и действуют как словари. На-  
пример, обращение к индексированным файлам данных в формате DBM   
во многом напоминает обращение к словарю, который сначала требует-  
ся открыть. Строки сохраняются и извлекаются с помощью операции   
индексирования по ключу:  
import anydbm  
file = anydbm.open(“filename”) # Ссылка на файл  
file[‘key’] = ‘data’ # Сохранение данных по ключу  
data = file[‘key’] # Извлечение данных по ключу  
В главе 27 будет показано, как таким же способом можно сохранять це-  
лые объекты Python, достаточно лишь заменить имя модуля anydbm на   
shelve (shelves (хранилища) – это базы данных с доступом к информации   
по ключу, предназначенные для хранения объектов Python). Для ра-  
боты в Интернете поддержка CGI-сценариев, предусмотренная в языке   
Python, также обеспечивает интерфейс, напоминающий словарь. Вызов   
метода cgi.FieldStorage возвращает объект, по своим характеристикам   
напоминающий словарь, – с одной записью для каждого поля ввода, на-  
ходящегося на клиентской веб-странице:  
import cgi  
form = cgi.FieldStorage() # Анализирует данные формы  
if ‘name’ in form:  
 showReply(‘Hello, ‘ + form[‘name’].value)  
Все эти объекты (и словари в том числе) являются примерами отображе-  
ний. Как только вы овладеете словарными интерфейсами, вы обнару-  
жите, что они имеют отношение ко множеству встроенных инструмен-  
тов языка Python.

Словари в действии   
275  
Другие способы создания словарей  
Наконец, обратите внимание, что благодаря практической ценности словарей   
с течением времени способы их создания пополнялись. В Python 2.3 и более   
поздних версиях, например, последние два вызова конструктора dict (в дей-  
ствительности – имени типа) в следующем ниже примере имеют тот же эффект,   
что и литералы и форма присваивания по отдельным ключам в примере выше:  
{‘name’: ‘mel’, ‘age’: 45} # Традиционное литеральное выражение  
   
D = {} # Динамическое присваивание по ключам  
D[‘name’] = ‘mel’  
D[‘age’] = 45  
   
dict(name=’mel’, age=45) # Форма именованных аргументов  
   
dict([(‘name’, ‘mel’), (‘age’, 45)]) # Кортежи ключ/значение  
Все четыре варианта создают один и тот же словарь, содержащий два элемента,   
которые удобно использовать в следующих случаях:  
 •  
Первый вариант удобен, если содержимое всего словаря известно заранее.  
 •  
Второй вариант удобно использовать, когда необходимо динамически соз-  
давать словарь по одному полю за раз.  
 •  
Третий вариант с использованием именованных аргументов даже компакт-  
нее, чем литералы, но он требует, чтобы все ключи были строками.  
 •  
Последний вариант удобен, когда ключи и значения во время выполнения   
программы необходимо хранить в виде последовательностей.  
С именованными аргументами мы уже встречались ранее, когда рассматривали   
операцию сортировки. Третья форма, показанная в листинге, стала особенно   
популярной в последнее время, как наиболее компактная (и, как следствие, ме-  
нее подверженная ошибкам). Как было показано в конце табл. 8.2, последний   
вариант также часто используется в соединении с функцией zip, которая по-  
зволяет объединить отдельные списки ключей и значений, создаваемые дина-  
мически во время выполнения (например, при извлечении данных из столбцов   
в файле). Подробнее об этой возможности рассказывается в следующем разделе.  
Если значения всех ключей словаря остаются все время одними и теми же,   
можно использовать специализированную форму инициализации словаря,   
при использовании которой достаточно просто передать список ключей и на-  
чальное значение (по умолчанию используется значение None):  
>>> dict.fromkeys([‘a’, ‘b’], 0)  
{‘a’: 0, ‘b’: 0}  
В настоящее время вполне можно обойтись простыми литералами и операцией   
присваивания по ключам, но вы наверняка найдете применение всем упомяну-  
тым вариантам создания словарей, как только приступите к созданию настоя-  
щих, гибких и динамических программ на языке Python.  
В листингах, которые приводятся в этом разделе, представлены различные   
способы создания словарей, общие для обеих версий Python 2.6 и 3.0. Однако   
существует еще один способ создания словарей, доступный только в версии Py-  
Py-  
thon 3.0 (и выше): генератор словарей. Чтобы посмотреть, как используется эта   
форма, мы должны перейти к следующему разделу.

276   
Глава 8. Списки и словари   
Изменения в словарях в Python 3.0  
До сих пор в этой главе мы рассматривали основные операции над словарями,   
общие для обеих версий Python, однако в Python 3.0 функциональные возмож-  
ности словарей несколько изменились. Если вы пользуетесь интерпретатором   
версии 2.X, вы можете столкнуться с некоторыми особенностями словарей,   
которые либо изменились, либо вообще отсутствуют в версии 3.0. Кроме того,   
версия 3.0 расширяет словари дополнительными особенностями, недоступны-  
ми в Python 2.X. В частности, в версии 3.0 словари:  
 •  
Поддерживают новые выражения генераторов словарей, близко напомина-  
ющие генераторы списков и множеств.  
 •  
Методы D.keys, D.values и D.items возвращают итерируемые представления   
вместо списков.  
 •  
Вследствие особенности, описанной в предыдущем пункте, требуют по ино-  
му выполнять обход ключей в отсортированном порядке.  
 •  
Больше не поддерживают возможность непосредственного сравнивания   
между собой – теперь сравнивание должно выполняться вручную.  
 •  
Больше не поддерживают метод D.has\_key – вместо него следует использо-  
вать оператор in проверки на вхождение.  
Давайте посмотрим, что нового появилось в словарях с выходом версии 3.0.  
Генераторы словарей  
Как отмечалось в конце предыдущего раздела, в версии 3.0 появилась возмож-  
ность создавать словари с помощью генераторов словарей. Как и генераторы   
множеств, с которыми мы встречались в главе 5, генераторы словарей доступ-  
ны только в версии 3.0 (они недоступны в версии 2.6). Подобно давно суще-  
ствующим генераторам списков, с которыми мы встречались в главе 4 и ранее   
в этой главе, генераторы словарей выполняют цикл, отбирают пары ключ/зна-  
чение в каждой итерации и заполняют ими новый словарь. Значения, получае-  
мые в ходе итераций, доступны в виде переменной цикла.  
Например, одним из стандартных способов динамической инициализации   
словаря в версиях 2.6 и 3.0 является использование функции zip для объеди-  
нения ключей и значений с последующей передачей результата функции dict.   
Как мы узнаем в главе 13, функция zip позволяет единственным вызовом соз-  
дать словарь из списков ключей и значений. Если множество ключей и значе-  
ний заранее не известно, вы всегда можете поместить их в списки в процессе   
вычислений и затем объединить их воедино:  
>>> list(zip([‘a’, ‘b’, ‘c’], [1, 2, 3])) # Объединить ключи и значения  
[(‘a’, 1), (‘b’, 2), (‘c’, 3)]  
   
>>> D = dict(zip([‘a’, ‘b’, ‘c’], [1, 2, 3])) # Создать словарь из результата   
>>> D # вызова функции zip  
{‘a’: 1, ‘c’: 3, ‘b’: 2}  
В Python 3.0 того же эффекта можно добиться с помощью генератора словарей.   
В следующем примере демонстрируется создание нового словаря из пар ключ/  
значение, которые извлекаются из результата вызова функции zip (это выра-  
жение читается практически точно так же, хотя и выглядит немного сложнее):

Словари в действии   
277  
C:\misc> c:\python30\python # Использование генератора словарей  
>>> D = {k: v for (k, v) in zip([‘a’, ‘b’, ‘c’], [1, 2, 3])}  
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2}  
Генераторы словарей выглядят менее компактными, но они гораздо более уни-  
версальны, чем можно было бы предположить, исходя из этого примера, – мы   
можем использовать их для отображения единственной последовательности   
значений в словари, вычисляя ключи с помощью выражений, как обычные   
значения:  
>>> D = {x: x \*\* 2 for x in [1, 2, 3, 4]} # Или: range(1, 5)  
>>> D  
{1: 1, 2: 4, 3: 9, 4: 16}  
   
>>> D = {c: c \* 4 for c in ‘SPAM’} # Цикл через итерируемый объект  
>>> D  
{‘A’: ‘AAAA’, ‘P’: ‘PPPP’, ‘S’: ‘SSSS’, ‘M’: ‘MMMM’}  
   
>>> D = {c.lower(): c + ‘!’ for c in [‘SPAM’, ‘EGGS’, ‘HAM’]}  
>>> D  
{‘eggs’: ‘EGGS!’, ‘ham’: ‘HAM!’, ‘spam’: ‘SPAM!’}  
Генераторы словарей удобно использовать для инициализации словарей из   
списков ключей, почти так же, как это делает метод fromkeys, с которым мы   
встречались в предыдущем разделе:  
>>> D = dict.fromkeys([‘a’, ‘b’, ‘c’], 0) # Инициализация списком ключей  
>>> D  
{‘a’: 0, ‘c’: 0, ‘b’: 0}  
   
>>> D = {k:0 for k in [‘a’, ‘b’, ‘c’]} # То же самое, но с помощью   
>>> D # генератора словаря  
{‘a’: 0, ‘c’: 0, ‘b’: 0}  
   
>>> D = dict.fromkeys(‘spam’) # Из другого итерируемого объекта,   
>>> D # используются значения по умолчанию  
{‘a’: None, ‘p’: None, ‘s’: None, ‘m’: None}  
   
>>> D = {k: None for k in ‘spam’}  
>>> D  
{‘a’: None, ‘p’: None, ‘s’: None, ‘m’: None}  
Подобно родственным инструментам, генераторы словарей поддерживают до-  
полнительные возможности, которые не были продемонстрированы здесь,   
включая вложенные циклы и условные инструкции if. К сожалению, чтобы   
полностью понять все возможности генераторов словарей, нам необходимо   
познакомиться поближе с концепцией и инструкциями итераций в языке   
Python, – но у нас пока недостаточно знаний, чтобы обсуждать эту тему. Мы   
познакомимся поближе со всеми разновидностями генераторов (списков, мно-  
жеств и словарей) в главах 14 и 20, поэтому отложим пока обсуждение деталей.   
Кроме того, в главе 13 мы поближе познакомимся со встроенной функцией zip,   
которую использовали в этом разделе, когда будем исследовать циклы for.  
Представления словарей  
В версии 3.0 методы словарей keys, values и items возвращают объекты пред-  
ставлений, тогда как в версии 2.6 они возвращают списки. Объекты представ-

278   
Глава 8. Списки и словари   
лений – это итерируемые объекты, то есть объекты, которые вместо всего спи-  
ска значений возвращают по одному значению за одно обращение. Кроме того,   
что они являются итерируемыми объектами, представления словарей также   
сохраняют оригинальный порядок следования компонентов словаря, отража-  
ют результаты операций, которые выполняются над словарем, и поддержива-  
ют операции над множествами. С другой стороны, они не являются списками   
и не поддерживают такие операции, как обращение к элементам по индексам   
или метод sort списков, а также не отображают значения своих элементов при   
выводе.  
Понятие итерируемого объекта более формально будет рассматриваться в гла-  
ве 14, а пока нам достаточно будет знать, что результаты этих трех методов   
необходимо обертывать вызовом встроенной функции list, – если появится   
необходимость применить операции над списками или отобразить значения   
элементов:  
>>> D = dict(a=1, b=2, c=3)  
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2}  
   
>>> K = D.keys() # В версии 3.0 создаст объект представления, а не список  
>>> K  
<dict\_keys object at 0x026D83C0>  
   
>>> list(K) # Принудительное создание списка в версии 3.0  
[‘a’, ‘c’, ‘b’]  
   
>>> V = D.values() # То же относится к представлениям значений и элементов  
>>> V  
<dict\_values object at 0x026D8260>  
   
>>> list(V)  
[1, 3, 2]  
   
>>> list(D.items())  
[(‘a’, 1), (‘c’, 3), (‘b’, 2)]  
   
>>> K[0] # Ошибка при попытке выполнить операцию над списком   
TypeError: ‘dict\_keys’ object does not support indexing  
>>> list(K)[0]  
‘a’  
Кроме случаев отображения результатов в интерактивной оболочке, вы едва ли   
будете обращать внимание на эту особенность, потому что конструкции обхо-  
да элементов в цикле в языке Python автоматически заставляют итерируемые   
объекты возвращать по одному результату в каждой итерации:  
>>> for k in D.keys(): print(k) # Работа с итераторами в циклах   
... # выполняется автоматически  
a  
c  
b  
Кроме того, в версии 3.0 словари по-прежнему остаются итерируемыми объ-  
ектами, которые последовательно возвращают ключи, как и в версии 2.6, – что   
исключает необходимость вызывать метод keys:  
>>> for key in D: print(key) # В итерациях по-прежнему не обязательно   
... # вызывать метод keys()

Словари в действии   
279  
a  
c  
b  
В отличие от списков, возвращаемых в виде результатов в версии 2.X, пред-  
X, пред-  
, пред-  
ставления словарей в версии 3.0 способны динамически отражать все после-  
дующие изменения в словарях, выполненные уже после создания объекта ото-  
бражения:  
>>> D = {‘a’:1, ‘b’:2, ‘c’:3}  
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2}  
   
>>> K = D.keys()  
>>> V = D.values()  
>>> list(K) # Представления сохраняют оригинальный   
[‘a’, ‘c’, ‘b’] # порядок следования ключей в словаре  
>>> list(V)  
[1, 3, 2]  
   
>>> del D[‘b’] # Изменяет словарь непосредственно  
>>> D  
{‘a’: 1, ‘c’: 3}  
   
>>> list(K) # Изменения в словаре отражаются на объектах представлений  
[‘a’, ‘c’]  
>>> list(V) # Но это не так в версии 2.X!  
[1, 3]  
Представления словарей и множества  
Кроме того, в отличие от списков результатов, возвращаемых в версии 2.X,   
объекты представлений в версии 3.0, возвращаемые методом keys, похожи на   
множества и поддерживают операции над множествами, такие как пересече-  
ние и объединение. Объекты представлений, возвращаемые методом values,   
такой особенностью не обладают, потому что они не являются уникальными,   
тогда как объекты представлений, возвращаемые методом items, такой особен-  
ностью обладают, если пары (key, value) являются уникальными и хешируе-  
мыми. Учитывая, что множества достаточно сильно похожи на словари, ключи   
которых не имеют значений (и даже литералы множеств в версии 3.0 заклю-  
чаются в фигурные скобки, как словари), это обстоятельство выглядит впол-  
не логичным. Подобно ключам словарей, элементы множеств неупорядочены,   
уникальны и относятся к разряду неизменяемых объектов.  
Ниже наглядно показано, как интерпретируются списки ключей, когда они   
используются в операциях над множествами. В таких операциях объекты   
представлений могут смешиваться с другими представлениями, множествами   
и словарями (в этом случае словари интерпретируются точно так же, как пред-  
ставления, возвращаемые методом keys):  
>>> K | {‘x’: 4} # Представления ключей (и иногда элементов)   
{‘a’, ‘x’, ‘c’} # похожи на множества  
   
>>> V & {‘x’: 4}  
TypeError: unsupported operand type(s) for &: ‘dict\_values’ and ‘dict’  
>>> V & {‘x’: 4}.values()  
TypeError: unsupported operand type(s) for &: ‘dict\_values’ and ‘dict\_values’

280   
Глава 8. Списки и словари   
>>> D = {‘a’:1, ‘b’:2, ‘c’:3}  
>>> D.keys() & D.keys() # Пересечение представлений ключей  
{‘a’, ‘c’, ‘b’}  
>>> D.keys() & {‘b’} # Пересечение представления ключей и множества  
{‘b’}  
>>> D.keys() & {‘b’: 1} # Пересечение представления ключей и словаря  
{‘b’}  
>>> D.keys() | {‘b’, ‘c’, ‘d’} # Объединение представления ключей и множества  
{‘a’, ‘c’, ‘b’, ‘d’}  
Представления элементов словарей также могут обладать свойствами мно-  
жеств, если они допускают возможность хеширования, – то есть, если они со-  
держат только неизменяемые объекты:  
>>> D = {‘a’: 1}  
>>> list(D.items()) # Элементы похожи на множества, если они допускают   
[(‘a’, 1)] # возможность хеширования  
>>> D.items() | D.keys() # Объединение представлений  
{(‘a’, 1), ‘a’}  
>>> D.items() | D # Словари интерпретируются как представление ключей  
{(‘a’, 1), ‘a’}  
   
>>> D.items() | {(‘c’, 3), (‘d’, 4)} # Множество пар ключ/значение  
{(‘a’, 1), (‘d’, 4), (‘c’, 3)}  
>>> dict(D.items() | {(‘c’, 3), (‘d’, 4)}) # Функция dict принимает также   
{‘a’: 1, ‘c’: 3, ‘d’: 4} # итерируемые объекты множеств  
За дополнительной информацией об операциях над множествами обращайтесь   
к главе 5. А теперь давайте коротко рассмотрим три другие особенности слова-  
рей, появившиеся в версии 3.0.  
Сортировка ключей словаря  
Во-первых, из-за того, что теперь метод keys не возвращает список, традици-  
онный прием просмотра содержимого словаря по отсортированным ключам,   
который используется в версии 2.X, непригоден в версии 3.0. Для этого необ-  
X, непригоден в версии 3.0. Для этого необ-  
, непригоден в версии 3.0. Для этого необ-  
ходимо либо преобразовать объект представления ключей в список, либо вос-  
пользоваться функцией sorted, представленной в главе 4 и выше в этой главе,   
применив ее к объекту, возвращаемому методу keys , или к самому словарю:  
>>> D = {‘a’:1, ‘b’:2, ‘c’:3}  
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2}  
   
>>> Ks = D.keys() # Сортировка объекта представления  
>>> Ks.sort() # не дает желаемого результата!  
AttributeError: ‘dict\_keys’ object has no attribute ‘sort’  
   
>>> Ks = list(Ks) # Преобразовать в список и потом отсортировать  
>>> Ks.sort()  
>>> for k in Ks: print(k, D[k])  
...  
a 1  
b 2  
c 3  
   
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2}

Словари в действии   
281  
>>> Ks = D.keys() # Или вызвать функцию sorted() с результатом вызова keys  
>>> for k in sorted(Ks): print(k, D[k]) # sorted() принимает итерируемые   
... # объекты и возвращает результат  
a 1  
b 2  
c 3  
   
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2} # Еще лучше отсортировать сам словарь  
>>> for k in sorted(D): print(k, D[k]) # Итератор словаря возвращает ключи  
...  
a 1  
b 2  
c 3  
Операция сравнения словарями больше не поддерживается   
Во-вторых, в Python 2.6 словари могут сравниваться между собой с помощью   
операторов <, > и других, но в Python 3.0 эта возможность больше не поддер-  
Python 3.0 эта возможность больше не поддер-  
 3.0 эта возможность больше не поддер-  
живается. Однако ее можно имитировать, сравнив отсортированные списки   
ключей вручную:  
sorted(D1.items()) < sorted(D2.items()) # То же, что и D1 < D2 в версии 2.6  
Тем не менее в версии 3.0 сохранилась возможность проверки словарей на ра-  
венство. Поскольку мы еще вернемся к этой теме в следующей главе, когда бу-  
дем рассматривать операции сравнения в более широком контексте, отложим   
до этого момента обсуждение деталей.  
Метод has\_key умер, да здравствует has\_key!   
Наконец, широко используемый метод has\_key словарей, выполняющий про-  
верку наличия ключей, был ликвидирован в версии 3.0. Вместо него рекомен-  
дуется использовать оператор in проверки на вхождение или проверять резуль-  
тат вызова метода get на равенство значению по умолчанию (первый вариант   
предпочтительнее):  
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2}  
   
>>> D.has\_key(‘c’) # Только в версии 2.X: True/False  
AttributeError: ‘dict’ object has no attribute ‘has\_key’  
   
>>> ‘c’ in D  
True  
>>> ‘x’ in D  
False  
>>> if ‘c’ in D: print(‘present’, D[‘c’]) # Предпочтительнее в версии 3.0  
...  
present 3  
   
>>> print(D.get(‘c’))  
3  
>>> print(D.get(‘x’))  
None  
>>> if D.get(‘c’) != None: print(‘present’, D[‘c’]) # Еще одна возможность  
...  
present 3

282   
Глава 8. Списки и словари   
Если вы пользуетесь версией 2.6 и вас волнует вопрос совместимости с верси-  
ей 3.0, учтите, что первые два изменения (генераторы словарей и представле-  
ния) доступны только в версии 3.0, а последние три (функция sorted, сравнива-  
ние вручную и оператор in) могут использоваться и в версии 2.6, что поможет   
облегчить переход на версию 3.0 в будущем.  
В заключение  
В этой главе мы исследовали списки и словари – два, пожалуй, наиболее ча-  
сто используемых, гибких и мощных типа коллекций, которые можно увидеть   
в программах на языке Python. Мы узнали, что списки представляют собой   
упорядоченные по позициям коллекции объектов произвольных типов и что   
они могут иметь неограниченное число уровней вложенности, могут увели-  
чиваться и уменьшаться в размерах по мере необходимости и многое другое.   
Словари представляют похожий тип данных, но в них элементы сохраняются   
по ключам, а не по позициям� словари не обеспечивают какой-либо надежный   
способ поддержки упорядочения элементов слева направо. И списки, и словари   
относятся к категории изменяемых объектов и поэтому поддерживают различ-  
ные операции непосредственного их изменения, недоступные для строк, на-  
пример списки можно увеличивать в размерах с помощью метода append, а сло-  
вари – за счет выполнения операции присваивания по новому ключу.  
В следующей главе мы закончим подробное исследование базовых типов объ-  
ектов, рассмотрев кортежи и файлы. После этого мы перейдем к инструкци-  
ям, которые реализуют логику обработки этих объектов, что позволит нам еще   
больше приблизиться к написанию полноценных программ. Но прежде чем за-  
няться этими темами, ответьте на контрольные вопросы главы.  
Закрепление пройденного  
Контрольные вопросы  
1. Назовите два способа создания списка, содержащего пять целочисленных   
значений, равных нулю.  
2. Назовите два способа создания словаря с двумя ключами ‘a’ и ‘b’, каждый   
из которых ассоциирован со значением 0.  
3. Назовите четыре операции, которые изменяют непосредственно объект спи-  
ска.  
4. Назовите четыре операции, которые изменяют непосредственно объект сло-  
варя.  
Ответы  
1. Литеральное выражение, такое как [0, 0, 0, 0, 0], и операция повторения,   
такая как [0]\*5, создадут список с пятью элементами, содержащими нули.   
Формально можно было бы построить список с помощью цикла, начиная   
с пустого списка, к которому добавляется значение 0 на каждой итерации:   
L.append(0). Генератор списков ([0 for i in range(5)]) сделал бы то же самое, но   
в этом случае пришлось бы выполнить лишнюю работу.

Закрепление пройденного   
283  
2. Литеральное выражение, например {‘a’: 0, ‘b’: 0}, или последовательность   
операций присваивания, таких как D = {}; D[‘a’] = 0, D[‘b’] = 0, создадут тре-  
буемый словарь. Можно также использовать более новый и более простой   
способ с использованием именованных аргументов dict(a=0, b=0) или бо-  
лее гибкий вариант с указанием последовательностей пар ключ/значение   
dict([(‘a’, 0), (‘b’, 0)]). Или, поскольку все ключи известны заранее и не бу-  
дут изменяться в дальнейшем, можно использовать специализированную   
форму dict.fromkeys([‘a’, ‘b’], 0). В версии 3.0 можно также воспользоваться   
генератором словарей: {k:0 for k in ‘ab’}.  
3. Методы append и extend увеличивают размер самого списка, методы sort   
и reverse упорядочивают и изменяют порядок следования элементов списка   
на обратный, метод insert вставляет элемент в указанную позицию, мето-  
ды remove и pop удаляют элементы списка по значениям и по смещению, ин-  
струкция del удаляет элемент списка или срез, а операции присваивания по   
индексу или срезу замещают элемент или целый сегмент списка. Для пра-  
вильного ответа на вопрос выберите четыре любых метода из указанных.  
4. Словари прежде всего изменяются с помощью инструкции присваивания   
по новому или по существующему ключу, что приводит к созданию или   
изменению записи в таблице. Кроме того, инструкция del удаляет ключ,   
метод update вклеивает один словарь в другой, а вызов D.pop(key) удаляет   
ключ и возвращает его значение. Словари имеют и другие, более необычные   
методы изменения, которые не были перечислены в этой главе, такие как   
setdefault. За дополнительной информацией обращайтесь к справочным ру-  
ководствам.

Глава 9.  
   
Кортежи, файлы и все остальное  
Эта глава завершает наше детальное исследование базовых типов объектов язы-  
ка Python рассмотрением кортежей – коллекций объектов, которые не могут   
изменяться, и файлов – интерфейсов к внешним файлам в вашем компьютере.   
Как вы узнаете, кортежи – это относительно простые объекты, поддерживаю-  
щие операции, которые по большей части вам уже знакомы по строкам и спи-  
скам. Объекты-файлы – это широко используемые многофункциональные ин-  
струменты для работы с файлами – краткий обзор файлов, который приводит-  
ся здесь, будет дополнен примерами их использования в последующих главах   
этой книги.   
Кроме того, эта глава завершает данную часть книги рассмотрением свойств,   
общих для объектов всех базовых типов, с которыми мы познакомились, – по-  
нятия равенства, сравнения, копирования объектов и так далее. Мы также   
кратко познакомимся и с другими типами объектов, присутствующими в ар-  
сенале Python, – несмотря на то, что мы рассмотрели все основные встроенные   
типы, спектр объектов в языке Python гораздо шире, чем я давал вам основа-  
ния полагать к этому моменту. В заключение мы закроем эту часть книги из-  
учением связанных с типами объектов ловушек, в которые часто попадаются   
программисты, и исследуем некоторые примеры, которые позволят вам поэк-  
спериментировать с освоенными идеями.1  
Кортежи  
Последний тип коллекций в нашем обзоре – это кортежи. Кортежи представ-  
ляют собой простые группы объектов. Они действуют точно так же, как спи-  
ски, за исключением того, что не допускают непосредственного изменения (они   
являются неизменяемыми) и в литеральной форме записываются как после-  
довательность элементов в круглых, а не в квадратных скобках. Хотя корте-  
1   
После детального изучения удобно пользоваться краткими справочниками, напри-  
мер представленными в справочном разделе официального сайта проекта Python.   
Особо стоит отметить Мастерскую отца-основателя Python (Python �orkshop by   
Guido van Rossum). Презентация доступна по адресу: http://www.python.org/doc/  
essays/ppt/acm-ws/. – Примеч. перев.

Кортежи   
285  
жи и не поддерживают многих методов списков, тем не менее они обладают   
большинством свойств, присущим спискам. Ниже коротко рассматриваются   
их свойства. Кортежи:  
Это упорядоченные коллекции объектов произвольных типов  
Подобно строкам и спискам, кортежи являются коллекциями объектов,   
упорядоченных по позициям (то есть они обеспечивают упорядочение свое-  
го содержимого слева направо). Подобно спискам, они могут содержать объ-  
екты любого типа.  
Обеспечивают доступ к элементам по смещению  
Подобно строками и спискам, доступ к элементам кортежей осуществляет-  
ся по смещению (а не по ключу) – они поддерживают все операции, которые   
основаны на использовании смещения, такие как индексирование и извле-  
чение среза.  
Относятся к категории неизменяемых последовательностей  
Подобно строкам и спискам, кортежи являются последовательностями   
и поддерживают многие операции над последовательностями. Однако, по-  
добно строкам, кортежи являются неизменяемыми объектами, поэтому   
они не поддерживают никаких операций непосредственного изменения, ко-  
торые применяются к спискам.  
Имеют  фиксированную  длину,  гетерогенны  и  поддерживают  произвольное   
число уровней вложенности  
Поскольку кортежи являются неизменяемыми объектами, вы не можете   
изменить размер кортежа, минуя процедуру создания копии. С другой сто-  
роны, кортежи могут хранить другие составные объекты (то есть списки,   
словари и другие кортежи), а следовательно, поддерживают произвольное   
число уровней вложенности.  
Массивы ссылок на объекты  
Подобно спискам, кортежи проще представлять, как массивы ссылок на   
объекты, – кортежи хранят указатели (ссылки) на другие объекты, а опера-  
ция индексирования над кортежами выполняется очень быстро.  
В табл. 9.1 приводятся наиболее часто используемые операции над кортежами.   
В программном коде кортежи записываются как последовательность объектов   
(точнее, выражений, которые создают объекты), разделенных запятыми, за-  
ключенная в круглые скобки. Пустые кортежи определяются как пара пустых   
круглых скобок.  
Таблица 9.1. Литералы кортежей и операции  
Операция  
Интерпретация  
()  
Пустой кортеж  
T = (0,)  
Кортеж из одного элемента (не выражение)  
T = (0, ‘Ni’, 1.2, 3)  
Кортеж из четырех элементов  
T = 0, ‘Ni’, 1.2, 3  
Еще один кортеж из четырех элементов   
(тот же самый, что и строкой выше)  
T = (‘abc’, (‘def’,   
‘ghi’))  
Вложенные кортежи

286   
Глава 9. Кортежи, файлы и все остальное   
Операция  
Интерпретация  
T = tuple(‘spam’)  
Создание кортежа из итерируемого объекта  
T[i]  
T[i][j]  
T[i:j]  
len(T)  
Индекс, индекс индекса, срез, длина  
T1 + T2  
T \* 3  
Конкатенация, повторение  
for x in T: print(x)  
‘spam’ in t2  
[x \*\* 2 for x in T]  
Обход в цикле, проверка вхождения  
T.index(‘Ni’)  
T.count(‘Ni’)  
Методы кортежей в версиях 2.6 и 3.0: поиск,   
подсчет вхождений  
Кортежи в действии  
Как обычно, запустите интерактивный сеанс работы с интерпретатором   
Python, чтобы приступить к исследованию кортежей в действии. Обратите   
внимание: как указано в табл. 9.1, кортежи не обладают методами, которые   
имеются у списков (например, кортежи не имеют метода append). Зато кортежи   
поддерживают обычные операции над последовательностями, которые приме-  
няются к строкам и к спискам:  
>>> (1, 2) + (3, 4) # Конкатенация  
(1, 2, 3, 4)  
   
>>> (1, 2) \* 4 # Повторение  
(1, 2, 1, 2, 1, 2, 1, 2)  
   
>>> T = (1, 2, 3, 4) # Индексирование, извлечение среза  
>>> T[0], T[1:3]  
(1, (2, 3))  
Особенности синтаксиса определения кортежей: запятые   
и круглые скобки  
Вторая и четвертая строки в табл. 9.1 заслуживают дополнительных поясне-  
ний. Поскольку круглые скобки могут также окружать выражения (глава 5),   
необходимо что-то предпринять, чтобы дать интерпретатору понять, что един-  
ственный объект в круглых скобках – это кортеж, а не простое выражение.   
Если вам действительно необходимо получить кортеж с единственным элемен-  
том, нужно просто добавить запятую после этого элемента, перед закрываю-  
щей круглой скобкой:  
>>> x = (40) # Целое число  
>>> x  
40  
Таблица 9.1 (продолжение)

Кортежи в действии   
287  
>>> y = (40,) # Кортеж, содержащий целое число  
>>> y  
(40,)  
В виде исключения при определении кортежей интерпретатор позволяет опу-  
скать открывающую и закрывающую круглые скобки, если синтаксически   
конструкция интерпретируется однозначно. Например, в четвертой строке   
таблицы кортеж создается простым перечислением четырех элементов, раз-  
деленных запятыми. В контексте операции присваивания интерпретатор рас-  
познает, что это кортеж, даже при отсутствии круглых скобок.  
Кто-то может посоветовать всегда заключать кортежи в круглые скобки, а кто-  
то – посоветовать никогда не использовать их (найдутся и те, кто вообще ни-  
чего не скажет, что делать с кортежами!). Единственное место, где круглые   
скобки являются обязательными, – при передаче кортежей функциям в виде   
литералов (где круглые скобки имеют важное значение) и при передаче их ин-  
струкции print в версии Python 2.X (где важное значение имеют запятые).  
Начинающим программистам можно посоветовать следующее – вероятно,   
легче использовать круглые скобки, чем выяснять ситуации, когда они могут   
быть опущены. Многие также считают, что круглые скобки повышают удобо-  
читаемость сценариев, делая кортежи в программном коде более заметными,   
но у вас может быть свое собственное мнение на этот счет.  
Преобразования, методы и неизменяемость  
Несмотря на отличия в синтаксисе литералов, операции, выполняемые над   
кортежами (последние три строки в табл. 9.1), идентичны операциям, приме-  
няемым к строкам и спискам. Единственное отличие состоит в том, что опе-  
рации +, \* и извлечения среза при применении к кортежам возвращают новые   
кортежи, а также в том, что в отличие от строк, списков и словарей, кортежи   
имеют сокращенный набор методов. Если, к примеру, необходимо отсортиро-  
вать содержимое кортежа, его сначала следует преобразовать в список, чтобы   
превратить в изменяемый объект и получить доступ к методу сортировки или   
задействовать новую функцию sorted, которая принимает объекты любых ти-  
пов последовательностей (и не только):  
>>> T = (‘cc’, ‘aa’, ‘dd’, ‘bb’)  
>>> tmp = list(T) # Создать список из элементов кортежа  
>>> tmp.sort() # Отсортировать списка  
>>> tmp  
[‘aa’, ‘bb’, ‘cc’, ‘dd’]  
>>> T = tuple(tmp) # Создать кортеж из элементов списка  
>>> T  
(‘aa’, ‘bb’, ‘cc’, ‘dd’)  
   
>>> sorted(T) # Или использовать встроенную функцию sorted  
[‘aa’, ‘bb’, ‘cc’, ‘dd’]  
Здесь list и tuple – это встроенные функции, которые используются для преоб-  
разования в список и затем обратно в кортеж. В действительности обе функции   
создают новые объекты, но благодаря им создается эффект преобразования.  
Для преобразования кортежей можно также использовать генераторы спи-  
сков. Например, ниже из кортежа создается список, причем попутно к каждо-  
му элементу прибавляется число 20:

288   
Глава 9. Кортежи, файлы и все остальное   
>>> T = (1, 2, 3, 4, 5)  
>>> L = [x + 20 for x in T]  
>>> L  
[21, 22, 23, 24, 25]  
Генераторы списков в действительности являются операциями над последова-  
тельностями – они всегда создают новые списки, но они могут использоваться   
для обхода содержимого любых объектов последовательностей, включая кор-  
тежи, строки и другие списки. Как будет показано дальше, они могут приме-  
няться даже к программным компонентам, которые физически не являются   
последовательностями, – к любым объектам, поддерживающим возможность   
выполнения итераций, включая файлы, которые автоматически читаются   
строка за строкой.  
Кортежи имеют весьма ограниченный набор методов, по сравнению со списка-  
ми и строками. В версиях Python 2.6 и 3.0 они обладают всего двумя метода-  
Python 2.6 и 3.0 они обладают всего двумя метода-  
 2.6 и 3.0 они обладают всего двумя метода-  
ми – index и count, которые действуют точно так же, как одноименные методы   
списков:  
>>> T = (1, 2, 3, 2, 4, 2) # Методы кортежей в Python 2.6 и 3.0  
>>> T.index(2) # Первое вхождение находится в позиции 2  
1  
>>> T.index(2, 2) # Следующее вхождение за позицией 2  
3  
>>> T.count(2) # Определить количество двоек в кортеже  
3  
До выхода версий 2.6 и 3.0 кортежи вообще не имели методов – это было древ-  
нее соглашение в языке Python, касающееся неизменяемых типов, которое   
несколько лет тому назад было из практических соображений нарушено для   
строк, а совсем недавно – для чисел и кортежей.  
Кроме того, следует заметить, что правило неизменяемости применяется   
только к самому кортежу, но не к объектам, которые он содержит. Например,   
список внутри кортежа может изменяться как обычно:  
>>> T = (1, [2, 3], 4)  
   
>>> T[1] = ‘spam’ # Ошибка: нельзя изменить сам кортеж  
TypeError: object doesn’t support item assignment  
   
>>> T[1][0] = ‘spam’ # Допустимо: вложенный изменяемый объект можно изменить  
>>> T  
(1, [‘spam’, 3], 4)  
Для большинства программ такой одноуровневой неизменяемости для обычно-  
го использования кортежей вполне достаточно. О чем, совершенно случайно,   
и рассказывается в следующем разделе.  
Зачем нужны кортежи, если есть списки?  
Похоже, что это самый первый вопрос, который задают начинающие програм-  
мисты, узнав о кортежах: «Зачем нам нужны кортежи, если у нас уже имеются   
списки?» Некоторые из причин корнями уходят в прошлое. Создатель языка   
Python – математик по образованию, и он рассматривал кортежи, как простые   
ассоциации объектов, а списки – как структуры данных, допускающие изме-  
нения в течение всего времени своего существования. На самом деле, такое ис-

Файлы   
289  
пользование слова «кортеж» уходит корнями в математику, так же, как и его   
использование для обозначения строк в таблицах реляционных баз данных.  
Однако более правильным будет считать, что неизменяемость кортежей обе-  
спечивает своего рода поддержку целостности – вы можете быть уверены, что   
кортеж не будет изменен посредством другой ссылки из другого места в про-  
грамме, чего нельзя сказать о списках. Тем самым кортежи играют роль объ-  
явлений «констант», присутствующих в других языках программирования,   
несмотря на то, что в языке Python это понятие связано с объектами, а не с пе-  
ременными.  
Кроме того, существуют ситуации, в которых кортежи можно использовать,   
а списки – нет. Например, в качестве ключей словаря (пример с разреженными   
матрицами в главе 8). Некоторые встроенные операции также могут требовать   
или предполагать использование кортежей, а не списков. Следует запомнить,   
что списки должны выбираться, когда требуются упорядоченные коллек-  
ции, которые может потребоваться изменить. Кортежи могут использоваться   
в остальных случаях, когда необходимы фиксированные ассоциации объектов.  
Файлы  
Возможно, вы уже знакомы с понятием файла – так называются именованные   
области постоянной памяти в вашем компьютере, которыми управляет опера-  
ционная система. Последний основной встроенный тип объектов, который мы   
исследуем в нашем обзоре, обеспечивает возможность доступа к этим файлам   
из программ на языке Python.  
Проще говоря, встроенная функция open создает объект файла, который обе-  
спечивает связь с файлом, размещенным в компьютере. После вызова функции   
open можно выполнять операции чтения и записи во внешний файл, используя   
методы полученного объекта.   
По сравнению с типами, с которыми вы уже знакомы, объекты файлов выгля-  
дят несколько необычно. Они не являются ни числами, ни последовательно-  
стями или отображениями – для задач работы с файлами они предоставляют   
одни только методы. Большинство методов файлов связаны с выполнением   
операций ввода-вывода во внешние файлы, ассоциированные с объектом, но   
существуют также методы, которые позволяют переходить на другую позицию   
в файле, выталкивать на диск буферы вывода и так далее. В табл. 9.2 приводят-  
ся наиболее часто используемые операции над файлами.  
Таблица 9.2. Часто используемые операции над файлами  
Операция  
Интерпретация  
output = open(r’C:\spam’, ‘w’)  
Открывает файл для записи   
(‘w’ означает write – запись)  
input = open(‘data’, ‘r’)  
Открывает файл для чтения   
(‘r’ означает read – чтение)  
input = open(‘data’)  
То же самое, что и в предыдущей строке   
(режим ‘r’ используется по умолчанию)  
aString = input.read()  
Чтение файла целиком в единственную строку

290   
Глава 9. Кортежи, файлы и все остальное   
Операция  
Интерпретация  
aString = input.read(N)  
Чтение следующих N символов (или байтов)   
в строку  
aString = input.readline()  
Чтение следующей текстовой строки   
(включая символ конца строки) в строку  
aList = input.readlines()  
Чтение файла целиком в список строк   
(включая символ конца строки)  
output.write(aString)  
Запись строки символов (или байтов) в файл  
output.writelines(aList)  
Запись всех строк из списка в файл  
output.close()  
Закрытие файла вручную (выполняется по   
окончании работы с файлом)  
output.flush()  
Выталкивает выходные буферы на диск,   
файл остается открытым  
anyFile.seek(N)  
Изменяет текущую позицию в файле для сле-  
дующей операции, смещая ее на N байтов от   
начала файла.  
for line in open(‘data’):   
операции над line  
Итерации по файлу, построчное чтение  
open(‘f.txt’,   
encoding=’latin-1’)  
Файлы с текстом Юникода в Python 3.0   
(строки типа str)  
open(‘f.bin’, ‘rb’)  
Файлы с двоичными данными в Python 3.0   
(строки типа bytes)  
Открытие файлов  
Чтобы открыть файл, программа должна вызвать функцию open, передав ей   
имя внешнего файла и режим работы. Обычно в качестве режима использует-  
ся строка ‘r’, когда файл открывается для чтения (по умолчанию), ‘w’ – когда   
файл открывается для записи или ‘a’ – когда файл открывается на запись в ко-  
нец. В строке режима могут также указываться другие параметры:  
 •  
Добавление символа b в строку режима означает работу с двоичными дан-  
ными (в версии 3.0 отключается интерпретация символов конца строки   
и кодирование символов Юникода).  
 •  
Добавление символа + означает, что файл открывается для чтения и для за-  
писи (то есть вы получаете возможность читать и записывать данные в один   
и тот же объект файла, часто совместно с операцией позиционирования   
в файле).  
Оба аргумента функции open должны быть строками. Кроме того, функция мо-  
жет принимать третий необязательный аргумент, управляющий буферизаци-  
ей выводимых данных, – значение ноль означает, что выходная информация   
не будет буферизироваться (то есть она будет записываться во внешний файл   
сразу же, в момент вызова метода записи). Имя внешнего файла может вклю-  
Таблица 9.2 (продолжение)

Файлы   
291  
чать платформозависимые префиксы абсолютного или относительного пути   
к файлу. Если путь к файлу не указан, предполагается, что он находится в те-  
кущем рабочем каталоге (то есть в каталоге, где был запущен сценарий). Здесь   
мы рассмотрим лишь самые основы работы с файлами и исследуем несколько   
простых примеров, но не будем исследовать все параметры, определяющие ре-  
жимы работы с файлами. За дополнительной информацией обращайтесь к ру-  
ководству по стандартной библиотеке языка Python.  
Использование файлов  
Как только будет получен объект файла, вы можете вызывать его методы для   
выполнения операций чтения или записи. В любом случае содержимое файла   
в программах на языке Python принимает форму строк – операция чтения воз-  
вращает текст в строках, и метод записи принимает информацию в виде строк.   
Существует несколько разновидностей методов чтения и записи, а в табл. 9.2   
перечислены лишь наиболее часто используемые из них. Ниже приводятся не-  
сколько самых основных замечаний по использованию файлов:  
Для чтения строк лучше использовать итераторы файлов  
Несмотря на то что методы чтения и записи, перечисленные в таблице, яв-  
ляются наиболее часто используемыми, имейте в виду, что самый лучший,   
пожалуй, способ чтения строк из файла на сегодняшний день состоит в том,   
вообще не использовать операцию чтения из файла – как будет показано   
в главе 14, файлы имеют итератор, который автоматически читает инфор-  
мацию из файла строку за строкой в контексте цикла for, в генераторах спи-  
сков и в других итерационных контекстах.  
Содержимое файлов находится в строках, а не в объектах  
Обратите внимание: в табл. 9.2 показано, что данные, получаемые из файла,   
всегда попадают в сценарий в виде строки, поэтому вам необходимо будет   
выполнять преобразование данных в другие типы объектов языка Python,   
если эта форма представления вам не подходит. Точно так же, при выпол-  
нении операции записи данных в файл, в отличие от инструкции print,   
интерпретатор Python не выполняет автоматическое преобразование объ-  
ектов в строки – вам необходимо передавать методам уже сформированные   
строки. Поэтому при работе с файлами вам пригодятся рассматривавшиеся   
ранее инструменты преобразования данных из строкового представления   
в числовое и наоборот (например, int, float, str, а также выражения форма-  
тирования строк и метод format). Кроме того, в состав Python входят допол-  
нительные стандартные библиотечные инструменты, предназначенные для   
работы с универсальным объектом хранилища данных (например, модуль   
pickle) и обработки упакованных двоичных данных в файлах (например,   
модуль struct). С действием обоих модулей мы познакомимся ниже, в этой   
же главе.  
Вызов метода close является необязательным  
Вызов метода close разрывает связь с внешним файлом. Как рассказыва-  
лось в главе 6, интерпретатор Python немедленно освобождает память, за-  
нятую объектом, как только в программе будет утеряна последняя ссылка   
на этот объект. Как только объект файла освобождается, интерпретатор   
автоматически закрывает ассоциированный с ним файл (что происходит   
также в момент завершения программы). Благодаря этому вам не требу-

292   
Глава 9. Кортежи, файлы и все остальное   
ется закрывать файл вручную, особенно в небольших сценариях, которые   
выполняются непродолжительное время. С другой стороны, вызов метода   
close не повредит, и его рекомендуется использовать в крупных системах.   
Строго говоря, возможность автоматического закрытия файлов не являет-  
ся частью спецификации языка, и с течением времени такое поведение мо-  
жет измениться. Следовательно, привычку вызывать метод close вручную   
можно только приветствовать. (Альтернативный способ автоматического   
закрытия файлов приводится ниже, в этом же разделе, где обсуждаются   
менеджеры  контекста объектов файлов, которые используются в новой   
инструкции with/as, появившейся в Python 2.6 и 3.0.)  
Файлы обеспечивают буферизацию ввода-вывода и позволяют производить по-  
зиционирование в файле  
В предыдущем абзаце отмечается важность явного закрытия файлов, по-  
тому что в этот момент освобождаются ресурсы операционной системы   
и выталкиваются выходные буферы. По умолчанию вывод в файлы всегда   
выполняется с помощью промежуточных буферов, то есть в момент записи   
текста в файл он не попадает сразу же на диск – буферы выталкиваются   
на диск только в момент закрытия файла или при вызове метода flush. Вы   
можете отключить механизм буферизации с помощью дополнительных па-  
раметров функции open, но это может привести к снижению производитель-  
ности операций ввода-вывода. Файлы в языке Python поддерживают также   
возможность позиционирования – метод seek позволяет сценариям управ-  
лять позицией чтения и записи.  
Файлы в действии  
Давайте рассмотрим небольшие примеры, демонстрирующие основы работы   
с файлами. В первом примере выполняется открытие нового текстового файла   
в режиме для записи, в него записываются две строки (завершающиеся сим-  
волом новой строки \n), после чего файл закрывается. Далее этот же файл от-  
крывается в режиме для чтения и выполняется чтение строк из него. Обратите   
внимание, что третий вызов метода readline возвращает пустую строку – таким   
способом методы файлов в языке Python сообщают о том, что был достигнут ко-  
нец файла (пустая строка в файле возвращается как строка, содержащая един-  
ственный символ новой строки, а не как действительно пустая строка). Ниже   
приводится полный листинг сеанса:  
>>> myfile = open(‘myfile.txt’, ‘w’) # Открывает файл (создает/очищает)  
>>> myfile.write(‘hello text file\n’) # Записывает строку текста  
16  
>>> myfile.write(‘goodbye text file\n’)  
18  
>>> myfile.close() # Выталкивает выходные буферы на диск  
   
>>> myfile = open(‘myfile.txt’) # Открывает файл: ‘r’ – по умолчанию  
>>> myfile.readline() # Читает строку  
‘hello text file\n’  
>>> myfile.readline()  
‘goodbye text file\n’  
>>> myfile.readline() # Пустая строка: конец файла  
‘’

Файлы   
293  
Обратите внимание, что в Python 3.0 метод write возвращает количество запи-  
санных символов – в версии 2.6 этого не происходит, поэтому в интерактив-  
ном сеансе вы не увидите эти числа. Этот пример записывает две строки текста   
в файл, добавляя в каждую из них символ конца строки \n� методы записи не   
добавляют символ конца строки, поэтому нам необходимо самостоятельно до-  
бавлять его в выводимые строки (в противном случае следующая операция за-  
писи просто продолжит текущую строку в файле).  
Если необходимо вывести содержимое файла, обеспечив правильную интер-  
претацию символов конца строки, его следует прочитать в строку целиком,   
с помощью метода read, и вывести:  
>>> open(‘myfile.txt’).read() # Прочитать файл целиком в строку  
‘hello text file\ngoodbye text file\n’  
   
>>> print(open(‘myfile.txt’).read()) # Более дружественная форма отображения  
hello text file  
goodbye text file  
А если необходимо просмотреть содержимое файла строку за строкой, лучшим   
выбором будет итератор файла:  
>>> for line in open(‘myfile’): # Используйте итераторы, а не методы чтения  
... print(line, end=’’)  
...  
hello text file  
goodbye text file  
В этом случае функцией open создается временный объект файла, содержимое   
которого автоматически будет читаться итератором и возвращаться по одной   
строке в каждой итерации цикла. Обычно такой способ компактнее, эффектив-  
нее использует память и может оказаться быстрее некоторых других вариан-  
тов (конечно, это зависит от множества параметров). Так как мы еще не каса-  
лись темы инструкций и итераторов, вам придется подождать до главы 14, где   
дается более полное описание этого примера.  
Текстовые и двоичные файлы в Python 3.0  
Строго говоря, в предыдущем примере выполняются операции с текстовыми   
файлами. В версиях Python 3.0 и 2.6 тип файла определяется вторым аргумен-  
Python 3.0 и 2.6 тип файла определяется вторым аргумен-  
 3.0 и 2.6 тип файла определяется вторым аргумен-  
том функции open – символ «b» в строке режима означает binary (двоичный).   
В языке Python всегда существовала поддержка текстовых и двоичных фай-  
Python всегда существовала поддержка текстовых и двоичных фай-  
 всегда существовала поддержка текстовых и двоичных фай-  
лов, но в Python 3.0 между этими двумя типами файлов была проведена более   
четкая грань:  
 •  
Содержимое текстовых  файлов представляется в виде обычных строк   
типа str, выполняется автоматическое кодирование/декодирование симво-  
лов Юникода и по умолчанию производится интерпретация символов кон-  
ца строки.  
 •  
Содержимое двоичных файлов представляется в виде строк типа bytes, и оно   
передается программе без каких-либо изменений.  
В Python 2.6, напротив, текстовые файлы могли содержать текст из 8-битных   
символов или двоичные данные, а для работы с текстом из символов Юникода   
использовался специальный строковый тип и интерфейс доступа к файлам

294   
Глава 9. Кортежи, файлы и все остальное   
(строки unicode и функция codecs.open). Изменения в Python 3.0 определялись   
тем фактом, что обычный текст и текст в Юникоде были объединены в еди-  
ный строковый тип, что имеет определенный смысл, если учесть, что любой   
текст может быть представлен в Юникоде, включая ASCII и другие 8-битные   
кодировки.  
Большинству программистов приходится иметь дело только с текстом ASCII,   
поэтому они могут пользоваться базовым интерфейсом доступа к текстовым   
файлам, как показано в предыдущем примере, и обычными строками. С тех-  
нической точки зрения, все строки в версии 3.0 являются строками Юникода,   
но для тех, кто использует только символы ASCII, это обстоятельство обычно   
остается незамеченным. В действительности операции над строками в верси-  
ях 3.0 и 2.6 выполняются одинаково, если область применения сценария огра-  
ничивается такими простыми формами текста.  
Если у вас имеется необходимость интернационализировать приложение или   
обрабатывать двоичные данные, отличия в версии 3.0 окажут большое влия-  
ние на программный код (обычно в лучшую сторону). Вообще говоря, для рабо-  
ты с двоичными файлами следует использовать строки bytes, а обычные строки   
str – для работы с текстовыми файлами. Кроме того, так как текстовые файлы   
реализуют автоматическое преобразование символов Юникода, вы не сможете   
открыть файл с двоичными данными в текстовом режиме – преобразование его   
содержимого в символы Юникода, скорее всего, завершится с ошибкой.  
Рассмотрим пример. Когда выполняется операция чтения двоичных данных   
из файла, она возвращает объект типа bytes – последовательность коротких   
целых чисел, представляющих абсолютные значения байтов (которые могут   
соответствовать символам, а могут и не соответствовать), который во многих   
отношениях очень близко напоминает обычную строку:  
>>> data = open(‘data.bin’, ‘rb’).read() # Открыть двоичный файл для чтения  
>>> data # Строка bytes хранит двоичные данные  
b’\x00\x00\x00\x07spam\x00\x08’  
>>> data[4:8] # Ведет себя как строка  
b’spam’  
>>> data[4:8][0] # Но в действительности хранит 8-битные целые числа  
115  
>>> bin(data[4:8][0]) # Функция bin() в Python 3.0   
‘0b1110011’  
Кроме того, двоичные файлы не выполняют преобразование символов конца   
строки – текстовые файлы по умолчанию отображают все разновидности сим-  
волов конца строки в и из символ \n в процессе записи и чтения, и производят   
преобразование символов Юникода в соответствии с указанной кодировкой.   
Так как операции с символами Юникода и с двоичными данными представля-  
ют особый интерес для многих программистов, мы отложим полное их обсуж-  
дение до главы 36. А пока перейдем к некоторым более насущным примерам   
использования файлов.  
Сохранение и интерпретация объектов Python в файлах  
Следующий пример записывает различные объекты в текстовый файл. Обра-  
тите внимание на использование средств преобразования объектов в строки.   
Напомню, что данные всегда записываются в файл в виде строк, а методы за-

Файлы   
295  
писи не выполняют автоматического форматирования строк (для экономии   
пространства я опустил вывод значений, возвращаемых методом write):  
>>> X, Y, Z = 43, 44, 45 # Объекты языка Python должны  
>>> S = ‘Spam’ # записываться в файл только в виде строк  
>>> D = {‘a’: 1, ‘b’: 2}  
>>> L = [1, 2, 3]  
>>>  
>>> F = open(‘datafile.txt’, ‘w’) # Создает файл для записи  
>>> F.write(S + ‘\n’) # Строки завершаются символом \n  
>>> F.write(‘%s,%s,%s\n’ % (X, Y, Z)) # Преобразует числа в строки  
>>> F.write(str(L) + ‘$’ + str(D) + ‘\n’) # Преобразует и разделяет символом $  
>>> F.close()  
Создав файл, мы можем исследовать его содержимое, открыв файл и прочитав   
данные в строку (одной операцией). Обратите внимание, что функция автома-  
тического вывода в интерактивной оболочке дает точное побайтовое представ-  
ление содержимого, а инструкция print интерпретирует встроенные символы   
конца строки, чтобы обеспечить более удобочитаемое отображение:  
>>> chars = open(‘datafile.txt’).read() # Отображение строки  
>>> chars # в неформатированном виде  
“Spam\n43,44,45\n[1, 2, 3]${‘a’: 1, ‘b’: 2}\n”  
>>> print(chars) # Удобочитаемое представление   
Spam  
43,44,45  
[1, 2, 3]${‘a’: 1, ‘b’: 2}  
Теперь нам необходимо выполнить обратные преобразования, чтобы получить   
из строк в текстовом файле действительные объекты языка Python. Интер-  
претатор Python никогда автоматически не выполняет преобразование строк   
в числа или в объекты других типов, поэтому нам необходимо выполнить соот-  
ветствующие преобразования, чтобы можно было использовать операции над   
этими объектами, такие как индексирование, сложение и так далее:  
>>> F = open(‘datafile.txt’) # Открыть файл снова  
>>> line = F.readline() # Прочитать одну строку  
>>> line  
‘Spam\n’  
>>> line.rstrip() # Удалить символ конца строки  
‘Spam’  
К этой строке мы применили метод rstrip, чтобы удалить завершающий сим-  
вол конца строки. Тот же эффект можно было бы получить с помощью извлече-  
ния среза line[:-1], но такой подход можно использовать, только если мы увере-  
ны, что все строки завершаются символом \n (последняя строка в файле иногда   
может не содержать этого символа).   
Пока что мы прочитали ту часть файла, которая содержит строку. Теперь про-  
читаем следующий блок, в котором содержатся числа, и выполним разбор это-  
го блока (то есть извлечем объекты):  
>>> line = F.readline() # Следующая строка из файла  
>>> line # Это - строка  
‘43,44,45\n’  
>>> parts = line.split(‘,’) # Разбить на подстроки по запятым

296   
Глава 9. Кортежи, файлы и все остальное   
>>> parts  
[‘43’, ‘44’, ‘45\n’]  
Здесь был использован метод split, чтобы разбить строку на части по запятым,   
которые играют роль символов-разделителей, – в результате мы получили спи-  
сок строк, каждая из которых содержит отдельное число. Теперь нам необхо-  
димо преобразовать эти строки в целые числа, чтобы можно было выполнять   
математические операции над ними:  
>>> int(parts[1]) # Преобразовать строку в целое число  
44  
>>> numbers = [int(P) for P in parts] # Преобразовать весь список   
>>> numbers  
[43, 44, 45]  
Как мы уже знаем, функция int преобразует строку цифр в объект целого чис-  
ла, а генератор списков, представленный в главе 4, позволяет применить функ-  
цию ко всем элементам списка в одной инструкции (подробнее о генераторах   
списков читайте далее в этой книге). Обратите внимание: для удаления завер-  
шающего символа \n в конце последней подстроки не был использован метод   
rstrip, потому что int и некоторые другие функции преобразования просто иг-  
норируют символы-разделители, окружающие цифры.  
Наконец, чтобы преобразовать список и словарь в третьей строке файла, мож-  
но воспользоваться встроенной функцией eval, которая интерпретирует строку   
как программный код на языке Python (формально – строку, содержащую вы-  
ражение на языке Python):  
>>> line = F.readline()  
>>> line  
“[1, 2, 3]${‘a’: 1, ‘b’: 2}\n”  
>>> parts = line.split(‘$’) # Разбить на строки по символу $  
>>> parts  
[‘[1, 2, 3]’, “{‘a’: 1, ‘b’: 2}\n”]  
>>> eval(parts[0]) # Преобразовать строку в объект  
[1, 2, 3]  
>>> objects = [eval(P) for P in parts] # То же самое для всех строк в списке  
>>> objects  
[[1, 2, 3], {‘a’: 1, ‘b’: 2}]  
Поскольку теперь все данные представляют собой список обычных объектов,   
а не строк, мы сможем применять к ним операции списков и словарей.  
Сохранение объектов Python с помощью модуля pickle  
Функция eval, использованная в предыдущем примере для преобразования   
строк в объекты, представляет собой мощный инструмент. И иногда даже слиш-  
ком мощный. Функция eval без лишних вопросов выполнит любое выражение   
на языке Python, даже если в результате будут удалены все файлы в компьюте-  
ре, если передать в выражение соответствующие права доступа! Если вам дей-  
ствительно необходимо извлекать объекты Python из файлов, но вы не можете   
доверять источнику этих файлов, идеальным решением будет использование   
модуля pickle, входящего в состав стандартной библиотеки Python.  
Модуль pickle позволяет сохранять в файлах практически любые объекты   
Python без необходимости с нашей стороны выполнять какие-либо преобразо-

Файлы   
297  
вания. Он напоминает суперуниверсальную утилиту форматирования и преоб-  
разования данных. Чтобы сохранить словарь в файле, например, мы передаем   
его непосредственно в функцию модуля pickle:  
>>> D = {‘a’: 1, ‘b’: 2}  
>>> F = open(‘datafile.pkl’, ‘wb’)  
>>> import pickle  
>>> pickle.dump(D, F) # Модуль pickle запишет в файл любой объект   
>>> F.close()  
Чтобы потом прочитать словарь обратно, можно просто еще раз воспользовать-  
ся возможностями модуля pickle:  
>>> F = open(‘datafile.pkl’ 'rb')  
>>> E = pickle.load(F) # Загружает любые объекты из файла  
>>> E  
{‘a’: 1, ‘b’: 2}  
Нам удалось получить назад точно такой же объект словаря без необходимости   
вручную выполнять какие-либо преобразования. Модуль pickle выполняет то,   
что называется сериализацией объектов, – преобразование объектов в строку   
байтов и обратно, не требуя от нас почти никаких действий. В действительно-  
сти, внутренняя реализация модуля pickle выполнила преобразование нашего   
словаря в строку, при этом незаметно для нас (и может выполнить еще более   
замысловатые преобразования при использовании модуля в других режимах):  
>>> open(‘datafile.pkl’, ‘rb’).read() # Формат может измениться!  
b’\x80\x03}q\x00(X\x01\x00\x00\x00aq\x01K\x01X\x01\x00\x00\x00bq\x02K\x02u.’  
Поскольку модуль pickle умеет реконструировать объекты из строкового пред-  
ставления, нам не требуется самим возиться с этим. Дополнительную инфор-  
мацию о модуле pickle вы найдете в руководстве по стандартной библиотеке   
языка Python или попробовав импортировать модуль в интерактивном сеансе   
и передав его имя функции help. Когда будете заниматься исследованием этого   
модуля, обратите также внимание на модуль shelve – инструмент, который ис-  
пользует модуль pickle для сохранения объектов Python в файлах с доступом по   
ключу, описание которых далеко выходит за рамки этой книги (впрочем, при-  
мер использования модуля shelve вы найдете в главе 27� кроме того, дополни-  
тельные примеры использования модуля pickle приводятся в главах 30 и 36).  
Обратите внимание, что в примере выше я открыл файл, где хра-  
нится сериализованный объект, в двоичном режиме. В Python 3.0   
такие файлы всегда следует открывать именно в двоичном ре-  
жиме, потому что модуль pickle создает и использует объекты   
типа bytes, а эти объекты предполагают, что файл открыт в дво-  
ичном режиме (в версии 3.0 при работе с текстовыми файлами   
используются строки типа str). В более ранних версиях Python   
допускается использовать текстовые файлы, когда выбирается   
протокол 0 (используется по умолчанию и создает текстовые   
файлы в кодировке ASCII), при условии непротиворечивого ис-  
пользования текстового режима. Протоколы с более высокими   
номерами допускают возможность работы только с двоичными   
файлами. В Python 3.0 по умолчанию используется протокол 3   
(двоичный), но в этой версии интерпретатора модуль pickle соз-

298   
Глава 9. Кортежи, файлы и все остальное   
дает строки типа bytes даже при использовании протокола 0. До-  
полнительные подробности по этой теме вы найдете в главе 36,   
в справочном руководстве по библиотеке языка Python или   
в других источниках.  
В версии Python 2.6 дополнительно имеется модуль cPickle –   
оптимизированная версия модуля pickle, который можно им-  
портировать для повышения скорости. В Python 3.0 этот модуль   
переименован в \_pickle и автоматически используется модулем   
pickle – сценарии просто импортируют модуль pickle и позволя-  
ют интерпретатору самому оптимизировать свою работу.  
Сохранение и интерпретация   
упакованных двоичных данных в файлах  
Прежде чем двинуться дальше, необходимо рассмотреть еще один аспект ра-  
боты с файлами: в некоторых приложениях приходится иметь дело с упако-  
ванными двоичными данными, которые создаются, например, программами   
на языке C. Стандартная библиотека языка Python включает инструмент, спо-  
собный помочь в этом, – модуль struct, который позволяет сохранять и восста-  
навливать упакованные двоичные данные. В некотором смысле, это совершен-  
но другой инструмент преобразования данных, интерпретирующий строки   
в файлах как двоичные данные.  
Например, чтобы создать файл с упакованными двоичными данными, открой-  
те его в режиме ‘wb’ (write binary – запись двоичных данных) и передайте мо-  
write binary – запись двоичных данных) и передайте мо-  
 binary – запись двоичных данных) и передайте мо-  
binary – запись двоичных данных) и передайте мо-  
 – запись двоичных данных) и передайте мо-  
дулю struct строку формата и некоторый объект Python. В следующем примере   
используется строка формата, которая определяет пакет данных, содержащий   
4-байтовое целое число, 4-символьную строку и 2-байтовое целое число, при-  
чем все представлены в формате big-endian – в порядке следования байтов «от   
старшего к младшему» (существуют также спецификаторы форматов, которые   
поддерживают наличие символов дополнения слева, числа с плавающей точ-  
кой и многие другие):  
>>> F = open(‘data.bin’, ‘wb’) # Открыть файл для записи в двоичном режиме  
>>> import struct  
>>> data = struct.pack(‘>i4sh’, 7, ‘spam’, 8) # Создать пакет двоичных данных  
>>> data  
b’\x00\x00\x00\x07spam\x00\x08’  
>>> F.write(data) # Записать строку байтов  
>>> F.close()  
Интерпретатор Python создаст строку bytes двоичных данных, которую затем   
мы запишем в файл обычным способом (строка состоит из экранированных по-  
следовательностей, представляющих шестнадцатеричные значения). Чтобы   
преобразовать эти значения в обычные объекты языка Python, достаточно про-  
Python, достаточно про-  
, достаточно про-  
сто прочитать строку обратно и распаковать ее с использованием той же строки   
формата. Следующий фрагмент извлекает значения, преобразуя их в обычные   
объекты (целые числа и строка):  
>>> F = open(‘data.bin’, ‘rb’)  
>>> data = F.read() # Получить упакованные двоичные данные  
>>> data  
b’\x00\x00\x00\x07spam\x00\x08’

Файлы   
299  
>>> values = struct.unpack(‘>i4sh’, data) # Преобразовать в объекты  
>>> values  
(7, ‘spam’, 8)  
Файлы с двоичными данными относятся к категории низкоуровневых средств,   
которые мы не будем рассматривать подробно. За дополнительной информаци-  
ей обращайтесь к главе 36 и к руководству по библиотеке языка Python или   
импортируйте модуль struct в интерактивном сеансе и передайте имя struct   
функции help. Обратите также внимание, что режимы доступа к двоичным   
файлам ‘wb’ и ‘rb’ могут использоваться для обработки простейших двоичных   
файлов, таких как изображения или аудиофайлы, без необходимости выпол-  
нять распаковку их содержимого.  
Менеджеры контекста файлов  
Вам также необходимо будет прочитать обсуждение поддержки менеджеров   
контекста файлов в главе 33, впервые появившейся в версиях Python 3.0 и 2.6.   
Даже при том, что менеджеры контекста в основном применяются для обра-  
ботки исключений, тем не менее они позволяют обертывать программный код,   
выполняющий операции с файлами, дополнительным слоем логики, который   
гарантирует, что после выхода за пределы блока инструкций менеджера файл   
будет закрыт автоматически, и позволяет не полагаться на автоматическое за-  
крытие файлов механизмом сборки мусора:  
with open(r’C:\misc\data.txt’) as myfile: # Подробности в главе 33   
 for line in myfile:  
 ...операции над строкой line...  
Аналогичную функциональность предоставляет конструкция try/finally,   
с которой мы познакомимся в главе 33, но за счет избыточного программного   
кода – три дополнительных строки, если быть более точным (впрочем, мы мо-  
жем не использовать ни один из вариантов и позволить интерпретатору самому   
закрывать файлы):  
myfile = open(r’C:\misc\data.txt’)  
try:  
 for line in myfile:  
 ...операции над строкой line...  
finally:  
 myfile.close()  
Поскольку для подробного описания обоих способов необходимо иметь допол-  
нительные знания, которыми мы еще не обладаем, мы обсудим в книге эти де-  
тали позже.  
Другие инструменты для работы с файлами  
Как показано в табл. 9.2, существуют более сложные инструменты для работы   
с файлами, более того, существуют и другие инструменты, которые отсутству-  
ют в таблице. Например, функция seek переустанавливает текущую позицию   
в файле (для следующей операции чтения или записи), функция flush прину-  
дительно выталкивает содержимое выходных буферов на диск (по умолчанию   
файлы всегда буферизуются) и так далее.

300   
Глава 9. Кортежи, файлы и все остальное   
Руководство по стандартной библиотеке и книги, упомянутые в предисловии,   
содержат полный перечень методов для работы с файлами. Чтобы кратко озна-  
комиться с ним, можно также воспользоваться функцией dir или help в ин-  
терактивном сеансе, передав ей объект открытого файла (в Python 3.0, но не   
в Python 2.6, где следует передать имя типа file). Дополнительные примеры   
работы с файлами вы найдете во врезке «Придется держать в уме: сканирова-  
ние файлов» в главе 13. В ней приводятся типичные примеры организации ци-  
клов для просмотра содержимого файлов и приводятся инструкции, которые   
мы еще не рассматривали.  
Следует также отметить, что функция open и объекты файлов, которые она воз-  
вращает, являются в языке Python основным интерфейсом к внешним файлам,   
однако в арсенале Python существуют и другие инструменты, напоминающие   
файлы. Назовем некоторые из них:  
Стандартные потоки ввода-вывода  
Объекты уже открытых файлов в модуле sys, такие как sys.stdout (смотрите   
раздел «Инструкция print» в главе 11)  
Дескрипторы файлов в модуле os  
Целочисленные дескрипторы файлов, обеспечивающие поддержку низко-  
уровневых операций, таких как блокировка файлов  
Сокеты, каналы и очереди (FIFO)  
Объекты, по своим характеристикам напоминающие файлы, используемые   
для синхронизации процессов или организации взаимодействий по сети   
Файлы с доступом по ключу, известные как «хранилища» («shelves»)  
Используются для хранения объектов языка Python по ключу (глава 27)  
Потоки командной оболочки  
Такие инструменты, как os.popen и subprocess.Popen, которые поддержива-  
которые поддержива-  
 поддержива-  
поддержива-  
ют возможность запуска дочерних процессов и выполнения операций с их   
стандартными потоками ввода-вывода  
Среди сторонних открытых модулей можно отыскать еще больше инструмен-  
тов, напоминающих файлы, включая реализацию поддержки обмена данны-  
ми через последовательный порт в расширении PySerial и поддержки обмена   
данными с интерактивными программами в системе pexpect. Дополнительную   
информацию о подобных инструментах вы найдете в специализированной ли-  
тературе по языку Python и в Сети.  
Примечание,  касающееся  различий  между  версиями: В версии   
Python 2.5 и в более ранних имя open встроенной функции фак-  
тически было синонимом имени типа file, а файлы, с техниче-  
ской точки зрения, можно было открыть вызовом функции open   
или file (хотя предпочтительнее использовать функцию open).   
В Python 3.0 имя file больше недоступно, потому что оно дубли-  
рует имя open.  
Пользователи Python 2.6 могут также использовать имя file   
как имя типа объекта файла для создания дочерних классов при   
использовании объектно-ориентированного стиля программи-  
рования (описывается ниже в этой книге). В Python 3.0 файлы   
изменились радикально. Классы, которые могут использовать-  
 радикально. Классы, которые могут использовать-  
радикально. Классы, которые могут использовать-  
. Классы, которые могут использовать-  
Классы, которые могут использовать-

Пересмотренный перечень категорий типов   
301  
ся в качестве родительских, для реализации собственных типов   
файлов, находятся в модуле io. За дополнительной информа-  
цией обращайтесь к документации по этому модулю или к ис-  
ходным текстам доступных классов. Попробуйте также вызвать   
функцию type(F), передав ей объект открытого файла F.  
Пересмотренный перечень категорий типов  
Теперь, когда мы познакомились со всеми встроенными базовыми типами   
языка Python в действии, давайте завершим наше турне рассмотрением не-  
которых общих для них свойств. В табл. 9.3 приводится классификация всех   
рассмотренных типов по категориям, которые были введены ранее. Напомню   
некоторые положения:  
 •  
Над объектами одной категории могут выполняться одни и те же опера-  
ции, например над строками, списками и кортежами можно выполнять   
операции для последовательностей, такие как конкатенация и определе-  
ние длины.   
 •  
Непосредственное изменение допускают только изменяемые объекты (спи-  
ски, словари и множества) – вы не сможете непосредственно изменить чис-  
ла, строки и кортежи.   
 •  
Файлы экспортируют только методы, поэтому понятие изменяемости к ним   
по-настоящему неприменимо – хотя они могут изменяться при выполнении   
операции записи, но это не имеет никакого отношения к ограничениям ти-  
пов в языке Python.  
 •  
В категорию «Числа» в табл. 9.3 входят все числовые типы: целые (длинные   
целые, в Python 2.6), вещественные, комплексные, фиксированной точно-  
сти и рациональные.   
 •  
В категорию «Строки» в табл. 9.3 входят строки типа str, а также bytes (в   
версии 3.0) и unicode (в версии 2.6). Строковый тип bytearray в версии 3.0 от-  
в версии 3.0 от-  
 версии 3.0 от-  
версии 3.0 от-  
 3.0 от-  
от-  
носится к изменяемым типам.  
 •  
Множества напоминают словари, в которых ключи не имеют значений. Эле-  
менты множеств не отображаются на значения и неупорядочены, поэтому   
множества нельзя отнести ни к отображениям, ни к последовательностям.   
Тип frozenset – это неизменяемая версия типа set.  
 •  
Вдобавок к вышесказанному, в версиях Python 2.6 и 3.0 все типы, перечис-  
Python 2.6 и 3.0 все типы, перечис-  
 2.6 и 3.0 все типы, перечис-  
ленные в табл. 9.3, обладают методами, которые обычно реализуют опера-  
ции, характерные для определенного типа.  
Таблица 9.3. Классификация объектов  
Тип объектов  
Категория  
Изменяемый?  
Числа (все)  
Числа  
Нет  
Строки  
Последовательности  
Нет  
Списки  
Последовательности  
Да  
Словари  
Отображения  
Да  
Кортежи  
Последовательности  
Нет

302   
Глава 9. Кортежи, файлы и все остальное   
Тип объектов  
Категория  
Изменяемый?  
Файлы  
Расширения  
Понятие неприменимо  
Множества  
Множества  
Да  
Фиксированные множества   
(frozenset)  
Множества  
Нет  
bytearray (3.0)  
Последовательности  
Да  
Придется держать в уме: перегрузка операторов  
В шестой части книги мы увидим, что объекты, реализованные с по-  
мощью классов, могут свободно попадать в любую из этих категорий.   
Например, если нам потребуется реализовать новый тип объектов по-  
следовательностей, который был бы совместим со встроенными типами   
последовательностей, мы могли бы описать класс, который перегружает   
операторы индексирования и конкатенации:  
class MySequence:  
 def \_\_getitem\_\_(self, index):  
 # Вызывается при выполнении операции self[index]   
 def \_\_add\_\_(self, other):  
 # Вызывается при выполнении операции self + other  
и так далее. Точно так же можно было бы создать новый изменяемый   
или неизменяемый объект, выборочно реализовав методы, вызываемые   
для выполнения операций непосредственного изменения (например, для   
выполнении операций присваивания self[index] = value вызывается ме-  
тод \_\_setitem\_\_). Существует также возможность выполнять реализацию   
новых объектов на других языках программирования, таких как C, в ви-  
де расширений. Чтобы определить выбор между множествами операций   
над числами, последовательностями и отображениями, необходимо бу-  
дет подставить указатели на функции C в ячейки структуры.  
Гибкость объектов  
В этой части книги были представлены несколько составных типов объектов   
(коллекции с компонентами). В общем случае:  
 •  
Списки, словари и кортежи могут хранить объекты любых типов.  
 •  
Списки, словари и кортежи допускают произвольную вложенность.  
 •  
Списки и словари могут динамически увеличиваться и уменьшаться.  
Поскольку составные типы объектов в языке Python поддерживают любые   
структуры данных, они прекрасно подходят для представления в программах   
данных со сложной структурой. Например, значениями элементов словарей   
могут быть списки, которые могут содержать кортежи, которые в свою очередь   
могут содержать словари, и так далее. Вложенность может быть произвольной   
Таблица 9.3 (продолжение)

Ссылки и копии   
303  
глубины, какая только потребуется, чтобы смоделировать структуру обраба-  
тываемых данных.  
Рассмотрим пример с вложенными компонентами. В следующем примере опре-  
деляется три вложенных составных объекта последовательностей, как показа-  
но на рис. 9.1. Для обращения к отдельным элементам допускается использо-  
вать столько индексов, сколько потребуется. В языке Python разыменование   
индексов производится слева направо и на каждом шаге извлекается объект   
на более глубоком уровне. Структура данных на рис. 9.1 может показаться из-  
лишне сложной, но она иллюстрирует синтаксис, который используется для   
доступа к данным:  
>>> L = [‘abc’, [(1, 2), ([3], 4)], 5]  
>>> L[1]  
[(1, 2), ([3], 4)]  
>>> L[1][1]  
([3], 4)  
>>> L[1][1][0]  
[3]  
>>> L[1][1][0][0]  
3  
abc  
5  
4  
3  
2  
1  
[0]  
[1]  
[2]  
[0]  
[1]  
[0]  
[1]  
[0]  
[0]  
[1]  
Рис. 9.1. Дерево вложенных объектов со смещениями их компонентов,   
созданное с помощью литерального выражения [‘abc’, [(1, 2), ([3], 4)], 5].    
Синтаксически вложенные объекты внутри реализованы в виде ссылок    
(то есть в виде указателей) на отдельные участки памяти  
Ссылки и копии  
В главе 6 говорилось, что при выполнении операции присваивания всегда со-  
храняется ссылка на объект, а не копия самого объекта. В большинстве слу-  
чаев именно это нам и требуется. Однако в процессе выполнения операций   
присваивания может быть создано несколько ссылок на один и тот же объект,   
поэтому очень важно понимать, что изменения, произведенные в изменяемом

304   
Глава 9. Кортежи, файлы и все остальное   
объекте с помощью одной ссылки, отразятся и на других ссылках, указываю-  
щих на этот же объект. Если такой порядок вещей является нежелательным,   
необходимо явно сообщить интерпретатору, чтобы он создал копию объекта.  
Мы изучили это явление в главе 6, но ситуация может обостриться, когда   
в игру вступят крупные объекты. Например, в следующем примере создается   
список, который присваивается переменной X, затем создается другой список,   
включающий ссылку на список X, который присваивается переменной L. Далее   
создается словарь D, который содержит еще одну ссылку на список X:  
>>> X = [1, 2, 3]  
>>> L = [‘a’, X, ‘b’] # Встроенная ссылка на объект X  
>>> D = {‘x’:X, ‘y’:2}  
В этот момент в программе появляется три ссылки на список, который был соз-  
дан первым: из переменной X, из элемента списка, соответствующего перемен-  
ной L, и из элемента словаря, соответствующего переменной D. Эта ситуация   
изображена на рис. 9.2.  
Имена  
Объекты  
1  
?  
3  
L  
X  
D  
Рис. 9.2. Разделяемые ссылки на объект: поскольку переменная X, элемент   
списка L и элемент словаря D ссылаются на один и тот же список, то    
изменения, произведенные с помощью X, отразятся также на L и D  
Так как списки относятся к категории изменяемых объектов, изменения в объ-  
екте списка, произведенные с помощью любой из трех ссылок, также отразятся   
на других двух ссылках:  
>>> X[1] = ‘surprise’ # Изменяет все три ссылки!  
>>> L  
[‘a’, [1, ‘surprise’, 3], ‘b’]  
>>> D  
{‘x’: [1, ‘surprise’, 3], ‘y’: 2}  
Ссылки – это более высокоуровневый аналог указателей в других языках про-  
граммирования. И хотя вы не можете извлечь непосредственное значение самой   
ссылки, тем не менее одну и ту же ссылку можно сохранить более чем в одном   
месте (в переменных, в списках и так далее). Это полезная особенность – вы   
можете передавать крупные объекты между компонентами программы без вы-  
полнения дорогостоящей операции копирования. Однако, когда действительно   
необходимо создать копию объекта, вы можете запросить их:

Ссылки и копии   
305  
 •  
Выражение извлечения среза с пустыми пределами (L[:]) создает копию по-  
следовательности.  
 •  
Метод словарей и множеств copy создает копию словаря (D.copy()).  
 •  
Некоторые встроенные функции, такие как list, создают копию списка   
(list(L)).  
 •  
Модуль copy, входящий в состав стандартной библиотеки, создает полные   
копии объектов.  
Например, предположим, что у нас имеются список и словарь, и для нас неже-  
лательно изменение их значений посредством других переменных:  
>>> L = [1,2,3]  
>>> D = {‘a’:1, ‘b’:2}  
Чтобы предотвратить такую возможность, достаточно связать с другими пере-  
менными копии объектов вместо того, чтобы связать с ними сами объекты:  
>>> A = L[:] # Вместо A = L (или list(L))  
>>> B = D.copy() # Вместо B = D (то же относится и к множествам)  
В этом случае изменения, произведенные с помощью других переменных, по-  
влияют на копии, а не на оригиналы:  
>>> A[1] = ‘Ni’  
>>> B[‘c’] = ‘spam’  
>>>  
>>> L, D  
([1, 2, 3], {‘a’: 1, ‘b’: 2})  
>>> A, B  
([1, ‘Ni’, 3], {‘a’: 1, ‘c’: ‘spam’, ‘b’: 2})  
Если вернуться к первоначальному примеру, то избежать побочных эффектов,   
связанных со ссылками, можно, использовав выражение извлечения среза из   
оригинального списка:  
>>> X = [1, 2, 3]  
>>> L = [‘a’, X[:], ‘b’] # Встроенные копии объекта X  
>>> D = {‘x’:X[:], ‘y’:2}  
Это меняет картину, представленную на рис. 9.2 – L и D теперь ссылаются на   
другие списки, отличные от X. Теперь изменения, выполненные с помощью пе-  
ременной X, отразятся только на самой переменной X, но не на L и D� точно так   
же изменения в L или D никак не будут воздействовать на X.  
Одно замечание по поводу копий: выражение извлечения среза с пустыми зна-  
чениями пределов и метод словаря copy создают поверхностные копии – то   
есть они не копируют вложенные структуры данных, даже если таковые при-  
сутствуют. Если необходима полная копия структуры произвольной глубины   
вложенности, следует использовать стандартный модуль copy: добавьте ин-  
струкцию import copy и вставьте выражение X = copy.deepcopy(Y), которое создаст   
полную копию объекта Y со сколь угодно большой глубиной вложенности. Эта   
функция выполняет рекурсивный обход объектов и копирует все составные   
части. Однако такой подход используется намного реже (потому что для этого   
приходится писать дополнительный программный код). Обычно ссылки – это   
именно то, что нам требуется, в противном случае чаще всего применяются та-  
кие способы копирования, как операция извлечения среза и метод copy.

306   
Глава 9. Кортежи, файлы и все остальное   
Сравнивание, равенство и истина  
Любые объекты языка Python поддерживают операции сравнения: проверка   
на равенство, относительная величина и так далее. Операции сравнения в язы-  
ке Python всегда проверяют все части составных объектов, пока результат не   
станет определенным. В действительности, при сравнивании вложенных объ-  
ектов интерпретатор Python всегда автоматически выполняет обход структу-  
Python всегда автоматически выполняет обход структу-  
 всегда автоматически выполняет обход структу-  
ры данных, чтобы применить операции сравнения рекурсивно, слева направо   
и настолько глубоко, насколько это необходимо. Первое найденное различие   
определяет результат сравнения.  
Например, при сравнении списков автоматически сравниваются все их компо-  
ненты:  
>>> L1 = [1, (‘a’, 3)] # Разные объекты с одинаковыми значениями  
>>> L2 = [1, (‘a’, 3)]  
>>> L1 == L2, L1 is L2 # Равны? Это один и тот же объект?  
(True, False)  
В данном случае L1 и L2 ссылаются на совершенно разные списки, но с одина-  
ковым содержимым. Из-за природы ссылок в языке Python (рассматривались   
в главе 6) существует два способа проверки на равенство:  
 •  
Оператор == проверяет равенство значений. Интерпретатор выполняет   
проверку на равенство, рекурсивно сравнивая все вложенные объекты.  
 •  
Оператор is проверяет идентичность объектов. Интерпретатор проверяет,   
являются ли сравниваемые объекты одним и тем же объектом (то есть рас-  
положены ли они по одному и тому же адресу в памяти).  
В предыдущем примере L1 и L2 прошли проверку == на равенство (они равны,   
потому что все их компоненты равны), но не прошли проверку is на идентич-  
ность (они ссылаются на разные объекты и соответственно, на разные области   
памяти). Однако обратите внимание, что происходит при сравнивании двух   
коротких строк:  
>>> S1 = ‘spam’  
>>> S2 = ‘spam’  
>>> S1 == S2, S1 is S2  
(True, True)  
Здесь у нас так же имеется два различных объекта с одинаковыми значениями:   
оператор == должен вернуть истину, а оператор is – ложь. Но так как интерпре-  
татор с целью оптимизации кэширует и повторно использует короткие строки,   
в действительности в обе переменные, S1 и S2, записывается ссылка на одну ту   
же строку ‘spam’ в памяти, поэтому оператор is проверки идентичности возвра-  
щает истину. Чтобы получить нормальное поведение, потребуется использо-  
вать более длинные строки:  
>>> S1 = ‘a longer string’  
>>> S2 = ‘a longer string’  
>>> S1 == S2, S1 is S2  
(True, False)  
Разумеется, так как строки являются неизменяемыми, использование меха-  
низма кэширования объектов не оказывает отрицательного влияния на про-  
граммный код – строки нельзя изменить непосредственно, независимо от того,

Сравнивание, равенство и истина   
307  
сколько переменных ссылаются на них. Если результат проверки на идентич-  
ность кажется вам обескураживающим, вернитесь к главе 6, чтобы освежить   
в памяти концепцию ссылок на объекты.  
Как правило, оператор == – это именно то, что требуется в большинстве случаев   
проверки равенства� оператор is предназначен для особых случаев. Мы рассмо-  
трим случаи использования этих операторов далее в книге.  
Операторы отношений к вложенным структурам также применяются рекур-  
сивно:  
>>> L1 = [1, (‘a’, 3)]  
>>> L2 = [1, (‘a’, 2)]  
>>> L1 < L2, L1 == L2, L1 > L2 # Меньше, равно, больше: кортеж результатов  
(False, False, True)  
Здесь L1 больше, чем L2, потому что вложенное число 3 больше, чем 2. Резуль-  
тат последней строки в этом примере в действительности представляет собой   
кортеж из трех объектов – результатов трех выражений (пример кортежа без   
объемлющих круглых скобок).  
В общем случае Python сравнивает типы следующим образом:  
 •  
Числа сравниваются по величине.  
 •  
Строки сравниваются лексикографически, символ за символом (“abc” <   
“ac”).  
 •  
При сравнении списков и кортежей сравниваются все компоненты слева на-  
право.  
 •  
Словари сравниваются как отсортированные списки (ключ, значение). Сло-  
вари в Python 3.0 не поддерживают операторы отношений, но они поддер-  
Python 3.0 не поддерживают операторы отношений, но они поддер-  
 3.0 не поддерживают операторы отношений, но они поддер-  
живаются в версии 2.6 и ниже, при этом они сравниваются как отсортиро-  
ванные списки (ключ, значение).  
 •  
Попытка сравнить значения несопоставимых нечисловых типов (например,   
1 < ‘spam’) в Python 3.0 вызывает ошибку. Зато такое сравнивание возмож-  
Python 3.0 вызывает ошибку. Зато такое сравнивание возмож-  
 3.0 вызывает ошибку. Зато такое сравнивание возмож-  
но в Python 2.6, однако при этом используется фиксированное и достаточно   
произвольно выбранное правило. То же относится и к операции сортиров-  
ки, которая реализована на основе операции сравнения: коллекции значе-  
ний нечисловых и несопоставимых типов в версии 3.0 отсортированы быть   
не могут.  
Вообще сравнивание структурированных объектов выполняется, как если бы   
сравнивались образующие их литералы, слева направо, по одному компоненту   
за раз. В последующих главах мы увидим другие типы объектов, которые мо-  
гут изменять способ сравнения.  
Сравнивание словарей в Python 3.0  
Предпоследний пункт в списке из предыдущего раздела заслуживает особого   
пояснения. В Python 2.6 и в более ранних версиях словари поддерживали воз-  
Python 2.6 и в более ранних версиях словари поддерживали воз-  
 2.6 и в более ранних версиях словари поддерживали воз-  
можность сравнения с помощью операторов отношений, как если бы сравнива-  
лись сортированные списки пар ключ/значение:  
C:\misc> c:\python26\python  
>>> D1 = {‘a’:1, ‘b’:2}  
>>> D2 = {‘a’:1, ‘b’:3}

308   
Глава 9. Кортежи, файлы и все остальное   
>>> D1 == D2  
False  
>>> D1 < D2  
True  
В Python 3.0 поддержка операторов отношений в словарях была ликвидирова-  
Python 3.0 поддержка операторов отношений в словарях была ликвидирова-  
 3.0 поддержка операторов отношений в словарях была ликвидирова-  
на, потому что они оказываются слишком затратными, когда просто требует-  
ся узнать, равны ли словари (проверка на равенство в версии 3.0 выполняется   
по оптимизированному алгоритму, который по факту не сравнивает отсорти-  
рованные списки пар ключ/значение). Чтобы выяснить относительный поря-  
док словарей в версии 3.0, можно написать цикл и вручную сравнить значе-  
ния ключей или вручную создать отсортированные списки пар ключ/значение   
и сравнить их – для этого вполне достаточно будет задействовать метод слова-  
рей items и встроенную функцию sorted:  
C:\misc> c:\python30\python  
>>> D1 = {‘a’:1, ‘b’:2}  
>>> D2 = {‘a’:1, ‘b’:3}  
>>> D1 == D2  
False  
>>> D1 < D2  
TypeError: unorderable types: dict() < dict()  
   
>>> list(D1.items())  
[(‘a’, 1), (‘b’, 2)]  
>>> sorted(D1.items())  
[(‘a’, 1), (‘b’, 2)]  
>>> sorted(D1.items()) < sorted(D2.items())  
True  
>>> sorted(D1.items()) > sorted(D2.items())  
False  
На практике в большинстве программ, где требуется выполнять сравнивание   
словарей, реализуются собственные, более эффективные процедуры сравне-  
ния данных в словарях, чем данный способ или оригинальная реализация   
в Python 2.6.  
Смысл понятий «истина» и «ложь» в языке Python  
Обратите внимание, что три значения в кортеже, возвращаемом последним   
примером из предыдущего раздела, – это логические значения «истина» (true)   
и «ложь» (false). Они выводятся на экран как слова True и False, но теперь, когда   
мы приступаем к использованию логических проверок, мне следует более фор-  
мально описать смысл этих названий.  
В языке Python, как в большинстве языков программирования, «ложь» пред-  
Python, как в большинстве языков программирования, «ложь» пред-  
, как в большинстве языков программирования, «ложь» пред-  
ставлена целочисленным значением 0, а «истина» – целочисленным значени-  
ем 1. Кроме того, интерпретатор Python распознает любую пустую структуру   
данных как «ложь», а любую непустую структуру данных – как «истину».   
В более широком смысле понятия истины и лжи – это свойства, присущие   
всем объектам в Python, – каждый объект может быть либо «истиной», либо   
«ложью»:  
 •  
Числа, отличные от нуля, являются «истиной».  
 •  
Другие объекты являются «истиной», если они непустые.

Сравнивание, равенство и истина   
309  
В табл. 9.4 приводятся примеры истинных и ложных значений объектов в язы-  
ке Python.  
Таблица 9.4. Примеры значений истинности объектов  
Объект  
Значение  
“spam”  
True  
“”  
False  
[]  
False  
{}  
False  
1  
True  
0.0  
False  
None  
False  
Так как объекты могут быть «истинными» или «ложными», в программах на   
языке Python часто можно увидеть такие условные инструкции, как if X:, ко-  
торая, если предположить, что X является строкой, равноценна инструкции if   
X != ‘’:. Говоря другими словами, можно проверить истинность объекта вместо   
того, чтобы сравнивать его с пустым объектом. (Подробнее об инструкции if   
рассказывается в третьей части III.)  
Объект None  
В языке Python имеется также специальный объект с именем None (последняя   
строка в табл. 9.4), который всегда расценивается как «ложь». Объект None был   
представлен в главе 4 – это единственный специальный тип данных в языке   
Python, который играет роль пустого заполнителя и во многом похож на ука-  
, который играет роль пустого заполнителя и во многом похож на ука-  
затель NULL в языке C.  
Например, вспомните, что списки не допускают присваивание значений отсут-  
ствующим элементам (списки не вырастают по волшебству, когда вы выпол-  
няете присваивание некоторого значения несуществующему элементу). Чтобы   
можно было выполнять присваивание любому из 100 элементов списка, обыч-  
но создается список из 100 элементов, который заполняется объектами None:  
>>> L = [None] \* 100  
>>>  
>>> L  
[None, None, None, None, None, None, None, ... ]  
В этом примере мы никак не ограничиваем размер списка (его по-прежнему   
позднее можно будет увеличить или уменьшить), мы просто создаем список   
определенного размера, чтобы обеспечить возможность присваивания по ин-  
дексам. Точно так же можно было бы инициализировать список нулями, но   
все-таки предпочтительнее использовать None, если заранее не известно, какие   
данные будет содержать этот список.  
Имейте в виду, что значение None не означает «неопределенный». То есть None –   
это не «ничто» (несмотря на свое имя!)� это настоящий объект, занимающий   
определенную область памяти, с зарезервированным именем. Другие примеры

310   
Глава 9. Кортежи, файлы и все остальное   
использования этого специального объекта вы найдете ниже в книге. Кроме   
того, этот объект является значением, возвращаемым функциями по умолча-  
нию, как будет показано в четвертой части.  
Тип bool  
Логический тип bool в языке Python (представленный в главе 5) просто уси-  
Python (представленный в главе 5) просто уси-  
 (представленный в главе 5) просто уси-  
ливает понятия «истина» и «ложь» в языке Python. Как мы узнали в главе 5,   
предопределенные имена True и False являются всего лишь разновидностями   
целых чисел 1 и 0, как если бы этим двум зарезервированным именам пред-  
варительно присваивались значения 1 и 0. Этот тип реализован так, что он   
действительно является лишь незначительным расширением к понятиям «ис-  
тина» и «ложь», уже описанным, и предназначенным для того, чтобы сделать   
значения истинности более явными:  
 •  
При явном использовании слов True и False в операциях проверки на истин-  
ность они интерпретируются как «истина» и «ложь», которые в действи-  
тельности являются специализированными версиями целых чисел 1 и 0.   
 •  
Результаты операций проверок также выводятся как слова True и False, а не   
как значения 1 и 0.  
Вам не обязательно использовать только логические типы в логических ин-  
струкциях, таких как if, – любые объекты по своей природе могут быть истин-  
ными или ложными, и все логические концепции, упоминаемые в этой главе,   
работают, как описано, и с другими типами. Кроме того, в языке Python име-  
Python име-  
 име-  
ется встроенная функция bool, которая может использоваться для проверки   
логического значения объекта (например, для проверки истинности объекта,   
чтобы убедиться, что объект не является пустым или не равен нулю):  
>>> bool(1)  
True  
>>> bool(‘spam’)  
True  
>>> bool({})  
False  
Однако на практике вам редко придется вручную пользоваться переменными   
типа bool, которые воспроизводятся логическими проверками, потому что ло-  
гические результаты автоматически используются инструкциями if и други-  
ми средствами выбора. Мы займемся исследованием логического типа, когда   
будем рассматривать логические инструкции в главе 12.  
Иерархии типов данных в языке Python  
На рис. 9.3 приводятся все встроенные типы объектов, доступные в языке Py-  
Py-  
thon, и отображены взаимоотношения между ними. Наиболее значимые из   
них мы уже рассмотрели� большая часть других видов объектов на рис. 9.3 со-  
ответствует элементам программ (таким как функции и модули) или отражает   
внутреннее устройство интерпретатора (такие как кадры стека и скомпилиро-  
ванный программный код).  
Главное, что следует учитывать, – в языке Python все элементы являются объ-  
ектами и могут быть использованы в ваших программах. Например, можно   
передать класс в функцию, присвоить его переменной, заполнить им список   
или словарь и так далее.

Иерархии типов данных в языке Python   
311  
Объекты типов  
Фактически даже сами типы в языке Python являются разновидностью объ-  
Python являются разновидностью объ-  
 являются разновидностью объ-  
ектов: объекты типов являются объектами типа type (попробуйте быстро про-  
изнести эту фразу три раза!). Говоря серьезно, вызов встроенной функции   
type(X) возвращает объект типа объекта X. Объекты типов могут использо-  
ваться для сравнения типов вручную в инструкции if. Однако по причинам,   
изложенным в главе 4, ручная проверка типа в языке Python обычно рассма-  
Python обычно рассма-  
 обычно рассма-  
тривается как нечто неправильное, что существенно ограничивает гибкость   
программного кода.  
Числа  
Целoе  
Вещественное  
Целые  
Комплексное  
Длинное целое  
(2.6)  
Рациональное  
Логическое  
C фиксированной  
точностью  
Коллекции  
Последовательности  
Отображения  
Неизменяемые  
Изменяемые  
Строка  
Список  
Словарь  
Юникод (2.6)  
Кортеж  
Связанный  
Несвязанный (2.6)  
Внутренние  
Тип  
Код  
Кадр  
Диагностика  
Массив   
байтов (3.0)  
Строки   
байтов (3.0)  
Множества  
Множество  
Фиксированное   
множество  
Вызываемые  
Функции  
Класс  
Метод  
Генератор  
Прочие  
Модуль  
Экземпляр  
Файл  
None  
Представление   
(3.0)  
Рис. 9.3. Основные встроенные типы в языке Python, организованные   
по категориям. Все в языке Python является объектом, даже сами типы –   
это объекты! Тип любого объекта – это объект типа «type»

312   
Глава 9. Кортежи, файлы и все остальное   
Одно замечание по поводу имен типов: в версии Python 2.2 у каждого базового   
типа появилось новое встроенное имя, добавленное для обеспечения поддерж-  
ки настройки типов через объектно-ориентированный механизм наследования   
классов: dict, list, str, tuple, int, float, complex, bytes, type, set и другие (в версии   
Python 2.6, но не в 3.0, имя типа file является синонимом для open). Эти имена   
представляют не просто функции преобразования, а настоящие конструкторы   
объектов, хотя при решении несложных задач вы можете рассматривать их   
как простые функции.  
Модуль types, входящий в состав стандартной библиотеки, также предостав-  
ляет дополнительные имена типов, не являющиеся встроенными (например,   
типы функций� в Python 2.6, но не в 3.0, этот модуль также включает сино-  
Python 2.6, но не в 3.0, этот модуль также включает сино-  
 2.6, но не в 3.0, этот модуль также включает сино-  
нимы встроенных типов) и предоставляет возможность проверять тип объекта   
с помощью функции isinstance. Например, все следующие операции проверки   
возвращают истину:  
type([1]) == type([]) # Сравнивание с типом другого списка  
type([1]) == list # Сравнивание с именем типа  
isinstance([1], list) # Список или объект класса, производного от list  
   
import types # В модуле types определены имена других типов  
def f(): pass  
type(f) == types.FunctionType  
Поскольку в современных версиях Python от типов можно порождать дочерние   
классы, рекомендуется не пренебрегать функцией isinstance. Подробнее о соз-  
дании дочерних классов от встроенных типов в версии Python 2.2 (и выше) рас-  
сказывается в главе 31.  
Кроме того, в главе 31 мы исследуем, как функция type(X) и операции провер-  
ки типа применяются к экземплярам пользовательских классов. В двух сло-  
вах: в Python 3.0 и для классов «нового стиля» в Python 2.6 типом экземпля-  
Python 3.0 и для классов «нового стиля» в Python 2.6 типом экземпля-  
 3.0 и для классов «нового стиля» в Python 2.6 типом экземпля-  
Python 2.6 типом экземпля-  
 2.6 типом экземпля-  
ра класса является сам класс, на основе которого был создан экземпляр. Для   
«классических» классов в Python 2.6 (и в более ранних версиях) экземпляры   
любых классов имеют тип «instance», поэтому, чтобы получить более или ме-  
instance», поэтому, чтобы получить более или ме-  
», поэтому, чтобы получить более или ме-  
нее осмысленные результаты, мы должны сравнивать атрибуты \_\_class\_\_ эк-  
земпляров. Так как мы пока не готовы обсуждать все, что касается классов,   
отложим дальнейшее обсуждение этой темы до главы 31.  
Другие типы в Python  
Помимо базовых объектов, рассмотренных в этой части книги, и объектов,   
представляющих элементы программ, такие как функции, модули и клас-  
сы, с которыми мы встретимся ниже, в составе Python обычно устанавлива-  
Python обычно устанавлива-  
 обычно устанавлива-  
ются десятки других типов объектов, доступных в виде связанных расши-  
рений на языке C или в виде классов языка Python – объекты регулярных   
выражений, файлы DBM, компоненты графического интерфейса, сетевые   
сокеты и так далее.  
Главное отличие между этими и встроенными типами, которые мы до сих пор   
рассматривали, состоит в том, что для встроенных типов языком предоставля-  
ется специальный синтаксис создания объектов этих типов (например, 4 – для

Ловушки встроенных типов   
313  
целого числа, [1,2] – для списка, функция open – для файла, а def и lambda – для   
функций). Другие типы обычно доступны в модулях стандартной библиотеки,   
которые необходимо импортировать перед использованием. Например, чтобы   
создать объект регулярного выражения, необходимо импортировать модуль re   
и вызвать функцию re.compile(). Полное руководство по всем типам, доступ-  
ным в программах на языке Python, вы найдете в справочнике по библиотеке   
Python.  
Ловушки встроенных типов  
Это окончание нашего обзора базовых типов данных. Мы завершаем эту часть   
книги обсуждением общих проблем, с которыми сталкиваются новые пользо-  
ватели (а иногда и эксперты), и их решений. Отчасти – это краткий обзор идей,   
которые мы уже рассматривали, но они достаточно важны, чтобы вновь вер-  
нуться к ним.  
Операция присваивания создает ссылку, а не копию  
Поскольку это центральное понятие, я напомню о нем еще раз: вы должны   
понимать, что может происходить с разделяемыми ссылками в вашей про-  
грамме. Например, ниже создается объект списка, связанный с именем L, на   
это имя также ссылается элемент списка M. Таким образом, изменение спи-  
ска с помощью имени L приведет к изменению списка, на который ссылается   
и список M:  
>>> L = [1, 2, 3]  
>>> M = [‘X’, L, ‘Y’] # Встраивает ссылку из L  
>>> M  
[‘X’, [1, 2, 3], ‘Y’]  
   
>>> L[1] = 0 # Список M также изменяется  
>>> M  
[‘X’, [1, 0, 3], ‘Y’]  
Обычно этот эффект обретает важность только в больших программах, и как   
правило, совместное использование ссылок – это именно то, что необходимо.   
Если это является нежелательным, вы всегда можете явно создать копию объ-  
екта. Для списков всегда можно создать поверхностную копию с помощью опе-  
рации извлечения среза с незаданными пределами:  
>>> L = [1, 2, 3]  
>>> M = [‘X’, L[:], ‘Y’] # Встраивается копия L  
>>> L[1] = 0 # Изменяется только L, но не M   
>>> L  
[1, 0, 3]  
>>> M  
[‘X’, [1, 2, 3], ‘Y’]  
Не забывайте, что значениями пределов по умолчанию являются 0 и длина по-  
следовательности, откуда извлекается срез. Если оба значения опущены, в ре-  
зультате операции будут извлечены все элементы последовательности, что соз-  
даст поверхностную копию (новый, не разделяемый ни с кем, объект).

314   
Глава 9. Кортежи, файлы и все остальное   
Операция повторения добавляет   
один уровень вложенности  
Операция повторения последовательности добавляет последовательность саму   
к себе заданное число раз. Это правда, но когда появляются вложенные изме-  
няемые последовательности, эффект может не всегда получиться таким, как   
вы ожидаете. Например, в следующем примере переменной X присваивается   
список L, повторенный четырежды, тогда как переменной Y присваивается по-  
вторенный четырежды список, содержащий L:  
>>> L = [4, 5, 6]  
>>> X = L \* 4 # Все равно, что [4, 5, 6] + [4, 5, 6] + ...  
>>> Y = [L] \* 4 # [L] + [L] + ... = [L, L,...]  
   
>>> X  
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]  
>>> Y  
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]  
Так как во второй операции повторения L является вложенным списком, то в Y   
попадают ссылки на оригинальный список, связанный с именем L, вследствие   
чего начинают проявляться те же побочные эффекты, о которых говорилось   
в предыдущем разделе:  
>>> L[1] = 0 # Воздействует на Y, но не на X  
>>> X  
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]  
>>> Y  
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]  
Здесь можно применить то же решение, что и в предыдущем разделе, так как   
это в действительности всего лишь другой способ получить разделяемые ссыл-  
ки на изменяемый объект. Если вы помните, операции повторения, конкате-  
нации и извлечения среза создают только поверхностные копии объектов, что   
чаще всего и бывает необходимо.  
Избегайте создания циклических структур данных  
На самом деле мы уже сталкивались с этим понятием в предыдущем упраж-  
нении: если объект-коллекция содержит ссылку на себя, он называется ци-  
клическим объектом. Всякий раз, когда интерпретатор Python обнаруживает   
циклическую ссылку, он выводит [...], чтобы не попасть в бесконечный цикл:  
>>> L = [‘grail’] # Добавление ссылки на самого себя  
>>> L.append(L) # Создает циклический объект: [...]  
>>> L  
[‘grail’, [...]]  
Кроме понимания того, что три точки в квадратных скобках означают цикли-  
ческую ссылку в объекте, это случай заслуживает особого внимания, т.к. он   
может привести к сбоям – циклические структуры могут вызывать неожи-  
данное зацикливание программного кода, если вы не все предусмотрели. На-  
пример, некоторые программы хранят список или словарь уже посещенных   
элементов, с помощью которого обнаруживают попадание в цикл. Советы по   
устранению этих неполадок приводятся в упражнениях к первой части книги,

В заключение   
315  
в главе 3� кроме того, решение проблемы приводится также в конце главы 24,   
в программе reloadall.py.  
Не создавайте циклические ссылки, если они действительно не нужны. Иногда   
бывают серьезные основания для создания циклических ссылок, но если ваш   
программный код не предусматривает их обработку, вам не следует слишком   
часто заставлять свои объекты ссылаться на самих себя.  
Неизменяемые типы не могут изменяться   
непосредственно  
Наконец, вы не можете изменять неизменяемые объекты непосредственно.   
Вместо этого создайте новый объект – с помощью операций извлечения среза,   
конкатенации и так далее, и присвойте, если это необходимо, первоначальной   
переменной:  
T = (1, 2, 3)  
   
T[2] = 4 # Ошибка!  
   
T = T[:2] + (4,) # Все в порядке: (1, 2, 4)  
Это может показаться лишней работой, но выигрыш в том, что неизменяемые   
объекты, такие как кортежи и строки, не порождают приведенных выше про-  
блем, т.к. они не могут изменяться непосредственно и не подвержены, как спи-  
ски, побочным эффектам такого рода.  
В заключение  
В этой главе были исследованы последние два базовых типа объектов – корте-  
жи и файлы. Мы узнали, что кортежи поддерживают все операции, обычные   
для последовательностей, но не имеют методов и не позволяют выполнять из-  
менения непосредственно в объекте, потому что они относятся к категории не-  
изменяемых объектов. Мы также узнали, что объекты-файлы возвращаются   
функцией open и предоставляют методы чтения и записи данных. Мы иссле-  
довали проблему преобразования объектов Python в строку и обратно, чтобы   
иметь возможность сохранять их в файле, и познакомились с модулями pickle   
и struct, реализующими дополнительные возможности (сериализация объек-  
тов и работа с двоичными данными). В заключение мы рассмотрели некото-  
рые свойства, общие для всех типов объектов (например, разделяемые ссыл-  
ки), и прошлись по списку часто встречающихся ошибок (ловушек), связанных   
с типами объектов.  
В следующей части мы обратимся к теме синтаксиса инструкций в языке   
Python – здесь мы исследуем все основные процедурные инструкции. Следу-  
ющая глава открывает эту часть книги с введения в общую синтаксическую   
модель языка Python, которая применима ко всем типам операторов. Одна-  
ко прежде чем двинуться дальше, ознакомьтесь с контрольными вопросами   
к главе, а затем проработайте упражнения к этой части, чтобы коротко по-  
вторить основные понятия. Операторы в основном всего лишь создают и обра-  
батывают объекты, поэтому прежде чем продолжать чтение, вам необходимо   
проверить владение этой темой, выполнив упражнения.

316   
Глава 9. Кортежи, файлы и все остальное   
Закрепление пройденного  
Контрольные вопросы  
1. Как определить размер кортежа? Почему этот инструмент стоит обособленно?  
2. Напишите выражение, которое изменит первый элемент в кортеже. Кортеж   
со значением (4,5,6) должен стать кортежем со значением (1,5,6).  
3. Какое значение используется по умолчанию в аргументе режима обработки   
файла в функции open?  
4. Каким модулем можно воспользоваться для сохранения объектов Python   
в файл, чтобы избежать выполнения преобразований объектов в строки   
вручную?  
5. Как можно выполнить копирование всех частей вложенной структуры   
в одной инструкции?  
6. В каких случаях интерпретатор рассматривает объект как «истину»?  
7. В чем состоит ваша цель?  
Ответы  
1. Встроенная функция len возвращает длину (количество содержащихся эле-  
ментов) любого контейнерного объекта, включая и кортежи. Это – встро-  
енная функция, а не метод, и потому может применяться к самым разным   
типам объектов. Вообще говоря, встроенные функции и операторы выра-  
жений зачастую могут применяться к объектам самых разных типов� ме-  
тоды являются более узкоспециализированными инструментами, которые   
могут применяться только к объектам одного типа, однако некоторые типы   
могут иметь одноименные методы (например, методом index обладают спи-  
ски и кортежи).  
2. Поскольку кортежи являются неизменяемыми, в действительности их   
нельзя изменить непосредственно, но можно создать новый кортеж с же-  
лаемым значением. Первый элемент заданного кортежа T = (4,5,6) можно   
изменить, создав новый по частям с помощью операций извлечения среза   
и конкатенации: T = (1,) + T[1:]. (Не забывайте, что в кортежах из одного эле-  
мента обязательно должна присутствовать завершающая запятая.) Также   
можно было бы преобразовать кортеж в список, выполнить необходимое из-  
менение непосредственно в списке и произвести обратное преобразование   
в кортеж, но это более дорогостоящая последовательность действий, кото-  
рая редко используется на практике� просто используйте списки, если зара-  
нее известно, что может потребоваться изменить объект непосредственно.  
3. Аргумент режима открытия файла в функции open по умолчанию имеет   
значение ‘r’, то есть файл открывается для чтения в текстовом режиме. Что-  
бы открыть текстовый файл для чтения, достаточно передать функции одно   
только имя файла.  
4. Для сохранения объектов Python в файле можно воспользоваться модулем   
pickle, что устранит необходимость явного преобразования объектов в стро-  
ки. Модуль struct позволяет выполнять похожие действия, но в предполо-  
жении, что данные хранятся в файле в упакованном двоичном формате.

Закрепление пройденного   
317  
5. Чтобы скопировать все вложенные части структуры X, можно импортиро-  
вать модуль copy и вызвать функцию copy.deepcopy(X). Однако такой способ   
редко можно встретить на практике – ссылок обычно бывает достаточно и,   
как правило, в большинстве случаев достаточно бывает создать поверхност-  
ную копию (например, aList[:], aDict.copy()).  
6. Объект рассматривается как «истина», если он является либо ненулевым   
числом, либо непустым объектом коллекции. Встроенные слова True и False   
по сути являются предопределенными именами числовых значений 1 и 0 со-  
ответственно.  
7. В число допустимых ответов входят «Изучить язык Python», «Перейти   
к следующей части книги» или «Найти святую чашу Грааля».  
Упражнения ко второй части  
В этом разделе вам предлагается снова пройтись по основам встроенных объ-  
ектов. Как и прежде, вам попутно могут встретиться новые понятия, поэтому   
обязательно сверьтесь с ответами в приложении B, когда закончите (и даже   
если еще не закончили).   
Если у вас не так много свободного времени, я рекомендую начать с упражне-  
ний 10 и 11 (так как они наиболее практичные), а затем, когда появится время,   
пройти все упражнения от первого до последнего. Во всех упражнениях необ-  
ходимо знание фундаментальных сведений, поэтому постарайтесь выполнить   
как можно большую их часть.  
1. Основы. Поэкспериментируйте в интерактивной оболочке с наиболее часто   
используемыми операциями, которые вы найдете в таблицах второй ча-  
сти. Для начала запустите интерактивный сеанс работы с интерпретатором   
Python, введите все нижеследующие выражения и попробуйте объяснить   
происходящее. Обратите внимание, что в некоторых строках точка с за-  
пятой используется как разделитель инструкций, что позволяет уместить   
в одной строке несколько инструкций, например: в строке X=1;X выполняет-  
ся присваивание значения переменной и последующий его вывод (подроб-  
нее о синтаксисе инструкций рассказывается в следующей части книги).   
Кроме того, запомните, что запятая между выражениями обычно означает   
создание кортежа, даже в отсутствие круглых скобок: выражение X,Y,Z –   
это кортеж из трех элементов, который интерпретатор выведет, заключив   
в круглые скобки.  
2 \*\* 16  
2 / 5, 2 / 5.0  
   
“spam” + “eggs”  
S = “ham”  
“eggs “ + S  
S \* 5  
S[:0]  
“green %s and %s” % (“eggs”, S)  
‘green {0} and {1}’.format(‘eggs’, S)  
   
(‘x’,)[0]  
(‘x’, ‘y’)[1]

318   
Глава 9. Кортежи, файлы и все остальное   
L = [1,2,3] + [4,5,6]  
L, L[:], L[:0], L[-2], L[-2:]  
([1,2,3] + [4,5,6])[2:4]  
[L[2], L[3]]  
L.reverse( ); L  
L.sort( ); L  
L.index(4)  
   
{‘a’:1, ‘b’:2}[‘b’]  
D = {‘x’:1, ‘y’:2, ‘z’:3}  
D[‘w’] = 0  
D[‘x’] + D[‘w’]  
D[(1,2,3)] = 4  
list(D.keys()), list(D.values()), (1,2,3) in D  
   
[[]], [“”,[],( ),{},None]  
2. Индексирование  и  извлечение  среза. В интерактивной оболочке создайте   
список с именем L, который содержит четыре строки или числа (например,   
L = [0,1,2,3]). Затем исследуйте следующие случаи – они могут никогда не   
встретиться вам на практике, но они заставят вас задуматься об основах   
реализации, что может оказаться полезным в практических ситуациях:  
 •  
Что произойдет, если попытаться получить доступ к элементу, индекс   
которого выходит за пределы списка (например, L[4])?  
 •  
Что произойдет, если попытаться извлечь срез, выходящий за пределы   
списка (например, L[-1000:100])?  
 •  
Наконец, как отреагирует интерпретатор на попытку извлечь последо-  
вательность в обратном порядке, когда нижняя граница больше верхней   
(например, L[3:1])? Подсказка: попробуйте выполнить операцию присва-  
ивания такому срезу (L[3:1] = [‘?’]) и посмотреть, куда будет помещено   
значение. Как вы думаете, это то же самое явление, что и при попытке   
извлечь срез, выходящий за пределы списка?  
3. Индексирование, извлечение среза и инструкция del. Создайте другой спи-  
сок L с четырьмя элементами и присвойте одному из элементов пустой спи-  
сок (например, L[2] = []). Что произошло? Затем присвойте пустой список   
срезу (L[2:3] = []). Что случилось на этот раз? Не забывайте, что операция   
присваивания срезу сначала удаляет срез, а затем вставляет новое значение   
в заданную позицию.  
Инструкция del удаляет элемент с указанным смещением, ключом, атрибу-  
том или именем. Используйте ее для удаления элемента вашего списка (на-  
пример, del L[0]). Что произойдет, если попробовать удалить целый срез (del   
L[1:])? Что произойдет, если срезу присвоить объект, который не является   
последовательностью (L[1:2] = 1)?  
4. Кортежи. Введите следующие строки:  
>>> X = ‘spam’  
>>> Y = ‘eggs’  
>>> X, Y = Y, X  
Как вы думаете, что произойдет с переменными X и Y после выполнения этой   
последовательности действий?

Закрепление пройденного   
319  
5. Ключи словарей. Рассмотрите следующий фрагмент:  
>>> D = {}  
>>> D[1] = ‘a’  
>>> D[2] = ‘b’  
Вы знаете, что словари не поддерживают доступ к элементам по смещени-  
ям� попробуйте объяснить происходящее здесь. Может быть, следующий   
пример прояснит ситуацию? (Подсказка: к какой категории типов относят-  
ся строки, целые числа и кортежи?)  
>>> D[(1, 2, 3)] = ‘c’  
>>> D  
{1: ‘a’, 2: ‘b’, (1, 2, 3): ‘c’}  
6. Индексирование словарей. Создайте словарь с именем D и с тремя записями   
для ключей ‘a’, ‘b’ и ‘c’. Что произойдет, если попытаться обратиться к эле-  
менту с несуществующим ключом (D{‘d’})? Что сделает интерпретатор, если   
попробовать присвоить значение несуществующему ключу (D[‘d’] = ‘spam’)?   
Как это согласуется с операциями доступа и присваивания элементам спи-  
сков при использовании индексов, выходящих за их пределы? Не напоми-  
нает ли вам такое поведение правила, применяемые к переменным?  
7. Общие операции. Получите в интерактивной оболочке ответы на следую-  
щие вопросы:  
 •  
Что произойдет, если попытаться использовать оператор + с операндами   
различных типов (например, строка+список, список+кортеж)?  
 •  
Будет ли работать оператор +, когда один из операндов является слова-  
рем?  
 •  
Будет ли работать метод append со списками и со строками? Можно ли ис-  
пользовать метод keys со списками? (Подсказка: что предполагает метод   
append о заданном объекте?)  
 •  
Наконец, какой тип объекта будет получен, когда операция конкатена-  
ции применяется к двум спискам или двум строкам?  
8. Индексирование  строк. Определите строку S из четырех символов: S =   
“spam”. Затем введите следующее выражение: S[0][0][0][0][0]. Можете ли вы   
объяснить, что произошло на этот раз? (Подсказка: не забывайте, строки –   
это коллекции символов, а символы в языке Python представлены односим-  
вольными строками.) Будет ли это выражение работать, если применить его   
к списку, такому как [‘s’, ‘p’, ‘a’, ‘m’]? Почему?  
9. Неизменяемые типы. Определите еще раз строку S из четырех символов:   
S = “spam”. Напишите операцию присваивания, которая изменила бы стро-  
ку на “slam”, используя только операции извлечения среза и конкатенации.   
Возможно ли выполнить то же самое действие с использованием операций   
индексирования и конкатенации? С помощью присваивания по индексу   
элемента?  
10. Вложенные  структуры. Создайте структуру данных для представления   
вашей личной информации: имя (имя, фамилия, отчество), возраст, долж-  
ность, адрес, электронный адрес и номер телефона. При построении струк-  
туры вы можете использовать любые комбинации встроенных объектов

320   
Глава 9. Кортежи, файлы и все остальное   
(списки, кортежи, словари, строки, числа). Затем попробуйте обратить-  
ся к отдельным элементам структуры по индексам. Являются ли какие-  
нибудь объекты более предпочтительными для данной структуры?  
11. Файлы. Напишите сценарий, который создает и открывает для записи но-  
вый файл с именем myfile.txt и записывает в него строку “Hello file world!”.   
Затем напишите другой сценарий, который открывает файл myfile.txt,   
читает и выводит на экран его содержимое. Запустите поочередно эти сце-  
нарии из командной строки. Появился ли новый файл в каталоге, откуда   
были запущены сценарии? Что произойдет, если указать другой каталог   
в имени файла, которое передается функции open? Примечание: методы за-  
писи в файлы не добавляют символ новой строки к записываемым строкам.   
Добавьте символ \n явно в конец вашей строки, если хотите получить в фай-  
ле полностью завершенную строку.

Часть III.  
Инструкции и синтаксис

Глава 10.  
   
Введение в инструкции языка Python  
Теперь, когда вы познакомились с базовыми встроенными типами объектов   
языка Python, мы начинаем исследование фундаментальных форм инструк-  
ций. Как и в предыдущей части книги, мы начнем с общего представления син-  
таксиса инструкций и затем, в нескольких следующих главах, более подробно   
рассмотрим конкретные инструкции.  
Выражаясь простым языком, инструкции – это то, что вы пишете, чтобы со-  
общить интерпретатору, какие действия должна выполнять ваша программа.   
Если программа «выполняет какие-то действия», то инструкции – это способ   
указать, какие именно действия должна выполнять программа. Python – это   
процедурный язык программирования, основанный на использовании ин-  
струкций� комбинируя инструкции, вы задаете процедуру, которую выполня-  
ет интерпретатор в соответствии с целями программы.   
Структура программы на языке Python  
Другой способ понять роль инструкций состоит в том, чтобы вновь вернуть-  
ся к иерархии понятий, представленной в главе 4, в которой рассказывалось   
о встроенных объектах и выражениях, управляющих ими. Эта глава рассма-  
тривает следующую ступень иерархии:  
1. Программы делятся на модули.  
2. Модули содержат инструкции.  
3. Инструкции состоят из выражений.  
4. Выражения создают и обрабатывают объекты.  
Синтаксис языка Python по сути построен на инструкциях и выражениях. Вы-  
ражения обрабатывают объекты и встраиваются в инструкции. Инструкции   
представляют собой более крупные логические блоки программы – они напря-  
мую используют выражения для обработки объектов, которые мы рассматри-  
вали в предыдущих главах. Кроме того, инструкции – это место, где создаются   
объекты (например, в инструкциях присваивания), а в некоторых инструкци-  
ях создаются совершенно новые виды объектов (функции, классы и так далее).

324   
Глава 10. Введение в инструкции языка Python   
Инструкции всегда присутствуют в модулях, которые сами управляются ин-  
струкциями.  
Инструкции в языке Python  
В табл. 10.1 приводится набор инструкций языка Python. В этой части книги   
рассматриваются инструкции, которые в таблице расположены от начала и до   
инструкций break и continue. Ранее неофициально вам уже были представлены   
некоторые из инструкций, присутствующих в табл. 10.1. В этой части книги   
будут описаны подробности, опущенные ранее� вашему вниманию будут пред-  
ставлены остальные процедурные инструкции языка Python, а также будет   
рассмотрена общая синтаксическая модель. Инструкции, расположенные   
в табл. 10.1 ниже, имеют отношение к крупным блокам программы – функци-  
ям, классам, модулям и исключениям, и заключают в себе крупные понятия   
программирования, поэтому каждой из них будет посвящен отдельный раздел.   
Более экзотические инструкции, такие как del (которая удаляет различные   
компоненты), раскрываются далее в книге или в стандартной документации   
по языку Python.  
Таблица 10.1. Инструкции языка Python  
Инструкция  
Роль  
Пример  
Присваивание  
Создание ссылок  
a, \*b = ‘good’, ‘bad’, ‘ugly’  
Вызовы и другие   
выражения  
Запуск функций  
log.write(“spam, ham”)  
Вызов функции   
print  
Вывод объектов  
print(‘The Killer’, joke)  
if/elif/else  
Операция выбора  
if “python” in text:  
 print(text)  
for/else  
Обход последовательно-  
сти в цикле  
for x in mylist:  
 print(x)  
while/else  
Циклы общего назначе-  
ния  
while X > Y:  
 print(‘hello’)  
pass  
Пустая инструкция-  
заполнитель  
while True:  
 pass  
break  
Выход из цикла  
while True:  
 if exittest(): break  
continue  
Переход в начало цикла  
while True:  
 if skiptest(): continue  
def  
Создание функций   
и методов  
def f(a, b, c=1, \*d):  
 print(a+b+c+d[0])  
return  
Возврат результата  
def f(a, b, c=1, \*d):  
 return a+b+c+d[0]

Структура программы на языке Python   
325  
Инструкция  
Роль  
Пример  
yield  
Функции-генераторы  
def gen(n):  
 for i in n: yield i\*2  
global  
Пространства имен  
x = ‘old’  
def function():  
 global x, y; x = ‘new’  
nonlocal  
Пространства имен (3.0+)  
def outer():  
 x = ‘old’  
 def function():  
 nonlocal x; x = ‘new’  
import  
Доступ к модулям  
import sys  
from  
Доступ к атрибутам   
модуля  
from sys import stdin  
class  
Создание объектов  
class Subclass(Superclass):  
 staticData = []  
 def method(self): pass  
try/except/finally  
Обработка исключений  
try:  
 action()  
except:  
 print(‘action error’)  
raise  
Возбуждение исключений raise endSearch(location)  
assert  
Отладочные проверки  
assert X > Y, ‘X too small’  
with/as  
Менеджеры контекста   
(2.6+)  
with open(‘data’) as myfile:  
 process(myfile)  
del  
Удаление ссылок  
del data[k]  
del data[i:j]  
del obj.attr  
del variable  
В табл. 10.1 перечислены разновидности инструкций в версии Python 3.0 – эле-  
 табл. 10.1 перечислены разновидности инструкций в версии Python 3.0 – эле-  
табл. 10.1 перечислены разновидности инструкций в версии Python 3.0 – эле-  
. 10.1 перечислены разновидности инструкций в версии Python 3.0 – эле-  
10.1 перечислены разновидности инструкций в версии Python 3.0 – эле-  
Python 3.0 – эле-  
 3.0 – эле-  
менты программного кода, каждый из которых имеет свой характерный син-  
таксис и назначение. Ниже приводятся несколько замечаний к таблице:  
 •  
Инструкции присваивания могут принимать различные синтаксические   
формы, которые описываются в главе 11: простое, присваивание последова-  
тельностей, комбинированное присваивание и другие.  
 •  
В версии 3.0 print не является ни зарезервированным словом, ни инструк-  
цией – это встроенная функция. Однако она практически всегда выпол-  
няется как инструкция (то есть занимает отдельную строку в программе),

326   
Глава 10. Введение в инструкции языка Python   
поэтому ее обычно воспринимают как инструкцию. Мы поближе познако-  
мимся с функцией print в главе 11.  
 •  
Начиная с версии 2.5 yield в действительности является выражением, а не   
инструкцией. Как и функция print, это выражение обычно занимает от-  
дельную строку, и потому оно было включено в табл. 10.1. Однако иногда   
в сценариях выполняется присваивание этой инструкции или извлечение   
результата из нее, как будет показано в главе 20. Кроме того, в отличие от   
print, имя yield является зарезервированным словом.  
Большая часть инструкций, перечисленных в табл. 10.1, также имеется в вер-  
сии Python 2.6. Ниже приводятся несколько замечаний для тех, кто пользует-  
Python 2.6. Ниже приводятся несколько замечаний для тех, кто пользует-  
 2.6. Ниже приводятся несколько замечаний для тех, кто пользует-  
ся Python 2.6 или более ранними версиями:  
 •  
В версии 2.6 инструкция nonlocal недоступна. Как мы увидим в главе 17,   
существуют другие способы добиться того же эффекта в отношении при-  
сваивания значений переменным.  
 •  
В версии 2.6 print является не функцией, а инструкцией со своим характер-  
ным синтаксисом, который описывается в главе 11.  
 •  
exec (в версии 3.0 – встроенная функция, позволяющая выполнять фраг-  
менты программного кода) в версии 2.6 также является инструкцией со   
своим характерным синтаксисом. Так как она поддерживает возможность   
заключения аргументов в круглые скобки, в версии 2.6 ее можно использо-  
вать как функцию.  
 •  
В версии 2.5 инструкции try/except и try/finally были объединены: ранее   
это были две самостоятельные инструкции, но теперь мы можем использо-  
вать предложения except и finally одновременно, в одной инструкции try.   
 •  
В версии 2.5 инструкция with/as была необязательным расширением, и она   
была недоступна, если в программный код не включить инструкцию from   
\_\_future\_\_ import with\_statement (глава 33).  
История о двух if  
Однако прежде чем углубиться в детали какой-либо конкретной инструкции   
из табл. 10.1, я хочу обратить ваше внимание на синтаксис инструкций в языке   
Python, показав, как не надо писать программный код, чтобы у вас была воз-  
можность сравнить его с другими синтаксическими моделями, которые, воз-  
можно, вы видели ранее.  
Рассмотрим следующую условную инструкцию на языке C:  
if (x > y) {  
 x = 1;  
 y = 2;  
}  
Это могла бы быть инструкция на языке C, C++, Java, JavaScript или Perl. А те-  
C, C++, Java, JavaScript или Perl. А те-  
, C++, Java, JavaScript или Perl. А те-  
C++, Java, JavaScript или Perl. А те-  
++, Java, JavaScript или Perl. А те-  
Java, JavaScript или Perl. А те-  
, JavaScript или Perl. А те-  
JavaScript или Perl. А те-  
 или Perl. А те-  
Perl. А те-  
. А те-  
перь взгляните на эквивалентную инструкцию на языке Python:  
if x > y:  
 x = 1  
 y = 2

История о двух if   
327  
Первое, что бросается в глаза, – инструкция на языке Python выглядит ком-  
Python выглядит ком-  
 выглядит ком-  
пактнее, точнее, в ней меньше синтаксических элементов. Это соответствует   
основным принципам языка� так как Python – это язык сценариев, его основ-  
Python – это язык сценариев, его основ-  
 – это язык сценариев, его основ-  
ная цель состоит в том, чтобы облегчить жизнь программистам за счет меньше-  
го объема ввода с клавиатуры.  
Если быть более точным, то, сравнив две синтаксических модели, можно заме-  
тить, что язык Python один новый элемент добавляет, а три элемента, которые   
присутствуют в языках, подобных языку C, ликвидирует.  
Что добавляет язык Python  
Один из новых синтаксических элементов в языке Python – это символ двое-  
Python – это символ двое-  
 – это символ двое-  
точия (:). Все составные инструкции в языке Python (то есть инструкции, ко-  
Python (то есть инструкции, ко-  
 (то есть инструкции, ко-  
торые включают вложенные в них инструкции) записываются в соответствии   
с одним и тем же общим шаблоном, когда основная инструкция завершается   
двоеточием, вслед за которым располагается вложенный блок кода, обычно   
с отступом под строкой основной инструкции, как показано ниже:  
Основная инструкция:  
 Вложенный блок инструкций  
Двоеточие является обязательным, а его отсутствие является самой распро-  
страненной ошибкой, которую допускают начинающие программисты, –   
я встречал тысячи подтверждений этому в учебных классах. Фактически   
если вы плохо знакомы с языком Python, то вы почти наверняка очень скоро   
забудете о символе двоеточия. Большинство текстовых редакторов, обладаю-  
щих функцией подсветки синтаксиса, делают эту ошибку легко заметной, а с   
опытом вырабатывается привычка вставлять двоеточие бессознательно (да   
так, что вы начинаете вводить двоеточие в программный код на языке C++,   
что приводит к большому числу весьма интересных сообщений об ошибках от   
компилятора C++!).  
Что Python устраняет  
Хотя Python требует ввода дополнительного символа двоеточия, существуют   
три элемента, обязательных для языков, подобных языку C, которые языку   
Python не требуются.  
Круглые скобки не обязательны  
Первый элемент – это пара круглых скобок, окружающих условное выражение   
в основной инструкции:  
if (x < y)  
Круглые скобки здесь являются обязательными во многих C-подобных языках.   
В языке Python это не так – мы просто можем опустить скобки, и инструкция   
будет работать точно так же:  
if x < y  
Точнее говоря, так как каждое выражение может быть заключено в скобки,   
присутствие их не будет противоречить синтаксису языка Python, и они не   
будут считаться ошибкой. Но не делайте этого: вы лишь понапрасну будете

328   
Глава 10. Введение в инструкции языка Python   
изнашивать свою клавиатуру, а окружающим сразу будет видно, что вы типич-  
ный программист на C, еще только изучающий Python (когда-то и я был таким   
же). Стиль языка Python состоит в том, чтобы вообще опускать скобки в подоб-  
Python состоит в том, чтобы вообще опускать скобки в подоб-  
 состоит в том, чтобы вообще опускать скобки в подоб-  
ных инструкциях.  
Конец строки является концом инструкции  
Второй, еще более важный синтаксический элемент, который вы не найде-  
те в программном коде на языке Python, – это точка с запятой. В языке Py-  
Python, – это точка с запятой. В языке Py-  
 – это точка с запятой. В языке Py-  
Py-  
thon не требуется завершать инструкции точкой с запятой, как это делается   
в C-подобных языках:  
x = 1;  
Общее правило в языке Python гласит, что конец строки автоматически счита-  
Python гласит, что конец строки автоматически счита-  
 гласит, что конец строки автоматически счита-  
ется концом инструкции, стоящей в этой строке. Другими словами, вы можете   
отбросить точку с запятой, и инструкция будет работать точно так же:  
x = 1  
Существует несколько способов обойти это правило, как будет показано чуть   
ниже. Но в общем случае большая часть программного кода на языке Python   
пишется по одной инструкции в строке, и тогда точка с запятой не требуется.  
В данном случае, если вы скучаете по тем временам, когда программирова-  
ли на языке C (если такое состояние вообще возможно…), можете продолжать   
вставлять точки с запятой в конце каждой инструкции – синтаксис языка до-  
пускает это. Но не делайте этого: потому что если вы будете поступать так,   
окружающим сразу будет видно, что вы остаетесь программистом на языке   
C, который никак не может переключиться на использование языка Python.   
Стиль языка Python состоит в том, чтобы вообще опускать точки с запятой.1  
Конец отступа – это конец блока  
Теперь третий, и последний, синтаксический компонент, который удаляет Py-  
Py-  
thon, и возможно, самый необычный для недавних экс-С-программеров (пока   
они не поработают с Python десять минут и не поймут, что в действительности   
это является достоинством языка), – вы не вводите ничего специально в ваш   
код, чтобы синтаксически пометить начало и конец вложенного блока кода.   
Вам не нужно вставлять begin/end, then/endif или фигурные скобки вокруг вло-  
женных блоков2, как это делается в C-подобных языках:  
if (x > y) {  
 x = 1;  
 y = 2;  
}  
Для этих целей в языке Python используются отступы, когда все инструкции   
в одном и том же вложенном блоке оформляются с одинаковыми отступами   
1   
Кстати, в JavaScript также можно не ставить точку с запятой в конце строки, но ши-  
роко распространена традиция ставить такой ограничитель, демонстрируя тем са-  
мым как раз хороший стиль оформления программного кода. – Примеч. перев.  
2   
Известна программистская шутка, уместная в данном контексте: это не bug, это   
feature. – Примеч. перев.

История о двух if   
329  
от левого края. По величине отступа интерпретатор определяет, где находится   
начало блока и где – конец:  
if x > y:  
 x = 1  
 y = 2  
Под отступами  в данном примере я подразумеваю пустое пространство слева   
от двух вложенных инструкций. Интерпретатор не накладывает ограничений   
на то, как выполняются отступы (для этого можно использовать символы про-  
бела или символы табуляции), и на величину отступов (допускается исполь-  
зовать любое число пробелов или символов табуляции). При этом отступ для   
одного вложенного блока может существенно отличаться от отступа для дру-  
гого блока. Синтаксическое правило состоит лишь в том, что все инструкции   
в пределах одного блока должны иметь один и тот же отступ от левого края.   
Если это не так, будет получена синтаксическая ошибка, и программный код   
не будет работать, пока вы не исправите отступ.  
Почему отступы являются частью синтаксиса?  
Правило оформления отступов может показаться необычным на первый взгляд   
для программистов, работавших с C-подобными языками программирования,   
но это было сделано преднамеренно и является одним из основных способов,   
которыми язык Python вынуждает программистов писать однородный и удо-  
бочитаемый программный код. По существу это означает, что вы должны вы-  
страивать свой программный код вертикально, выравнивая его в соответствии   
с логической структурой. В результате получается менее противоречивый и бо-  
лее удобочитаемый программный код (в отличие от большей части кода, напи-  
санного на C-подобных языках).  
Требования к выравниванию программного кода в соответствии с его логи-  
ческой структурой – это главная составляющая, способствующая созданию   
удобочитаемого кода, и, следовательно, кода, пригодного к многократному ис-  
пользованию и простого в сопровождении как вами, так и другими. Факти-  
чески даже если ранее вы никогда не использовали Python, после прочтения   
этой книги у вас должна выработаться привычка оформлять отступы в про-  
граммном коде для удобочитаемости в любом языке программирования. Язык   
Python вносит определенные ограничения, сделав отступы частью синтаксиса,   
но они достаточно важны для любого языка программирования и оказывают   
огромное влияние на применимость вашего программного кода.  
Ваш опыт может отличаться от моего, но когда я занимался разработкой пол-  
ный рабочий день, мне платили зарплату за работу над крупными старыми   
программами, написанными на языке C++, над которыми долгие годы труди-  
лись множество программистов. Практически у каждого программиста был   
свой стиль оформления программного кода. Например, мне часто приходилось   
изменять циклы while, написанные на языке C++, которые начинались при-  
мерно так:  
while (x > 0) {  
Даже еще не дойдя до отступов, мы можем столкнуться с тремя или четырьмя   
способами расстановки фигурных скобок в C-подобных языках. В организа-  
циях часто ведутся жаркие споры и пишутся стандарты по оформлению ис-

330   
Глава 10. Введение в инструкции языка Python   
ходных текстов программ (обсуждение которых сильно выходит за рамки про-  
блем, которые решаются в процессе программирования). Оставим этот вопрос   
в стороне, ниже приводится пример оформления, с которым мне часто прихо-  
дилось сталкиваться в программном коде на языке C++. Первый программист,   
который поработал над этим циклом, оформлял отступы четырьмя пробелами:  
while (x > 0) {  
 --------;  
 --------;  
Со временем этот программист был переведен на руководящую должность,   
и его место занял другой, который предпочитал использовать более широкие   
отступы:   
while (x > 0) {  
 --------;  
 --------;  
 --------;  
 --------;  
Позднее и он ушел на другую работу, а его место занял третий, которому не   
нравилось делать отступы:  
while (x > 0) {  
 --------;  
 --------;  
 --------;  
 --------;  
--------;  
--------;  
}  
И так далее. В конце блок завершается закрывающей фигурной скобкой (}),   
которая и делает этот фрагмент структурированным (о чем можно говорить   
с определенной долей сарказма). В любом блочно-структурированном языке   
программирования, будь то Python или другой язык, если вложенные блоки не   
имеют непротиворечивых отступов, их очень сложно читать, изменять и при-  
спосабливать для многократного использования. Отступ – это важная состав-  
ляющая поддержки удобочитаемости.  
Ниже приводится другой пример, на котором, возможно, вам приходилось   
обжечься, если вы достаточно много программировали на C-подобном языке.   
Взгляните на следующие инструкции языка C:  
if (x)  
 if (y)  
 statement1;  
else  
 statement2;  
К какому оператору if относится инструкция else? Это удивительно, но ин-  
струкция else относится к вложенному оператору if (if(y)), даже при том, что   
визуально она выглядит так, как если бы относилась к внешнему оператору   
if (if(x)). Это классическая ловушка языка C и она может привести к непра-  
вильному пониманию программного кода тем, кто его изменяет, и к появлению   
ошибок при его изменении, которые будут обнаружены, только когда марсоход   
врежется в скалу!

История о двух if   
331  
Такого не может произойти в языке Python, потому что отступы для него име-  
ют важное значение и программный код работает именно так, как выглядит.   
Взгляните на эквивалентный фрагмент на языке Python:  
if x:  
 if y:  
 statement1  
else:  
 statement2  
В этом примере инструкции if и else, расположенные на одной вертикаль-  
ной линии, связаны логически (внешний оператор if x). В некотором смысле   
Python – это язык типа ��SI��G (�hat �ou See Is �hat �ou Get – что ви-  
�hat �ou See Is �hat �ou Get – что ви-  
 �ou See Is �hat �ou Get – что ви-  
�ou See Is �hat �ou Get – что ви-  
 See Is �hat �ou Get – что ви-  
See Is �hat �ou Get – что ви-  
 Is �hat �ou Get – что ви-  
Is �hat �ou Get – что ви-  
 �hat �ou Get – что ви-  
�hat �ou Get – что ви-  
 �ou Get – что ви-  
�ou Get – что ви-  
 Get – что ви-  
Get – что ви-  
 – что ви-  
дишь, то и получаешь) – что вы видите, то и получаете, потому что порядок   
оформления программного кода определяет порядок его выполнения, незави-  
симо от предпочтений того, кто его пишет.  
Если этого недостаточно, чтобы осознать преимущества синтаксиса языка   
Python, приведу одну историю. В начале своей карьеры я работал в одной бла-  
гополучной компании, занимавшейся разработкой программных систем на   
языке C, который вообще не предъявляет никаких требований к оформлению   
отступов. Но, несмотря на это, когда в конце рабочего дня мы отправляли свой   
программный код в репозитарий, он автоматически проверялся сценарием,   
анализировавшим оформление отступов. Если сценарий обнаруживал несоот-  
ветствия, на следующий день утром мы получали электронные письма с заме-  
чаниями – и такой порядок завели наши начальники!  
По моему мнению, даже если язык программирования и не требует этого, хо-  
роший программист должен понимать, какое важное значение имеет выравни-  
вание для удобочитаемости и высокого качества программного кода. Тот факт,   
что в языке Python отступы были возведены в ранг синтаксиса, по большей   
части выглядит как достоинство языка.  
Наконец, имейте в виду, что практически любой текстовый редактор с друже-  
ственным (для программистов) интерфейсом обладает встроенной поддерж-  
кой синтаксической модели языка Python. В Python-среде разработки IDLE,   
например, отступы оформляются автоматически1, когда начинается ввод вло-  
женного блока� нажатие клавиши Backspace (забой) возвращает на один уровень   
вложенности выше, а кроме того, IDLE позволяет настроить величину отступов   
во вложенном блоке. Нет никаких стандартных требований к оформлению от-  
ступов: чаще всего используются четыре пробела или один символ табуляции   
на каждый уровень вложенности� вам самим решать, какой ширины отступы   
вы будете использовать. Выполняйте отступ вправо, чтобы открыть вложен-  
ный блок, и возвращайтесь на предыдущий уровень, чтобы закрыть его.   
Вообще говоря, недопустимо смешивать символы табуляции и пробелы для   
оформления отступов в одном и том же блоке, если делать это неединообраз-  
1   
Напомним, IDLE – среда разработки, специально созданная Гвидо ван Россумом   
в проекте Python. Первый релиз вышел вместе с версией Python 1.5.2. Одно из объ-  
яснений использования такого слова (idle – бездействующий, ленивый), в отличие   
от принятого в других языковых средах термина IDE, объявлено на сайте python.org   
(http://www.python.org/idle/doc/idlemain.html) и выглядит так: “IDLE – это акроним   
от Integrated DeveLopment Environment”. См. также главу 3. – Примеч. перев.

332   
Глава 10. Введение в инструкции языка Python   
но. Для оформления отступов в блоке используйте либо символы табуляции,   
либо пробелы, но не одновременно те и другие (в действительности Python 3.0   
теперь считает ошибкой непоследовательное использование символов табуля-  
ции и пробелов, как мы увидим в главе 12). Точно так же нежелательно сме-  
шивать символы табуляции и пробелы для оформления отступов в любом дру-  
гом структурированном языке программирования – такой программный код   
очень трудно будет читать следующему программисту, если в его текстовом   
редакторе отображение символов табуляции будет настроено иначе, чем у вас.   
C-подобные языки позволяют программистам нарушать это правило, но делать   
этого не следует, – в результате может получиться жуткая мешанина.  
Я не могу не подчеркнуть, что независимо от того, на каком языке вы програм-  
мируете, вы всегда должны быть последовательными в оформлении отступов   
для повышения удобочитаемости. Фактически если в начале вашей карьеры   
ваши учителя не приучили вас к этому, они просто навредили вам. Большин-  
ство программистов, особенно те, кому приходится читать чужой код, считают   
великим благом, что Python возвел отступы в ранг синтаксиса. Кроме того, на   
практике замена фигурных скобок символами табуляции является достаточно   
простой задачей для инструментальных средств, которые должны выводить   
программный код на языке Python. Вообще, делайте все то же, что вы делаете   
на языке C, но уберите фигурные скобки, и ваш программный код будет удо-  
влетворять правилам синтаксиса языка Python.  
Несколько специальных случаев  
Как уже упоминалось ранее, в синтаксической модели языка Python:  
 •  
Конец строки является концом инструкции, расположенной в этой строке   
(точка с запятой не требуется).  
 •  
Вложенные инструкции объединяются в блоки по величине отступов (без   
фигурных скобок).  
Эти правила охватывают большую часть программного кода на языке Python,   
который вы будете писать или с которым придется столкнуться. Однако суще-  
ствуют некоторые специальные правила, которые касаются оформления как   
отдельных инструкций, так и вложенных блоков.  
Специальные случаи оформления инструкций  
Обычно на каждой строке располагается одна инструкция, но вполне возмож-  
но для большей компактности записать несколько инструкций в одной строке,   
разделив их точками с запятой:  
a = 1; b = 2; print(a + b) # Три инструкции на одной строке  
Это единственный случай, когда в языке Python необходимо использовать точ-  
ки с запятой: как разделители инструкций. Однако такой подход не может   
применяться к составным инструкциям. Другими словами, в одной строке   
можно размещать только простые инструкции, такие как присваивание, print   
и вызовы функций. Составные инструкции по-прежнему должны находиться   
в отдельной строке (иначе всю программу можно было бы записать в одну стро-  
ку, что, скорее всего, не нашло бы понимания у ваших коллег!).

История о двух if   
333  
Другое специальное правило, применяемое к инструкциям, по сути является   
обратным к предыдущему: допускается записывать одну инструкцию в не-  
скольких строках. Для этого достаточно заключить часть инструкции в пару   
скобок – круглых (()), квадратных ([]) или фигурных ({}). Любой программный   
код, заключенный в одну из этих конструкций, может располагаться на не-  
скольких строках: инструкция не будет считаться законченной, пока интер-  
претатор Python не достигнет строки с закрывающей скобкой. Например, ли-  
терал списка можно записать так:  
mlist = [111,  
 222,  
 333]  
Так как программный код заключен в пару квадратных скобок, интерпретатор   
всякий раз переходит на следующую строку, пока не обнаружит закрывающую   
скобку. Литералы словарей в фигурных скобках (а также литералы множеств   
и генераторы словарей и множеств в Python 3.0) тоже могут располагаться в не-  
скольких строках, а с помощью круглых скобок можно оформить многостроч-  
ные кортежи, вызовы функций и выражения. Отступы в строках, где продол-  
жается инструкция, в учет не принимаются, хотя здравый смысл диктует, что   
строки все-таки должны иметь некоторые отступы для обеспечения удобочи-  
таемости.  
Круглые скобки являются самым универсальным средством, потому что в них   
можно заключить любое выражение. Добавьте левую скобку, и вы сможете пе-  
рейти на следующую строку и продолжить свою инструкцию:  
X = (A + B +  
 C + D)  
Между прочим, такой прием допускается применять и к составным инструк-  
циям. Если вам требуется записать длинное выражение, оберните его круглы-  
ми скобками и продолжите на следующей строке:  
if (A == 1 and  
 B == 2 and  
 C == 3):  
 print(‘spam’ \* 3)  
Еще одно старое правило также позволяет переносить инструкцию на следую-  
щую строку: если предыдущая строка заканчивается символом обратного сле-  
ша:  
X = A + B + \ # Альтернативный способ, который может быть источником ошибок  
 C + D  
Но это устаревшее правило, которое не рекомендовано к использованию в но-  
вых программах, потому что символы обратного слеша малозаметны и нена-  
дежны – не допускается наличие каких-либо других символов после символа   
обратного слеша, а случайное удаление символа обратного слеша может при-  
водить к неожиданным эффектам, если следующая строка может интерпрети-  
роваться, как самостоятельная инструкция. Кроме того, это рассматривается   
как пережиток языка C, где очень часто используются макроопределения #de-  
de-  
fine� в Питонляндии нужно все делать так, как это делают Питонисты, а не как   
программисты на языке C.

334   
Глава 10. Введение в инструкции языка Python   
Специальный случай оформления блока  
Как уже говорилось выше, инструкции во вложенном блоке обычно объединя-  
ются по величине отступа. Специальный случай: тело составной инструкции   
может располагаться в той же строке, что и основная инструкция, после сим-  
вола двоеточия:  
if x > y: print(x)  
Это позволяет записывать в одной строке условные операторы, циклы и так да-  
лее. Однако такой прием будет работать, только если тело составной инструк-  
ции не содержит других составных инструкций. То есть после двоеточия могут   
следовать только простые инструкции – инструкции присваивания, инструк-  
ции print, вызовы функций и подобные им. Крупные инструкции по-прежнему   
должны записываться в отдельных строках. Дополнительные части составных   
инструкций (такие как блоки else в условных инструкциях if, с которыми мы   
встретимся ниже) также должны располагаться в отдельных строках. Тело ин-  
струкции может состоять из нескольких простых инструкций, разделенных   
точкой с запятой, но такой стиль оформления не приветствуется.  
Вообще, хотя это не является обязательным требованием, но если вы будете   
размещать инструкции в отдельных строках и всегда будете оформлять отсту-  
пы для вложенных блоков, ваш программный код будет проще читать и вно-  
сить в него изменения. Кроме того, некоторые инструменты профилирования   
оказываются неспособными справиться с ситуациями, когда несколько ин-  
струкций располагаются в одной строке или когда тело составной инструкции   
оказывается на одной строке вместе с основной инструкцией. Поэтому в ваших   
интересах всегда стремиться оформлять программный код как можно проще.  
Чтобы увидеть одно из этих правил в действии (когда однострочная инструкция   
if используется для прерывания выполнения цикла), давайте перейдем к сле-  
дующему разделу, где мы напишем и опробуем настоящий программный код.  
Короткий пример: интерактивные циклы  
Мы увидим все эти синтаксические правила в действии, когда будем совершать   
турне по конкретным составным инструкциям языка Python в нескольких сле-  
дующих главах, но эти правила работают одинаково везде в языке Python. Для   
начала рассмотрим короткий практический пример, который продемонстри-  
рует способ применения синтаксических правил к оформлению инструкций   
и вложенных блоков на практике и попутно познакомит вас с некоторыми ин-  
струкциями.  
Простой интерактивный цикл  
Предположим, что от нас требуется написать программу на языке Python, ко-  
торая взаимодействует с пользователем в окне консоли. Возможно, программа   
будет принимать информацию для дальнейшей передачи в базу данных или   
числа для выполнения расчетов. Независимо от конечной цели, вам необходи-  
мо написать цикл, который будет считывать одну или более строк, введенных   
пользователем с клавиатуры, и выводить их обратно на экран. Другими слова-  
ми, вам нужно написать классический цикл, выполняющий операции чтения/  
вычисления/вывода.

Короткий пример: интерактивные циклы   
335  
В языке Python для реализации таких интерактивных циклов используется   
типичный шаблон, который выглядит, как показано ниже:  
while True:  
 reply = input(‘Enter text:’)  
 if reply == ‘stop’: break  
 print(reply.upper())  
В этом фрагменте использованы несколько новых для вас идей:  
 •  
Программный код выполняется в цикле while, который в языке Python   
является наиболее универсальной инструкцией цикла. Подробнее об ин-  
струкции while мы будем говорить позднее, но в двух словах замечу, что она   
состоит из слова while, за которым следует условное выражение. Результат   
этого выражения интерпретируется как истина или как ложь. Далее следу-  
ет вложенный блок программного кода, который продолжает выполняться   
в цикле, пока условное выражение возвращает истину (слово True здесь все-  
гда возвращает значение истины).  
 •  
Встроенная функция input, с которой мы уже встречались ранее в этой кни-  
ге, используется здесь как универсальное средство получения ввода с кла-  
виатуры – она выводит подсказку, текст которой содержится в необязатель-  
ном строковом аргументе, и возвращает введенный пользователем ответ   
в виде строки.  
 •  
Однострочная инструкция if, которая оформлена в соответствии со специ-  
альным правилом для вложенных блоков: тело инструкции if располагает-  
ся в той же строке, что и основная инструкция, после символа двоеточия,   
а не на отдельной строке под ней с соответствующим отступом. Она одина-  
ково хорошо работала бы при любом из двух вариантов оформления, но при   
таком подходе нам удалось сэкономить одну строку.  
 •  
Наконец, для немедленного выхода из цикла использована инструкция   
break – она просто выполняет выход за пределы инструкции цикла и про-  
грамма продолжает свою работу с первой инструкции, которая расположе-  
на вслед за циклом. Без этой инструкции цикл while работал бы вечно, по-  
скольку его условное выражение всегда возвращает истину.  
В результате такая комбинация инструкций означает следующее: «читать   
строки, введенные пользователем,4 и выводить их после преобразования всех   
символов в верхний регистр, пока пользователь не введет строку “stop”». Су-  
stop”». Су-  
”». Су-  
ществуют и другие способы записи такого цикла, но данная форма очень часто   
встречается в программах на языке Python.  
Обратите внимание, что все три строки, вложенные в инструкцию цикла while,   
имеют одинаковые отступы, благодаря этому они визуально образуют верти-  
кальную линию блока программного кода, ассоциированного с инструкцией   
цикла while. Тело цикла завершается либо с концом файла, либо с первой ин-  
струкцией, имеющей меньший отступ.  
Запустив этот фрагмент, мы получили следующий порядок взаимодействий   
с ним:  
Enter text:spam  
SPAM  
Enter text:42  
42  
Enter text:stop

336   
Глава 10. Введение в инструкции языка Python   
Примечание, касающееся различий между версиями: Этот при-  
мер написан для работы под управлением интерпретатора вер-  
сии 3.0. Если вы используете Python 2.6 или более раннюю вер-  
сию, этот пример будет работать точно так же, но вместо функ-  
ции input вы должны использовать функцию raw\_input и можете   
опустить круглые скобки в инструкции print. В версии 3.0 функ-  
ция raw\_input была переименована, а инструкция print превра-  
тилась во встроенную функцию (подробнее об инструкции print   
рассказывается в следующей главе).  
Математическая обработка данных пользователя  
Итак, наш сценарий работает, а теперь предположим, что вместо преобразова-  
ния символов текстовой строки в верхний регистр нам необходимо выполнить   
некоторые математические действия над введенными пользователем числа-  
ми, – например, вычислить квадраты чисел, что может получиться не совсем   
так, как ожидают наиболее молодые и нетерпеливые пользователи. Для дости-  
жения желаемого эффекта мы могли бы попробовать использовать следующие   
инструкции:  
>>> reply = ‘20’  
>>> reply \*\* 2  
...текст сообщения об ошибке опущен...  
TypeError: unsupported operand type(s) for \*\* or pow(): ‘str’ and ‘int’  
Этот прием не работает, потому что (как обсуждалось в предыдущей части кни-  
ги) интерпретатор выполняет преобразование типов объектов в выражениях,   
только если они являются числами, а ввод пользователя всегда передается сце-  
нарию в виде строки. Мы не можем возвести строку цифр в степень, не преоб-  
разовав ее в целое число вручную:  
>>> int(reply) \*\* 2  
400  
Вооружившись этой информацией, мы можем переделать наш цикл для вы-  
полнения математических действий:  
while True:  
 reply = input(‘Enter text:’)  
 if reply == ‘stop’: break  
 print(int(reply) \*\* 2)  
print(‘Bye’)  
В этом сценарии используется однострочная инструкция if, которая, как   
и прежде, производит выход из цикла по получении от пользователя строки   
“stop”, а кроме того, выполняется преобразование введенной строки для вы-  
stop”, а кроме того, выполняется преобразование введенной строки для вы-  
”, а кроме того, выполняется преобразование введенной строки для вы-  
полнения необходимой математической операции. В данную версию сценария   
также было добавлено сообщение, которое выводится в момент завершения   
работы сценария. Поскольку инструкция print в последней строке не имеет та-  
кого же отступа, как инструкции вложенного блока, она не считается частью   
тела цикла и будет выполняться только один раз – после выхода из цикла:  
Enter text:2  
4

Короткий пример: интерактивные циклы   
337  
Enter text:40  
1600  
Enter text:stop  
Bye  
Небольшое примечание: я предполагаю, что этот фрагмент был сохранен   
в файле сценария и запускается из него. Если вы вводите этот программный   
код в интерактивном сеансе, не забудьте добавить пустую строку (то есть на-  
жать клавишу Enter дважды) перед заключительной инструкцией print, чтобы   
завершить тело цикла. Впрочем, нет никакого смысла вводить последнюю ин-  
струкцию print в интерактивном сеансе (вы будете вводить ее уже после того,   
как завершите ввод чисел в цикле!).  
Обработка ошибок проверкой ввода  
Пока все идет хорошо, но посмотрите, что произойдет, если пользователь вве-  
дет неверную строку:  
Enter text:xxx  
...текст сообщения об ошибке опущен...  
ValueError: invalid literal for int() with base 10: ‘xxx’  
Встроенная функция int возбуждает исключение, когда сталкивается с ошиб-  
кой. Если нам необходимо обеспечить устойчивость сценария, мы должны   
предварительно проверить содержимое строки с помощью строкового метода   
isdigit:  
>>> S = ‘123’  
>>> T = ‘xxx’  
>>> S.isdigit(), T.isdigit()  
(True, False)  
Для этого в наш пример необходимо добавить вложенные операторы. В сле-  
дующей версии нашего интерактивного сценария используется полная версия   
условной инструкции if, с помощью которой предотвращается возможность   
появления исключений:  
while True:  
 reply = input(‘Enter text:’)  
 if reply == ‘stop’:  
 break  
 elif not reply.isdigit( ):  
 print(‘Bad!’ \* 8)  
 else:  
 print(int(reply) \*\* 2)  
print ‘Bye’  
Более подробно мы будем рассматривать инструкцию if в главе 12. Это очень   
простой инструмент программирования логики выполнения сценария. В своей   
полной форме инструкция содержит слово if, за которым следуют выражение   
проверки условия и вложенный блок кода, один или более необязательных   
проверок elif (“else if”) и соответствующих им вложенных блоков кода и не-  
обязательная часть else со связанным с ней блоком кода, который выполняется   
при несоблюдении условия. Интерпретатор выполняет первый блок кода, для   
которого проверка дает в результате истину, проходя инструкцию сверху вниз,   
либо часть else, если все проверки дали в результате ложь.

338   
Глава 10. Введение в инструкции языка Python   
Части if, elif и else в предыдущем примере принадлежат одной и той же ин-  
струкции, так как вертикально они расположены на одной линии (то есть име-  
ют одинаковые отступы). Инструкция if простирается до начала инструкции   
print в последней строке. В свою очередь, весь блок инструкции if является   
частью цикла while, потому что вся она смещена вправо относительно основной   
инструкции цикла. Вложение инструкций станет для вас естественным, как   
только вы приобретете соответствующие навыки.  
Теперь новый сценарий будет обнаруживать ошибки прежде, чем они будут об-  
наружены интерпретатором, и выводить (возможно, глупое) сообщение:  
Enter text:5  
25  
Enter text:xyz  
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!  
Enter text:10  
100  
Enter text:stop  
Обработка ошибок с помощью инструкции try  
Предыдущее решение работает, но, как будет показано далее в книге, в язы-  
ке Python существует более универсальный способ, который состоит в том,   
чтобы перехватывать и обрабатывать ошибки с помощью инструкции try. Эта   
инструкция подробно будет рассматриваться в последней части книги, но, ис-  
пользовав инструкцию try в качестве предварительного знакомства, мы можем   
упростить программный код предыдущей версии сценария:  
while True:  
 reply = input(‘Enter text:’)  
 if reply == ‘stop’: break  
 try:  
 num = int(reply)  
 except:  
 print(‘Bad!’ \* 8)  
 else:  
 print(int(reply) \*\* 2)  
print ‘Bye’  
Эта версия работает точно так же, как и предыдущая, только здесь мы замени-  
ли явную проверку наличия ошибки программным кодом, который предпола-  
гает, что преобразование будет выполнено и выполняет обработку исключения,   
если такое преобразование невозможно. Эта инструкция try состоит из слова   
try, вслед за которым следует основной блок кода (действие, которые мы пыта-  
емся выполнить), с последующей частью except, где располагается программ-  
ный код обработки исключения. Далее следует часть else, программный код   
которой выполняется, если в части try исключение не возникло. Интерпрета-  
тор сначала выполняет часть try, затем выполняет либо часть except (если воз-  
никло исключение), либо часть else (если исключение не возникло).  
В терминах вложенности инструкций, так как слова try, except и else имеют   
одинаковые отступы, все они считаются частью одной и той же инструкции   
try. Обратите внимание, что в данном случае часть else связана с инструкцией   
try, а не с условной инструкцией if. Как будет показано далее в книге, ключе-

Короткий пример: интерактивные циклы   
339  
вое слово else в языке Python может появляться не только в инструкции if,   
но и в инструкции try и в циклах – величина отступа наглядно показывает,   
частью какой инструкции оно является. В данном случае инструкция try на-  
чинается со слова try и продолжается до конца вложенного блока кода, сле-  
дующего за словом else, потому что else располагается на том же расстоянии   
от левого края, что и try. Инструкция if в этом примере занимает всего одну   
строку и завершается сразу же за словом break.  
Напомню, что к инструкции try мы еще вернемся далее в этой книге. А пока   
вам достаточно будет знать, что эта инструкция может использоваться для   
перехвата любых ошибок, уменьшения объема программного кода, проверяю-  
щего наличие ошибок, и представляет собой достаточно универсальный под-  
ход к обработке необычных ситуаций. Если бы нам, к примеру, потребовалось   
обеспечить поддержку ввода не целых, а вещественных чисел, использование   
инструкции try существенно упростило бы реализацию по сравнению с про-  
веркой – нам достаточно было бы просто вызвать функцию float и перехватить   
исключение вместо того, чтобы пытаться анализировать все возможные спосо-  
бы записи вещественных чисел.  
Три уровня вложенности программного кода  
Теперь рассмотрим последнюю версию сценария. В случае необходимости мы   
могли бы создать еще один уровень вложенности, например, чтобы выполнить   
проверку правильности ввода, основываясь на величине введенного числа:  
while True:  
 reply = input(‘Enter text:’)  
 if reply == ‘stop’:  
 break  
 elif not reply.isdigit():  
 print(‘Bad!’ \* 8)  
 else:  
 num = int(reply)  
 if num < 20:  
 print(‘low’)  
 else:  
 print(num \*\* 2)  
print(‘Bye’)  
В эту версию сценария добавлена еще одна инструкция if, вложенная в выра-  
жение else другой условной инструкции if, которая в свою очередь вложена   
в цикл while. Когда код выполняется по некоторому условию или в цикле, как   
в данном случае, достаточно просто выполнить дополнительный отступ впра-  
во. В результате сценарий работает так же, как и предыдущая версия, но для   
чисел меньше 20 выводит слово «low» (низкое значение):  
Enter text:19  
low  
Enter text:20  
400  
Enter text:spam  
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!  
Enter text:stop  
Bye

340   
Глава 10. Введение в инструкции языка Python   
В заключение  
В этой главе мы коротко познакомились с синтаксисом инструкций языка Py-  
Py-  
thon. Здесь были представлены основные правила записи инструкций и блоков   
программного кода. Как было сказано, в языке Python обычно в каждой строке   
программы располагается единственная инструкция и все инструкции в одном   
и том же блоке имеют одинаковые отступы (отступы являются частью синтак-  
сиса языка Python). Кроме того, мы рассмотрели несколько исключений из   
этих правил, включая многострочные инструкции, однострочные условные   
инструкции и циклы. Наконец мы воплотили эти идеи в интерактивный сце-  
нарий, в котором продемонстрировали ряд полезных инструкций, а также рас-  
смотрели синтаксис инструкций в действии.  
В следующей главе мы приступим к глубокому изучению основных процедур-  
ных инструкций языка Python. Как будет показано далее, все инструкции сле-  
Python. Как будет показано далее, все инструкции сле-  
. Как будет показано далее, все инструкции сле-  
дуют тем же основным правилам, которые были представлены здесь.  
Закрепление пройденного  
Контрольные вопросы  
1. Какие три синтаксических элемента, обязательные в языках, подобных   
языку C, опущены в языке Python?  
2. Каким образом обычно завершаются инструкции в языке Python?  
3. Как обычно определяется принадлежность инструкций к вложенному бло-  
ку в языке Python?  
4. Как можно разместить одну инструкцию в нескольких строках?  
5. Как можно разместить составную инструкцию в одной строке?  
6. Существуют ли какие-либо объективные причины для завершения ин-  
струкций точкой с запятой?  
7. Для чего предназначена инструкция try?  
8. Какую наиболее распространенную ошибку допускают начинающие про-  
граммисты на языке Python?  
Ответы  
1. Обязательными синтаксическими элементами в С-подобных языках про-  
граммирования являются круглые скобки, окружающие выражения про-  
верки условий в некоторых инструкциях, точка с запятой, завершающая   
каждую инструкцию, и фигурные скобки, окружающие вложенные блоки   
программного кода.  
2. Конец строки является концом инструкции, расположенной в этой строке.   
Если в одной строке располагается несколько инструкций, они должны за-  
вершаться точками с запятой. Если инструкция располагается в несколь-  
ких строках, она должна завершаться закрывающей скобкой.  
3. Все инструкции во вложенном блоке имеют одинаковые отступы, состоя-  
щие из одинакового числа символов пробела или табуляции.  
4. Инструкция может располагаться в нескольких строках, если заключить   
ее в пару круглых, квадратных или фигурных скобок. Инструкция закан-

Закрепление пройденного   
341  
чивается в строке, где интерпретатор Python встречает закрывающую пар-  
ную скобку.  
5. Тело составной инструкции может располагаться в той же строке, что   
и основная инструкция, сразу после символа двоеточия, но при условии,   
что тело не содержит других составных инструкций.  
6. Только когда возникает потребность разместить несколько инструкций   
в одной строке, при условии, что ни одна из инструкций не является состав-  
ной. Однако такой способ размещения инструкций приводит к снижению   
удобочитаемости программного кода.  
7. Инструкция try используется для перехвата и обработки исключений (оши-  
бок), возникающих в сценариях на языке Python. Как правило, она пред-  
ставляет альтернативу программному коду, который создается для выявле-  
ния ошибочных ситуаций.  
8. Наиболее распространенная ошибка среди начинающих программистов со-  
стоит в том, что они забывают добавлять двоеточие в конце основной части   
составных инструкций. Если вы еще не совершали такой ошибки, значит,   
скоро столкнетесь с ней!

Глава 11  
.  
   
Присваивание, выражения и print  
Теперь, когда мы кратко ознакомились с синтаксисом инструкций в языке   
Python, начиная с этой главы, мы приступим к более подробному изучению   
конкретных инструкций языка Python. Сначала мы рассмотрим самые осно-  
Python. Сначала мы рассмотрим самые осно-  
. Сначала мы рассмотрим самые осно-  
вы – инструкции присваивания, инструкции выражений и операции вывода   
на экран. Мы уже видели все эти инструкции в действии, но здесь мы воспол-  
ним подробности, которые опускались ранее. Несмотря на всю простоту этих   
инструкций, в чем вы могли убедиться ранее, каждая из них имеет дополни-  
тельные возможности, которые пригодятся, когда вы начнете писать настоя-  
щие программы на языке Python.   
Инструкции присваивания  
Ранее мы уже использовали инструкции присваивания для назначения имен   
объектам. В канонической форме цель инструкции присваивания записыва-  
ется слева от знака равно, а объект, который присваивается, – справа. Цель   
слева может быть именем или компонентом объекта, а объектом справа может   
быть произвольное выражение, которое в результате дает объект. В большин-  
стве случаев присваивание выполняется достаточно просто, однако оно облада-  
ет следующими особенностями, которые вам следует иметь в виду:  
 •  
Инструкция присваивания создает ссылку на объект. Как говорилось   
в главе 6, в языке Python инструкция присваивания сохраняет ссылки на   
объекты в переменных или в элементах структур данных. Они всегда соз-  
дают ссылки на объекты и никогда не создают копии объектов. Вследствие   
этого переменные в языке Python больше напоминают указатели, чем об-  
Python больше напоминают указатели, чем об-  
 больше напоминают указатели, чем об-  
ласти хранения данных.  
 •  
Переменные создаются при первом присваивании. Интерпретатор Python   
создает переменные, когда им впервые присваиваются значения (то есть   
ссылки на объекты), благодаря этому отсутствует необходимость пред-  
варительного объявления переменных. Иногда (но не всегда) в результате   
операции присваивания создаются элементы структур данных (например,   
элементы в словарях, некоторые атрибуты объектов). После выполнения   
операции присваивания всякий раз, когда имя переменной будет встречено

Инструкции присваивания   
343  
в выражении, оно замещается объектом, на который ссылается эта пере-  
менная.  
 •  
Прежде чем переменную можно будет использовать, ей должно быть при-  
своено значение. Нельзя использовать переменную, которой еще не было   
присвоено значение. В этом случае интерпретатор возбуждает исключение   
вместо того, чтобы вернуть какое-либо значение по умолчанию, – если бы   
он возвращал значение по умолчанию, это только осложнило бы поиск опе-  
чаток в программном коде.  
 •  
Некоторые инструкции неявно тоже выполняют операцию присваивания.   
В этом разделе мы сосредоточим все свое внимание на инструкции =, однако   
в языке Python присваивание может выполняться в самых разных контек-  
Python присваивание может выполняться в самых разных контек-  
 присваивание может выполняться в самых разных контек-  
стах. Например, далее мы увидим, что импорт модуля, определение функ-  
ции или класса, указание переменной в цикле for и передача аргументов   
функции неявно выполняют присваивание. Операция присваивания вы-  
полняется одинаково в любом месте, где бы она ни происходила, поэтому   
во всех этих контекстах просто выполняется связывание имен со ссылками   
на объекты.  
Формы инструкции присваивания  
Несмотря на то что в языке Python присваивание является универсальным   
и повсеместным понятием, в этой главе мы впервые сосредоточимся на ин-  
струкциях присваивания. В табл. 11.1 приводятся различные формы инструк-  
ции присваивания, которые встречаются в языке Python.  
Таблица 11.1. Формы инструкции присваивания  
Операция  
Интерпретация  
spam = ‘Spam’  
Каноническая форма  
spam, ham = ‘yum’, ‘YUM’  
Присваивание кортежей (позиционное)  
[spam, ham] = [‘yum’,   
‘YUM’]  
Присваивание списков (позиционное)  
a, b, c, d = ‘spam’  
Присваивание последовательностей, обобщенное  
a, \*b = ‘spam’  
Расширенная операция распаковывания последо-  
вательностей (Python 3.0)  
spam = ham = ‘lunch’  
Групповое присваивание одного значения   
spams += 42  
Комбинированная инструкция присваивания   
(эквивалентно инструкции spams = spams + 42)  
Первая форма из табл. 11.1 является наиболее распространенной: она связы-  
вает переменную (или элемент структуры данных) с единственным объектом.   
Другие формы в таблице имеют особое назначение и являются необязательны-  
ми, но многие программисты находят очень удобными:  
Присваивание кортежей и списков  
Вторая и третья формы в таблице являются родственными. Когда слева   
от оператора = указывается кортеж или список, интерпретатор связывает   
объекты справа с именами слева, согласно их местоположениям, выполняя

344   
Глава 11. Присваивание, выражения и print   
присваивание слева направо. Например, во второй строке табл. 11.1 с име-  
нем spam ассоциируется строка ‘yum’, а с именем ham ассоциируется строка   
‘YUM’. Внутри интерпретатор Python сначала создает элементы кортежа   
справа, поэтому часто эта операция называется распаковыванием кортежа.  
Присваивание последовательностей  
В недавних версиях Python операции присваивания кортежей и списков   
были обобщены в то, что теперь называется операцией присваивания  по-  
следовательностей, – любая последовательность имен может быть связана   
с любой последовательностью значений, и интерпретатор свяжет элемен-  
ты согласно их позициям. Фактически в последовательностях мы можем   
смешивать разные типы. Инструкция присваивания в четвертой строке   
табл. 11.1, например, связывает кортеж имен со строкой символов: имени   
a присваивается символ ‘s’, имени b присваивается символ ‘p’ и так далее.  
Расширенное распаковывание последовательностей  
В Python 3.0 появилась новая форма инструкции присваивания, позволя-  
Python 3.0 появилась новая форма инструкции присваивания, позволя-  
 3.0 появилась новая форма инструкции присваивания, позволя-  
ющая более гибко выбирать присваиваемые фрагменты последовательно-  
стей. В пятой строке табл. 11.1, например, переменной a будет присвоен пер-  
вый символ из строки справа, а переменной b – оставшаяся часть строки, то   
есть переменной a будет присвоено значение ‘s’, а переменной b – значение   
‘pam’. В результате мы получаем простую альтернативу операции извлече-  
ния срезов.  
Групповое присваивание одного значения  
Шестая строка в табл. 11.1 демонстрирует форму группового присваива-  
ния. В этой форме интерпретатор присваивает ссылку на один и тот же объ-  
ект (самый правый объект) всем целям, расположенным слева. Инструк-  
ция в таблице присвоит обоим именам spam и ham ссылку на один и тот же   
объект, строку ‘lunch’. Результат будет тот же, как если бы были выпол-  
нены две инструкции: ham = ‘lunch’ и spam = ham, поскольку здесь ham интер-  
претируется как оригинальный объект-строка (то есть не отдельная копия   
этого объекта).  
Комбинированное присваивание  
Последняя строка в табл. 11.1 – это пример комбинированной инструкции   
присваивания – краткая форма, которая объединяет в себе выражение   
и присваивание. Например, инструкция spam += 42 даст тот же результат,   
что и инструкция spam = spam + 42� единственное отличие состоит в том, что   
комбинированная форма имеет более компактный вид и обычно выполня-  
ется быстрее. Кроме того, если объект справа относится к категории изме-  
няемых объектов и поддерживает указанную операцию, комбинированная   
инструкция присваивания может выполняться даже быстрее, за счет непо-  
средственного изменения объекта вместо создания и изменения его копии.   
Для каждого двухместного оператора в языке Python существует своя ком-  
Python существует своя ком-  
 существует своя ком-  
бинированная инструкция присваивания.  
Присваивание последовательностей  
В этой книге мы уже использовали инструкцию присваивания в канонической   
форме. Ниже приводится несколько примеров инструкций присваивания по-  
следовательностей в действии:

Инструкции присваивания   
345  
% python  
>>> nudge = 1  
>>> wink = 2  
>>> A, B = nudge, wink # Присваивание кортежей  
>>> A, B # Что равносильно A = nudge; B = wink  
(1, 2)  
>>> [C, D] = [nudge, wink] # Присваивание списков  
>>> C, D  
(1, 2)  
Обратите внимание: в третьей инструкции этого примера в действительности   
присутствует два кортежа, просто мы опустили охватывающие их круглые   
скобки. Интерпретатор Python сопоставляет значения элементов кортежа   
справа от оператора присваивания с переменными в кортеже слева и выполня-  
ет присваивание значений в одной инструкции.  
Операция присваивания кортежей дает возможность использовать прием, ко-  
торый представлен в упражнениях ко второй части книги. Так как в процессе   
выполнения инструкции интерпретатор создает временный кортеж, где сохра-  
няются оригинальные значения переменных справа, данная форма присваива-  
ния может использоваться для реализации обмена значений переменных без   
создания временной переменной – кортеж справа автоматически запоминает   
предыдущие значения переменных:  
>>> nudge = 1  
>>> wink = 2  
>>> nudge, wink = wink, nudge # Кортежи: обмен значениями  
>>> nudge, wink # То же, что и T = nudge; nudge = wink; wink = T  
(2, 1)  
В конечном итоге формы присваивания кортежей и списков были обобщены,   
чтобы обеспечить возможность указывать любые типы последовательностей   
справа при условии, что они будут иметь ту же длину. Допускается присваи-  
вать кортеж значений списку переменных, строки символов – кортежу пере-  
менных и так далее. В любом случае интерпретатор свяжет элементы после-  
довательности справа с переменными в последовательности слева согласно их   
позициям в направлении слева направо:  
>>> [a, b, c] = (1, 2, 3) # Кортеж значений присваивается списку переменных  
>>> a, c  
(1, 3)  
>>> (a, b, c) = “ABC” # Строка символов присваивается кортежу переменных  
>>> a, c  
(‘A’, ‘C’)  
С технической точки зрения в правой части инструкции присваивания после-  
довательностей допускается указывать не только последовательности, но и лю-  
бые объекты, обеспечивающие возможность итераций по элементам. Эту, еще   
более общую концепцию, мы будем рассматривать в главах 14 и 20.  
Дополнительные варианты инструкции присваивания   
последовательностей  
Даже при том, что допускается смешивать разные типы последовательностей   
по обе стороны оператора =, обе последовательности должны иметь одно и то   
же число элементов, в противном случае мы получим сообщение об ошибке.

346   
Глава 11. Присваивание, выражения и print   
В Python 3.0 допускается использовать еще более обобщенную форму присваи-  
Python 3.0 допускается использовать еще более обобщенную форму присваи-  
 3.0 допускается использовать еще более обобщенную форму присваи-  
вания, применяя расширенный синтаксис распаковывания последовательно-  
стей, который описывается в следующем разделе. Но, как правило (и всегда   
в Python 2.X), количество переменных слева должно соответствовать количе-  
Python 2.X), количество переменных слева должно соответствовать количе-  
 2.X), количество переменных слева должно соответствовать количе-  
X), количество переменных слева должно соответствовать количе-  
), количество переменных слева должно соответствовать количе-  
ству элементов последовательности справа:  
>>> string = ‘SPAM’  
>>> a, b, c, d = string # Одинаковое число элементов с обеих сторон  
>>> a, d  
(‘S’, ‘M’)  
   
>>> a, b, c = string # В противном случае выводится сообщение об ошибке  
...текст сообщения об ошибке опущен...  
ValueError: too many values to unpack  
В общем случае нам необходимо получить срез. Существует несколько вариан-  
тов извлечения среза, чтобы исправить дело:  
>>> a, b, c = string[0], string[1], string[2:] # Элементы и срез  
>>> a, b, c  
(‘S’, ‘P’, ‘AM’)  
   
>>> a, b, c = list(string[:2]) + [string[2:]] # Срезы и конкатенация  
>>> a, b, c  
(‘S’, ‘P’, ‘AM’)  
   
>>> a, b = string[:2] # То же самое, только проще  
>>> c = string[2:]  
>>> a, b, c  
(‘S’, ‘P’, ‘AM’)  
   
>>> (a, b), c = string[:2], string[2:] # Вложенные последовательности  
>>> a, b, c  
(‘S’, ‘P’, ‘AM’)  
Последний пример демонстрирует, что мы можем присваивать даже вложен-  
ные последовательности, и интерпретатор распаковывает их части в соответ-  
ствии с их представлением, как и ожидается. В данном случае выполняется   
присваивание кортежа из двух элементов, где первый элемент – это вложенная   
последовательность (строка), что точно соответствует следующему случаю:  
>>> ((a, b), c) = (‘SP’, ‘AM’) # Связывание производится в соответствии   
>>> a, b, c # с формой и местоположением  
(‘S’, ‘P’, ‘AM’)  
Интерпретатор связывает первую строку справа (‘SP’) с первым кортежем слева   
((a, b)), присваивая каждому его элементу по одному символу, а затем выпол-  
няет присваивание второй строки целиком (‘AM’) переменной c. В этом случае   
вложенная последовательность слева, имеющая форму объекта, должна соот-  
ветствовать объекту справа. Присваивание вложенных последовательностей –   
это достаточно сложная операция, которая редко встречается на практике, но   
такой способ присваивания может оказаться удобным для присваивания части   
структуры данных известной формы.   
Например, как будет показано в главе 13, этот прием можно использовать в ци-  
клах for для присваивания элементов итерируемой последовательности не-  
скольким переменным, указанным в инструкции цикла:

Инструкции присваивания   
347  
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: ... # Простое присваивание   
 # кортежей  
for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: ... # Присваивание вложенных   
 # кортежей  
В главе 18 мы увидим, что форма присваивания вложенных кортежей (в дей-  
ствительности – последовательностей) может также использоваться в списках   
аргументов функций в Python 2.6 (но не в Python 3.0), потому что передача ар-  
Python 2.6 (но не в Python 3.0), потому что передача ар-  
 2.6 (но не в Python 3.0), потому что передача ар-  
гументов выполняется присваиванием для распаковывания списков аргумен-  
тов функций:  
def f(((a, b), c)): # Для распаковывания аргументов в Python 2.6, но не в 3.0  
f(((1, 2), 3))  
Кроме того, операция присваивания последовательности с распаковыванием   
дает начало еще одному распространенному обороту программирования на   
языке Python – присваиванию последовательности целых чисел множеству   
переменных:  
>>> red, green, blue = range(3)  
>>> red, blue  
(0, 2)  
В этом примере три переменные инициализируются целочисленными значе-  
ниями 0, 1 и 2 соответственно (это эквивалент перечислимых типов данных   
в языке Python, которые, возможно, вам приходилось встречать в других язы-  
Python, которые, возможно, вам приходилось встречать в других язы-  
, которые, возможно, вам приходилось встречать в других язы-  
ках программирования). Чтобы понять происходящее, вы должны знать, что   
встроенная функция range генерирует непрерывный список последовательных   
целых чисел:  
>>> range(3) # Используйте list(range(3)) в Python 3.0  
[0, 1, 2]  
Поскольку функция range часто используется в циклах for, мы еще поговорим   
о ней в главе 13.   
Другой случай применения операции присваивания кортежей – разделение   
последовательности на начальную и остальную части в циклах, как показано   
ниже:  
>>> L = [1, 2, 3, 4]  
>>> while L:  
... front, L = L[0], L[1:] # Вариант для 3.0 приводится в след. разделе  
... print(front, L)  
...  
1 [2, 3, 4]  
2 [3, 4]  
3 [4]  
4 []  
Присваивание кортежа в цикле здесь можно было бы заменить двумя следую-  
щими строками, но часто бывает удобнее объединить их в одну строку:  
... front = L[0]  
... L = L[1:]  
Обратите внимание: в этом примере список используется в качестве стека –   
структуры данных, поведение которой реализуют методы списков append и pop.

348   
Глава 11. Присваивание, выражения и print   
В данном случае эффект, который дает операция присваивания кортежа, мож-  
но было бы получить инструкцией front = L.pop(0), но это будет операция непо-  
средственного изменения объекта. О циклах while и о других (часто лучших)   
способах обхода последовательностей с помощью циклов for вы узнаете больше   
в главе 13.  
Расширенная операция распаковывания   
последовательностей в Python 3.0  
В предыдущем примере демонстрировалось, как вручную организовать извле-  
чение срезов, чтобы сделать инструкцию присваивания последовательностей   
более универсальной. В Python 3.0 (но не в 2.6) инструкция присваивания по-  
Python 3.0 (но не в 2.6) инструкция присваивания по-  
 3.0 (но не в 2.6) инструкция присваивания по-  
следовательностей была обобщена еще больше, что еще больше упростило ее   
использование. В двух словах: чтобы описать более общий случай присваи-  
вания, слева от оператора присваивания допускается указывать одно имя со   
звездочкой, например \*X, – имени со звездочкой будет присвоен список всех   
элементов последовательности, не присвоенных другим переменным слева. Это   
особенно удобно для реализации таких распространенных операций, как раз-  
биение последовательности на «начало» и «остаток», как было показано в по-  
следнем примере предыдущего раздела.  
Расширенная операция распаковывания в действии  
Рассмотрим пример. Как мы уже знаем, в операции распаковывания последо-  
 пример. Как мы уже знаем, в операции распаковывания последо-  
пример. Как мы уже знаем, в операции распаковывания последо-  
. Как мы уже знаем, в операции распаковывания последо-  
Как мы уже знаем, в операции распаковывания последо-  
вательностей количество имен слева от оператора присваивания должно точно   
соответствовать количеству элементов в последовательности справа. При несо-  
блюдении этого правила мы будем получать сообщение об ошибке (если вруч-  
ную не предусмотрим извлечение срезов из последовательности справа, как   
было показано в предыдущем разделе):  
C:\misc> c:\python30\python  
>>> seq = [1, 2, 3, 4]  
>>> a, b, c, d = seq  
>>> print(a, b, c, d)  
1 2 3 4  
>>> a, b = seq  
ValueError: too many values to unpack  
Однако в Python 3.0 в списке переменных слева можно указать одно имя со   
звездочкой, чтобы ослабить правило соответствия. В представленном ниже   
продолжении предыдущего интерактивного сеанса переменной a присваивает-  
ся первый элемент последовательности, а переменной b – все остальные:  
>>> a, \*b = seq  
>>> a  
1  
>>> b  
[2, 3, 4]  
Когда в левой части инструкции присутствует имя со звездочкой, количество   
переменных в левой части не обязательно должно соответствовать количеству   
элементов в последовательности справа. Фактически имя со звездочкой может   
указываться в любой позиции слева. Например, в примере ниже переменной   
b соответствует последний элемент последовательности, а переменной a – все   
элементы, предшествующие последнему:

Инструкции присваивания   
349  
>>> \*a, b = seq  
>>> a  
[1, 2, 3]  
>>> b  
4  
Когда имя со звездочкой указывается в середине списка переменных, ей при-  
сваиваются все элементы последовательности справа, которые остаются после   
присваивания остальным переменным без звездочек. То есть в следующем при-  
мере переменным a и c будут присвоены первый и последний элементы, а пере-  
менной b – все остальные, что находятся между ними:  
>>> a, \*b, c = seq  
>>> a  
1  
>>> b  
[2, 3]  
>>> c  
4  
В более широком смысле, в какой бы позиции ни появлялась переменная со   
звездочкой, ей будет присвоен список, содержащий все неприсвоенные элемен-  
ты, соответствующие этой позиции:  
>>> a, b, \*c = seq  
>>> a  
1  
>>> b  
2  
>>> c  
[3, 4]  
Естественно, как и обычная операция присваивания последовательностей,   
расширенная операция распаковывания последовательностей может приме-  
няться к последовательностям любых типов, не только к спискам. Ниже при-  
водится пример распаковывания символов строки:  
>>> a, \*b = ‘spam’  
>>> a, b  
(‘s’, [‘p’, ‘a’, ‘m’])  
>>> a, \*b, c = ‘spam’  
>>> a, b, c  
(‘s’, [‘p’, ‘a’], ‘m’)  
Этот прием напоминает способ, основанный на извлечении срезов, но это не   
совсем одно и то же – инструкция присваивания последовательностей всегда   
возвращает список с множеством соответствующих элементов, тогда как опе-  
рация извлечения среза возвращает последовательность того же типа, что и по-  
следовательность, из которой извлекается срез:  
>>> S = ‘spam’  
>>> S[0], S[1:] # Тип среза зависит от типа исходной последовательности,  
(‘s’, ‘pam’) # расширенная операция распаковывания всегда возвращает список  
>>> S[0], S[1:3], S[3]  
(‘s’, ‘pa’, ‘m’)  
Используя эту новую возможность, появившуюся в Python 3.0, применительно   
к спискам, мы можем еще больше упростить последний пример из предыдуще-

350   
Глава 11. Присваивание, выражения и print   
го раздела и избавиться от операций извлечения среза при получении первого   
и остального элементов:  
>>> L = [1, 2, 3, 4]  
>>> while L:  
... front, \*L = L # Получить первый и остальные элементы   
... print(front, L) # без операции извлечения среза  
...  
1 [2, 3, 4]  
2 [3, 4]  
3 [4]  
4 []  
Граничные случаи  
Расширенная операция распаковывания последовательностей обладает доста-  
точной гибкостью, тем не менее нам следует отметить некоторые граничные   
случаи. Во-первых, переменной со звездочкой может соответствовать един-  
ственный элемент, но ей всегда присваивается список:  
>>> seq  
[1, 2, 3, 4]  
>>> a, b, c, \*d = seq  
>>> print(a, b, c, d)  
1 2 3 [4]  
Во-вторых, если на долю переменной со звездочкой не остается неприсвоенных   
элементов, ей присваивается пустой список, независимо от того, в какой пози-  
ции эта переменная находится. В следующем примере каждой из переменных   
a, b, c и d соответствует свой элемент последовательности, но вместо того, чтобы   
возбудить исключение, интерпретатор присваивает переменной e пустой спи-  
сок:  
>>> a, b, c, d, \*e = seq  
>>> print(a, b, c, d, e)  
1 2 3 4 []  
   
>>> a, b, \*e, c, d = seq  
>>> print(a, b, c, d, e)  
1 2 3 4 []  
Наконец, ошибкой будет считаться, если указать несколько переменных со   
звездочкой� если значений окажется недостаточно, а слева не окажется пере-  
менной со звездочкой (как и ранее) и если переменная со звездочкой окажется   
единственной вне последовательности:  
>>> a, \*b, c, \*d = seq  
SyntaxError: two starred expressions in assignment  
   
>>> a, b = seq  
ValueError: too many values to unpack  
   
>>> \*a = seq  
SyntaxError: starred assignment target must be in a list or tuple  
   
>>> \*a, = seq  
>>> a  
[1, 2, 3, 4]

Инструкции присваивания   
351  
Полезное удобство  
Имейте в виду, что расширенная операция распаковывания последовательно-  
стей – это всего лишь удобство. Мы можем добиться того же эффекта, исполь-  
зуя явно операции индексирования и извлечения среза (и фактически эту аль-  
тернативу придется использовать в Python 2.X), но расширенная инструкция   
распаковывания выглядит компактнее. Типичный прием разбиения последо-  
вательности «первый, остаток», например, можно реализовать тем или иным   
способом, но операция извлечения среза более трудозатратна:  
>>> seq  
[1, 2, 3, 4]  
   
>>> a, \*b = seq # Первый, остаток  
>>> a, b  
(1, [2, 3, 4])  
   
>>> a, b = seq[0], seq[1:] # Первый, остаток: традиционная реализация  
>>> a, b  
(1, [2, 3, 4])  
Прием разбиения последовательности «остаток, последний» может быть реа-  
лизован похожим способом, но с применением новой расширенной инструкции   
распаковывания последовательностей требуется заметно меньшее количество   
нажатий на клавиши:  
>>> \*a, b = seq # Остаток, последний  
>>> a, b  
([1, 2, 3], 4)  
   
>>> a, b = seq[:-1], seq[-1] # Остаток, последний: традиционная реализация  
>>> a, b  
([1, 2, 3], 4)  
Расширенная инструкция распаковывания последовательностей выглядит не   
только проще, но и естественнее, поэтому со временем она наверняка получит   
широкое распространение в программах на языке Python.  
Использование в циклах for  
Поскольку переменные цикла в инструкции for также участвуют в присваи-  
вании, расширенная операция распаковывания последовательностей может   
применяться и к ним. Мы уже немного познакомились с инструкцией for во   
второй части книги и продолжим знакомство с ней в главе 13. В Python 3.0 рас-  
Python 3.0 рас-  
 3.0 рас-  
ширенная инструкция распаковывания может помещаться после слова for, где   
обычно указывается простое имя переменной:  
for (a, \*b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:  
 ...  
При таком использовании в каждой итерации интерпретатор будет просто при-  
сваивать очередной кортеж значений кортежу переменных. На первом прохо-  
де, например, будет выполнено присваивание, как если бы оно было реализова-  
но в виде выражения:  
a, \*b, c = (1, 2, 3, 4) # Переменная b получит значение [2, 3]

352   
Глава 11. Присваивание, выражения и print   
Переменные a, b и c можно использовать в теле цикла для ссылки на извлечен-  
ные компоненты. В действительности, в этом вообще нет ничего особенного –   
это лишь разновидность более общей операции присваивания. Как мы видели   
выше в этой главе, того же эффекта можно добиться с помощью простой опера-  
ции присваивания кортежей в обеих версиях Python 2.X и 3.X:  
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: # a, b, c = (1, 2, 3), ...  
И мы всегда сможем имитировать поведение расширенной инструкции распа-  
ковывания последовательностей в Python 2.6, применив операцию извлечения   
среза:  
for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:  
 a, b, c = all[0], all[1:3], all[3]  
Мы пока недостаточно подготовлены к анализу подробностей синтаксиса цик-  
ла for, поэтому мы еще вернемся к этой теме в главе 13.  
Групповое присваивание  
При групповом присваивании объект, расположенный справа, присваивается   
всем указанным переменным. В следующем примере трем переменным a, b и c   
присваивается строка ‘spam’:  
>>> a = b = c = ‘spam’  
>>> a, b, c  
(‘spam’, ‘spam’, ‘spam’)  
Эта инструкция эквивалентна (но записывается компактнее) следующим трем   
инструкциям присваивания:  
>>> c = ‘spam’  
>>> b = c  
>>> a = b  
Групповое присваивание и разделяемые ссылки  
Имейте в виду, что в этом случае существует всего один объект, разделяемый   
всеми тремя переменными (все они указывают на один и тот же объект в памя-  
ти). Такой способ присваивания хорошо подходит для неизменяемых объектов,   
например для инициализации нескольких счетчиков нулевым значением (не   
забывайте, что в языке Python переменная должна быть инициализирована,   
прежде чем к ней можно будет обратиться, поэтому вам всегда придется уста-  
навливать начальные значения в счетчиках, прежде чем они смогут использо-  
ваться для счета):  
>>> a = b = 0  
>>> b = b + 1  
>>> a, b  
(0, 1)  
Здесь изменение переменной b затронет только переменную b, потому что числа   
не допускают возможность непосредственного изменения. Если присваивае-  
мый объект является неизменяемым, совершенно не важно, как много ссылок   
на него будет создано.  
Но, как обычно, следует проявлять осторожность, выполняя присваивание   
переменным изменяемых объектов, таких как списки или словари:

Инструкции присваивания   
353  
>>> a = b = []  
>>> b.append(42)  
>>> a, b  
([42], [42])  
На этот раз, поскольку a и b ссылаются на один и тот же объект, непосредствен-  
ное добавление значения к объекту через переменную b будет воздействовать   
и на переменную a. В действительности это всего лишь другой пример взаи-  
мовлияния разделяемых ссылок, с которым мы впервые встретились в главе 6.   
Чтобы избежать этой проблемы, инициализацию изменяемыми объектами   
следует производить в отдельных инструкциях, чтобы в каждой из них созда-  
вался новый пустой объект с помощью отдельных литеральных выражений:  
>>> a = []  
>>> b = []  
>>> b.append(42)  
>>> a, b  
([], [42])  
Комбинированные инструкции присваивания  
Начиная с версии Python 2.0, в языке появился набор дополнительных ин-  
Python 2.0, в языке появился набор дополнительных ин-  
 2.0, в языке появился набор дополнительных ин-  
струкций присваивания, перечисленных в табл. 11.2. Известные как комби-  
нированные инструкции присваивания и заимствованные из языка C, они по   
существу являются лишь более компактной формой записи. Они комбинируют   
в себе выражение и операцию присваивания. Например, следующие две фор-  
мы записи практически эквивалентны:  
X = X + Y # Традиционная форма записи  
X += Y # Новая, комбинированная форма записи  
Таблица 11.2. Комбинированные инструкции присваивания  
X += Y  
X &= Y  
X -= Y  
X |= Y  
X \*= Y  
X ^= Y  
X /= Y  
X >>= Y  
X %= Y  
X <<= Y  
X \*\*= Y  
X //= Y  
Комбинированные операции присваивания существуют для любого поддержи-  
ваемого двухместного оператора. Например, ниже приводится два способа при-  
бавления 1 к значению переменной:  
>>> x = 1  
>>> x = x + 1 # Традиционная форма записи  
>>> x  
2  
>>> x += 1 # Комбинированная  
>>> x  
3  
Если комбинированную инструкцию применить к строкам, будет выполнена   
операция конкатенации. Таким образом, вторая строка ниже эквивалентна бо-  
лее длинной инструкции S = S + “SPAM”:  
>>> S = “spam”  
>>> S += “SPAM” # Выполняется операция конкатенации

354   
Глава 11. Присваивание, выражения и print   
>>> S  
‘spamSPAM’  
Как показано в табл. 11.2, для каждого двухместного оператора (то есть для   
оператора, слева и справа от которого располагаются значения, участвующие   
в операции) в языке Python существует своя комбинированная инструкция   
присваивания. Например, инструкция X \*= Y выполняет умножение и присваи-  
вание, X >>= Y – сдвиг вправо и присваивание, и так далее. Инструкция X //= Y   
(деление с округлением вниз) была добавлена в версии Python 2.2.  
Комбинированные инструкции присваивания обладают следующими пре иму-  
ществами:1   
 •  
Уменьшается объем ввода с клавиатуры. Нужно ли мне продолжать?  
 •  
Левая часть инструкции должна быть получена всего один раз. В инструк-  
ции «X += Y» X может оказаться сложным выражением, которое в комбини-  
рованной форме должно быть вычислено всего один раз. В более длинной   
форме записи «X = X + Y» X появляется дважды, и поэтому данное выражение   
должно быть вычислено дважды. Вследствие этого комбинированные ин-  
струкции присваивания выполняются обычно быстрее.  
 •  
Автоматически выбирается оптимальный алгоритм выполнения. Для объ-  
ектов, поддерживающих возможность непосредственного изменения, ком-  
бинированные инструкции присваивания автоматически выполняются   
непосредственно на самих объектах, вместо выполнения более медленной   
операции копирования.  
И последний момент, который требует дополнительных разъяснений. Комби-  
нированные инструкции присваивания, при применении к изменяемым объ-  
ектам, могут служить для оптимизации. Вспомним, что списки могут расши-  
ряться разными способами. Чтобы добавить в список единственный элемент,   
мы можем выполнить операцию конкатенации или вызвать метод append:  
>>> L = [1, 2]  
>>> L = L + [3] # Конкатенация: более медленная  
>>> L  
[1, 2, 3]  
>>> L.append(4) # Более быстрая, но изменяет сам объект  
>>> L  
[1, 2, 3, 4]  
А чтобы добавить несколько элементов, мы можем либо снова выполнить опе-  
рацию конкатенации, либо вызвать метод extend2:  
>>> L = L + [5, 6] # Конкатенация: более медленная  
>>> L  
[1, 2, 3, 4, 5, 6]  
>>> L.extend([7, 8]) # Более быстрая, но изменяет сам объект  
1   
Программисты C/C++, конечно, заметят, что несмотря на появление в языке Python   
таких инструкций, как X +=Y, в нем до сих пор отсутствуют операторы инкремента   
и декремента (например, X++, --X). Однако эти операторы вообще не соответствуют   
модели языка Python, в котором отсутствует возможность непосредственно изменять   
неизменяемые объекты, такие как числа.  
2   
Как предлагалось в главе 6, для этого также можно было бы использовать операцию   
присваивания срезу (например, L[len(L):] = [11,12,13]), но этот прием работает прак-  
тически так же, как метод extend.

Инструкции присваивания   
355  
>>> L  
[1, 2, 3, 4, 5, 6, 7, 8]  
В обоих случаях операция конкатенации несет в себе меньше побочных эффек-  
тов при работе с разделяемыми ссылками на объекты, но вообще она выполня-  
ется медленнее, чем эквивалентные операции, воздействующие на объект не-  
посредственно. Операция конкатенации должна создать новый объект, копию   
списка слева, и затем скопировать в него список справа. В противовес ей метод,   
воздействующий на объект непосредственно, просто добавляет новый элемент   
в конец блока памяти.  
При использовании комбинированной инструкции присваивания для расши-  
рения списка мы можем не думать об этих деталях – интерпретатор Python ав-  
Python ав-  
 ав-  
томатически вызовет более быстрый метод extend вместо использования более   
медленной операции конкатенации, которую предполагает оператор +:  
>>> L += [9, 10] # Выполняется как L.extend([9, 10])  
>>> L  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
Комбинированные инструкции присваивания   
и разделяемые ссылки  
Такой порядок вещей чаще всего именно то, что нам требуется, но необходимо   
учитывать – он предполагает, что применительно к спискам операция += вы-  
полняет изменения непосредственно в объекте, а это далеко не то же самое, что   
операция конкатенации +, в результате которой всегда создается новый объ-  
ект. Как всегда в случае использования разделяемых ссылок, различия в по-  
ведении этих операций могут иметь значение, если имеются другие ссылки на   
изменяемый объект:  
>>> L = [1, 2]  
>>> M = L # L и M ссылаются на один и тот же объект  
>>> L = L + [3, 4] # Операция конкатенации создает новый объект  
>>> L, M # Изменяется L, но не M  
([1, 2, 3, 4], [1, 2])  
   
>>> L = [1, 2]  
>>> M = L  
>>> L += [3, 4] # Операция += предполагает вызов метода extend  
>>> L, M # Значение M тоже изменилось!  
([1, 2, 3, 4], [1, 2, 3, 4])  
Все это имеет значение только для изменяемых объектов, таких как списки   
и словари, и к тому же это не совсем ясный случай (по крайней мере, пока его   
влияние не скажется на вашем программном коде!). Если вам требуется избе-  
жать создания системы разделяемых ссылок, создавайте копии изменяемых   
объектов.  
Правила именования переменных  
Теперь, когда мы исследовали инструкции присваивания, настало время более   
формально подойти к вопросу выбора имен переменных. В языке Python име-  
на появляются в момент, когда им присваиваются некоторые значения, одна-  
ко существует несколько правил, которым желательно следовать при выборе   
имен для всего сущего в ваших программах:

356   
Глава 11. Присваивание, выражения и print   
Синтаксис: (символ подчеркивания или алфавитный символ) + (любое число   
символов, цифр или символов подчеркивания)  
Имена переменных должны начинаться с символа подчеркивания или с ал-  
фавитного символа, за которым может следовать произвольное число алфа-  
витных символов, цифр или символов подчеркивания. Допустимыми име-  
нами являются: \_spam, spam и Spam\_1, а 1\_Spam, spam$ и @#! – недопустимыми.  
Регистр символов в именах имеет значение: имена SPAM и spam считаются раз-  
личными  
В языке Python регистр символов всегда принимается во внимание, как   
в именах, которые вы создаете, так и в зарезервированных словах. Напри-  
мер, имена X и x – это две разные переменные. Для обеспечения переноси-  
мости регистр символов учитывается и в именах импортируемых модулей,   
даже на платформах, где файловые системы не учитывают регистр симво-  
лов.  
Запрещено использовать зарезервированные слова  
Имена определяемых вами переменных не могут совпадать с зарезервиро-  
ванными словами, имеющими в языке Python специальное назначение. На-  
пример, если попытаться использовать переменную с именем class, интер-  
претатор выведет сообщение о синтаксической ошибке, однако имена klass   
и Class являются вполне допустимыми. В табл. 11.3 перечислены слова, ко-  
торые в настоящее время зарезервированы языком Python (и, следователь-  
но, запрещены для использования в качестве имен переменных).  
Таблица 11.3. Зарезервированные слова в версии Python 3.0  
False  
class  
finally  
is  
return  
None  
continue  
for  
lambda  
try  
True  
def  
from  
nonlocal  
while  
and  
del  
global  
not  
with  
as  
elif  
if  
or  
yield  
assert  
else  
import  
pass  
break  
except  
in  
raise  
В табл. 11.3 приводятся слова, зарезервированные в версии Python 3.0. В Py-  
Python 3.0. В Py-  
 3.0. В Py-  
Py-  
thon 2.6 набор зарезервированных слов немного отличается:  
 •  
print – это зарезервированное слово, потому что оно соответствует инструк-  
ции вывода, а не встроенной функции (подробнее об этом рассказывается   
ниже в этой главе).  
 •  
exec – это зарезервированное слово, потому что оно соответствует инструк-  
ции, а не встроенной функции.  
 •  
nonlocal – не является зарезервированным словом, потому что эта инструк-  
ция недоступна в версии 2.6.  
В более ранних версиях Python ситуация с зарезервированными словами скла-  
Python ситуация с зарезервированными словами скла-  
 ситуация с зарезервированными словами скла-  
дывалась почти так же, с небольшими отклонениями:

Инструкции присваивания   
357  
 •  
with и as не были зарезервированными словами до версии Python 2.6, в кото-  
Python 2.6, в кото-  
 2.6, в кото-  
рой впервые официально появились менеджеры контекста.  
 •  
yield не было зарезервированным словом до версии Python 2.3, в которой   
впервые появились функции-генераторы.  
 •  
В версии 2.5 инструкция yield была преобразована в выражение, но не ста-  
ла встроенной функцией, а осталась зарезервированным словом.  
В большинстве своем, зарезервированные слова языка Python записываются   
символами нижнего регистра, и они действительно зарезервированы – в отли-  
чие от имен в области видимости по умолчанию, о которой будет рассказывать-  
ся в следующей части книги, вы не можете переопределять зарезервированные   
слова посредством операции присваивания (например, and = 1 приведет к появ-  
лению ошибки).1  
Кроме того, что первые три зарезервированных слова в табл. 11.3, True, False   
и None, записываются символами разных регистров, они еще имеют не совсем   
обычное назначение – они определены в области видимости по умолчанию,   
о которой рассказывается в главе 17, и с технической точки зрения являются   
именами переменных, которым присвоены объекты. Однако во всех остальных   
смыслах они являются зарезервированными словами и не могут использовать-  
ся для других целей, кроме как представлять присвоенные им объекты. Все   
остальные зарезервированные слова являются частью синтаксиса языка Py-  
Py-  
thon и могут появляться только в определенном контексте.  
Кроме того, поскольку имена модулей в инструкции import становятся пере-  
менными в ваших сценариях, это накладывает ограничения на имена файлов   
с модулями. Вы можете создать файлы с именами and.py и my-code.py, но вы не   
сможете импортировать их, потому что их имена без расширения «.py» в про-  
граммном коде будут преобразованы в имена переменных и поэтому должны   
следовать только что обозначенным правилам (зарезервированные слова запре-  
щены, а символы дефиса являются недопустимыми, хотя можно использовать   
символы подчеркивания). Мы еще вернемся к этой теме в пятой части книги.  
Правила внесения изменений  
Будет интересно отметить, насколько постепенно и поэтапно вносятся   
изменения в перечень зарезервированных слов языка. Когда появляется   
какая-нибудь новая особенность, которая может отрицательно сказаться   
на работоспособности существующих программ, она обычно сначала вво-  
дится как необязательная, а в интерпретатор встраивается вывод преду-  
преждения об использовании «нерекомендуемой» возможности, которое   
действует на протяжении одного или более выпусков, прежде чем новая   
возможность официально будет включена в состав языка.   
1   
Тем не менее в Jython, реализации Python на языке Java, имена пользовательских   
переменных иногда могут совпадать с зарезервированными словами языка Python.   
Краткий обзор Jython приводится в главе 2.

358   
Глава 11. Присваивание, выражения и print   
Идея состоит в том, чтобы дать пользователям достаточное время, чтобы   
обратить внимание на предупреждения и внести изменения в программы   
перед переходом на использование новой версии интерпретатора. Это не   
относится к случаям, когда выходит новая версия со следующим основ-  
ным номером, такая как 3.0 (в таких случаях обратная совместимость   
с существующими программами не гарантируется без всяких предупре-  
ждений), но это справедливо для всех остальных случаев.  
Например, слово yield было необязательным расширением в версии   
Python 2.2, но, начиная с версии 2.3, оно стало стандартным заре-  
зервированным словом. Оно используется совместно с функциями-  
генераторами. Это один из небольшого числа элементов языка, в кото-  
ром была нарушена обратная совместимость с прежними версиями. При   
этом отношение к использованию слова yield менялось поэтапно – в вер-  
сии 2.2 при его использовании генерировалось предупреждение, и оно   
было недопустимым до версии 2.3.  
Похожим образом в Python 2.6 появились новые зарезервированные   
слова with и as для использования в менеджерах контекстов (новая фор-  
ма обработки исключений). В версии 2.5 эти слова не считаются заре-  
зервированными, пока менеджер контекстов не будет включен вручную   
с помощью инструкции импортирования from\_\_future\_\_ (подробности   
приводятся ниже в этой книге). При использовании в версии 2.5 слова   
with и as вызывают появление предупреждений о грядущих изменени-  
ях – исключение составляет версия IDLE в Python 2.5, где эта особен-  
ность включается автоматически (то есть использование этих слов в ка-  
честве имен переменных в версии 2.5 будет вызывать ошибку, но только   
в интерфейсе IDLE).  
Соглашения по именованию  
Помимо указанных правил существует еще целый ряд соглашений – правил,   
которые не являются обязательными, но которым обычно следуют на прак-  
тике. Например, имена с двумя символами подчеркивания в начале и в конце   
(например, \_\_name\_\_) обычно имеют особый смысл для интерпретатора, поэто-  
му вам следует избегать их использования для именования своих переменных.   
Ниже приводится список соглашений, которым было бы желательно следовать:  
 •  
Имена, начинающиеся с одного символа подчеркивания (\_X), не импортиру-  
ются инструкцией from module import \* (описывается в главе 22).  
 •  
Имена, имеющие два символа подчеркивания в начале и в конце (\_\_X\_\_),   
являются системными именами, которые имеют особый смысл для интер-  
претатора.  
 •  
Имена, начинающиеся с двух символов подчеркивания и не оканчивающи-  
еся двумя символами подчеркивания (\_\_X), являются локальными («иска-  
женными») для объемлющего класса (смотрите обсуждение псевдочастных   
атрибутов в главе 30).  
 •  
Имя, состоящее из единственного символа подчеркивания (\_), хранит ре-  
зультат последнего выражения при работе в интерактивной оболочке.

Инструкции присваивания   
359  
В дополнение к этим соглашениям существует еще ряд других соглашений,   
которым обычно стремятся следовать программисты. Например, далее в кни-  
ге мы увидим, что имена классов обычно начинаются с заглавного символа,   
а имена модулей – со строчного. Что имя self, хотя и не являющееся зарезер-  
вированным, играет особую роль в классах. В главе 17 мы познакомимся с еще   
одной крупной категорией имен, которые называются встроенными, то есть   
предопределенными, но не зарезервированными (вследствие чего они допуска-  
ют переопределение: инструкция open = 42 является вполне допустимой, но ино-  
гда вы можете пожалеть, что это так!).  
Имена не имеют типа, тип – это характеристика объектов  
По большей части это лишь краткий обзор, но вы должны помнить, что край-  
не важно сохранить четкое понимание различий между именами и объектами   
в языке Python. Как говорилось в главе 6, объекты имеют тип (например, целое   
число, список) и могут быть изменяемыми или нет. С другой стороны, имена   
(они же переменные) всегда являются всего лишь ссылками на объекты – они   
не имеют информации об изменяемости или о типе отдельно от типа объекта,   
на который они ссылаются в данный момент времени.  
Это вполне нормально, когда в разные моменты времени одно и то же имя свя-  
зывается с объектами разных типов:  
>>> x = 0 # Имя x связывается с целочисленным объектом  
>>> x = “Hello” # Теперь оно представляет строку  
>>> x = [1, 2, 3] # А теперь – список  
В последующих примерах вы увидите, что такая универсальная природа имен   
может рассматриваться как существенное преимущество при программирова-  
нии на языке Python.1 В главе 17 вы узнаете, что имена находятся внутри об-  
ласти видимости, которая определяет, где эти имена могут использоваться, –   
место, где выполняется присваивание, определяет, где это имя будет видимо.  
Дополнительные предложения, касающиеся именования пере-  
менных, приводятся в предыдущем разделе «Соглашения по   
именованию», который, по сути, повторяет положения из полу-  
официального руководства по оформлению программного кода   
на языке Python, известного как PEP 8. Это руководство доступ-  
но по адресу http://www.python.org/dev/peps/pep-0008. Его также   
можно отыскать в Сети, выполнив поиск по строке «Python   
PEP 8». С технической точки зрения этот документ формализует   
стандарты оформления исходных текстов, используемые в стан-  
дартной библиотеке языка Python.  
1   
Если вам приходилось пользоваться языком C++, возможно, вас заинтересует, что   
в отличие от C++ в языке Python отсутствует объявление const. Некоторые объек-  
ты могут быть неизменяемыми, но имена всегда допускают выполнение операции   
присваивания. Кроме того, в языке Python имеются средства сокрытия имен в клас-  
сах и модулях, но они также не являются аналогами объявлений в языке C++ (если   
вам интересна тема сокрытия атрибутов, обращайтесь к обсуждению имен модулей   
вида \_X в главе 24, имен классов вида \_\_X в главе 30, а также к примерам декораторов   
классов Private и Public в главе 38).

360   
Глава 11. Присваивание, выражения и print   
Несмотря на важность стандартов, хочется сделать несколько   
обычных предупреждений. Во-первых, документ PEP 8 содер-  
PEP 8 содер-  
 8 содер-  
жит множество тонкостей, встретиться с которыми вы, вероят-  
но, еще не готовы. Честно говоря, этот документ более сложный   
и субъективный, чем должен был бы быть, – некоторые из пред-  
ложений не являются общепринятыми и не поддерживаются   
программистами-практиками. Кроме того, некоторые из наибо-  
лее известных компаний, использующих язык Python, приняли   
собственные стандарты оформления программного кода.  
Тем не менее документ PEP 8 содержит действительно полезные   
сведения, и начинающим программистам совсем не лишним бу-  
дет ознакомиться с ним – при условии, что рекомендации в нем   
будут восприниматься как рекомендации, а не как обязательное   
руководство к действию.  
Инструкции выражений  
В языке Python выражения также могут использоваться в качестве инструк-  
ций (то есть в отдельной строке). Однако, поскольку результат вычисления та-  
ких выражений не сохраняется, использовать такую возможность имеет смысл   
только в том случае, если выражение выполняет какие-то полезные действия   
в виде побочного эффекта. В качестве инструкций выражения используются   
обычно в двух ситуациях:  
Для вызова функций и методов  
Некоторые функции и методы выполняют огромный объем работы, не воз-  
вращая никакого значения. В других языках программирования такие   
функции иногда называют процедурами. Поскольку они не возвращают   
значений, которые могло бы потребоваться сохранить, вы можете вызывать   
эти функции в инструкциях выражений.  
Для вывода значений в интерактивной оболочке  
В ходе интерактивного сеанса интерпретатор автоматически выводит ре-  
зультаты вводимых выражений. С технической точки зрения они также   
являются инструкциями выражений и играют роль сокращенной версии   
инструкции print.  
В табл. 11.4 перечислены некоторые наиболее часто используемые в языке   
Python формы инструкций выражений. При вызове функций и методов им   
передаются ноль или более объектов в виде аргументов (в действительности –   
выражений, результатом вычисления которых являются объекты) в круглых   
скобках, следующих за именем функции или метода.  
Таблица 11.4. Наиболее часто используемые в языке Python инструкции   
выражений  
Операция  
Интерпретация  
spam(eggs, ham)  
Вызов функции  
spam.ham(eggs)  
Вызов метода

Инструкции выражений   
361  
Операция  
Интерпретация  
spam  
Вывод значения переменной в интерактивной   
оболочке интерпретатора  
print(a, b, c, sep=’’)  
Операция ввода в Python 3.0  
yield x \*\* 2  
Инструкция выражения yield  
Последние две строки в таблице представляют специальные случаи: как бу-  
дет показано далее в этой главе, инструкция вывода в версии Python 3.0 была   
преобразована в функцию, вызов которой обычно оформляется в виде отдель-  
ной строки, и операция yield в функциях-генераторах (рассматриваются в гла-  
ве 20) также часто оформляется как инструкция. В действительности оба слу-  
чая являются всего лишь примерами инструкций выражений.  
Например, даже при том, что функция print вызывается в отдельной строке,   
как инструкция выражения, тем не менее она возвращает значение, как и лю-  
бая другая функция (она возвращает значение None – оно возвращается всеми   
функциями, которые явно не возвращают какого-либо значимого значения):  
>>> x = print(‘spam’) # print – это выражение вызова функции в версии 3.0,  
spam  
>>> print(x) # но может использоваться, как инструкция выражения  
None  
Несмотря на то, что выражения могут играть роль инструкций в языке Python,   
сами инструкции не могут использоваться в качестве выражений. Например,   
язык Python не допускает встраивание инструкции присваивания (=) в выра-  
жения. Сделано это специально, чтобы помочь избежать ошибок, – вы могли   
бы случайно изменить переменную, введя вместо оператора проверки равен-  
ства == оператор присваивания =. В главе 13 будет показано, как отсутствие та-  
кой возможности может быть компенсировано в языке Python, когда мы будем   
обсуждать цикл while.  
Инструкции выражений   
и непосредственное изменение объектов  
Инструкции выражений являются причиной распространенной ошибки при   
программировании на языке Python. Инструкции выражений часто исполь-  
зуются для вызова методов списка, которые непосредственно изменяют сам   
список:  
>>> L = [1, 2]  
>>> L.append(3) # Метод append изменяет сам список  
>>> L  
[1, 2, 3]  
Однако начинающие программисты нередко записывают такие операции   
в виде инструкций присваивания, пытаясь связать имя L со списком:  
>>> L = L.append(4) # Но метод append возвращает значение None, а не L  
>>> print L # Поэтому мы теряем весь список!  
None  
Такая операция дает неверный результат – такие методы списка, как apend,   
sort и reverse, всегда выполняют непосредственное изменение объекта, но они

362   
Глава 11. Присваивание, выражения и print   
не возвращают список, который был изменен с их помощью. В действительно-  
сти они возвращают объект None. Если результат такой операции присвоить той   
же переменной, вы потеряете список (скорее всего, он будет уничтожен в ходе   
процесса сборки мусора!).  
Поэтому не делайте этого. Мы еще раз вернемся к этому явлению в разделе   
«Распространенные ошибки программирования» в конце этой части книги,   
потому что подобные ситуации могут складываться в контексте выполнения   
некоторых операторов цикла, с которыми мы познакомимся в последующих   
главах.  
Операция print  
В языке Python инструкция print – это просто удобный для программистов ин-  
терфейс к стандартному потоку вывода.   
С технической точки зрения эта инструкция преобразует объекты в текстовое   
представление и либо посылает результат в поток стандартного вывода, либо   
передает другому объекту, похожему на файл. Инструкция print тесно связана   
с понятием файлов и потоков в языке Python.  
Методы объектов файлов  
В главе 9 мы рассматривали некоторые методы для работы с файлами, кото-  
рые выводят текст (например, file.write(str)). Инструкция print чем-то по-  
хожа на них, но она имеет более специализированное назначение: инструк-  
ция print по умолчанию записывает объекты в поток stdout (с соблюдением   
некоторого форматирования), в то время как методы файлов записывают   
строки в произвольные файлы. В отличие от методов файлов, инструкция   
print не требует преобразовывать объекты в строковое представление.  
Поток стандартного вывода  
Поток стандартного вывода в языке Python (известный под именем stdout) –   
это просто объект, куда программы выводят текст. Наряду с потоками стан-  
дартного ввода и стандартного вывода сообщений об ошибках, поток стан-  
дартного вывода является одним из трех потоков, которые создаются в мо-  
мент запуска сценария. Поток стандартного вывода обычно отображается   
на окно терминала, где была запущена программа на языке Python, если   
явно не было выполнено перенаправление вывода в файл или в конвейер си-  
стемной командной оболочки.  
Так как поток стандартного вывода в языке Python доступен в виде объекта   
stdout из встроенного модуля sys (то есть sys.stdout), вполне возможно имити-  
ровать поведение инструкции print с помощью методов записи в файл, хотя ис-  
пользование print выглядит проще.  
Операция вывода является одной из тех, в которых различия между Python 3.0   
и 2.6 оказались наиболее заметными. Фактически различия в реализации этой   
операции являются одной из основных причин, обусловливающих невозмож-  
ность выполнения программ, написанных для версии 2.X, под управлением   
версии 3.X без внесения изменений в программный код. В частности, от версии   
интерпретатора зависит способ оформления операции print:  
 •  
В Python 3.X инструкция print превратилась во встроенную функцию, ко-  
торая принимает именованные аргументы, определяющие специальные ре-  
жимы вывода.

Операция print   
363  
 •  
В Python 2.X print – это инструкция, имеющая свой характерный синтак-  
сис.  
Так как в этой книге рассматриваются обе версии интерпретатора, 3.0 и 2.6,   
мы рассмотрим особенности реализации print в каждой из них по отдельности.   
Если вам повезло настолько, что вы работаете с программами, написанными   
для какой-то одной версии Python, вы можете прочитать только тот раздел, ко-  
Python, вы можете прочитать только тот раздел, ко-  
, вы можете прочитать только тот раздел, ко-  
торый подходит для вашего случая. Однако ваша ситуация может измениться,   
поэтому для вас не будет лишним познакомиться с обеими версиями.  
Функция print в Python 3.0  
Строго говоря, print не является разновидностью инструкции в версии 3.0,   
это просто одна из инструкций выражений, с которыми мы познакомились   
в преды дущем разделе.  
Обычно вызов встроенной функции print оформляется в виде отдельной строки,   
потому что она не возвращает никакого значения (точнее, она возвращает объ-  
ект None), о сохранении которого стоило бы позаботиться. Так как в версии 3.0   
print – это обычная функция, для обращения к ней используется не какая-то   
специальная форма, а стандартный синтаксис вызова функций. Вследствие   
этого специальные режимы работы определяются с помощью именованных ар-  
гументов. Такая форма не только более универсальна, но и обеспечивает луч-  
шую поддержку будущих расширений.  
Для сравнения, инструкция print в версии Python 2.6 имеет специализирован-  
Python 2.6 имеет специализирован-  
 2.6 имеет специализирован-  
ный синтаксис поддержки таких особенностей, как подавление вывода симво-  
ла конца строки и перенаправление вывода в файл. Кроме того, инструкция   
print в версии 2.6 вообще не позволяет определить строки-разделители – в вер-  
сии 2.6 вам намного чаще придется конструировать строки заранее, чем в вер-  
сии 3.0.  
Формат вызова  
Синтаксис вызова функции print в версии 3.0 имеет следующий вид:  
print([object, ...][, sep=’ ‘][, end=’\n’][, file=sys.stdout])  
Здесь элементы в квадратных скобках являются необязательными и могут   
быть опущены, а значения, следующие за знаком =, определяют значения по   
умолчанию. Проще говоря, эта встроенная функция выводит в файл stream   
один или более объектов в текстовом представлении, разделенных строкой sep,   
и завершает вывод строкой end.  
Параметры sep, end и file, если они необходимы, должны передаваться не   
в виде позиционных, а в виде именованных аргументов, то есть с использо-  
ванием специального синтаксиса «имя�значение». Именованные аргументы   
по дробно рассматриваются в главе 18, но они достаточно просты в использо-  
вании. Именованные аргументы могут указываться в любом порядке в вызове   
функции, но после объектов, предназначенных для вывода, и определяют па-  
раметры вывода:  
 •  
sep – строка, которая должна вставляться между объектами при выводе.   
По умолчанию состоит из одного пробела. Чтобы подавить вывод строки-  
разделителя, в этом аргументе следует передать пустую строку.

364   
Глава 11. Присваивание, выражения и print   
 •  
end – строка, добавляемая в конец выводимого текста. По умолчанию содер-  
жит символ конца строки \n. Если в этом аргументе передать пустую стро-  
ку, следующий вызов функции print начнет вывод текста с позиции, где за-  
кончился вывод текущей строки.  
 •  
file – объект файла, стандартный поток или другой объект, похожий на   
файл, куда будет выводиться текст. По умолчанию используется объект   
sys.stdout стандартного потока вывода. В этом аргументе можно передать   
любой объект, поддерживающий метод файлов write(string)� если передает-  
ся настоящий объект файла, он должен уже быть открыт для записи.  
Текстовое представление любого объекта, который передается для вывода,   
функция print получает с помощью встроенной функции str. Как мы уже зна-  
ем, эта функция возвращает «удобочитаемое» строковое представление любо-  
го объекта.1 При вызове без аргументов функция print просто выведет символ   
конца строки в поток стандартного вывода, что выглядит, как вывод пустой   
строки.  
Функция print в действии  
Пользоваться функцией print в версии 3.0 намного проще, чем можно было   
бы предположить, после знакомства с некоторыми ее особенностями. Для ил-  
 ил-  
ил-  
люстрации выполним несколько коротких примеров. Ниже демонстрируется   
порядок вывода объектов различных типов в поток стандартного вывода, ис-  
пользуемый по умолчанию, со строкой-разделителем по умолчанию и с добав-  
лением символа конца строки (эти значения параметров вывода были выбра-  
ны значениями по умолчанию, потому что они используются в подавляющем   
большинстве случаев):   
C:\misc> c:\python30\python  
>>>  
>>> print() # Выведет пустую строку  
>>> x = ‘spam’  
>>> y = 99  
>>> z = [‘eggs’]  
>>>  
>>> print(x, y, z) # Выведет три объекта  
spam 99 [‘eggs’]  
В данном примере нет никакой необходимости преобразовывать объекты   
в строки, как этого требуют методы записи в файлы. По умолчанию функция   
print выводит символ пробела между объектами. Чтобы подавить вывод про-  
бела, достаточно просто передать пустую строку в именованном аргументе sep   
или указать иную строку-разделитель:  
1   
С технической точки зрения, при выводе объектов внутри используется не сама   
функция str, а некоторая эквивалентная реализация, хотя конечный результат по-  
лучается тем же самым. Имя str не только представляет функцию преобразования   
в строку, но также является именем строкового типа данных и может использовать-  
ся для декодирования строк Юникода из строк байтов, принимая дополнительный   
аргумент с названием кодировки, о чем подробно рассказывается в главе 36. Эта по-  
следняя роль является дополнительной особенностью, которую здесь можно спокой-  
но игнорировать.

Операция print   
365  
>>> print(x, y, z, sep=’’) # Строка-разделитель выводиться не будет  
spam99[‘eggs’]  
>>>  
>>> print(x, y, z, sep=’, ‘) # Нестандартная строка-разделитель  
spam, 99, [‘eggs’]  
Кроме того, в конце выводимой строки функция print добавляет символ конца   
строки. Вывод этого символа можно подавить, передав пустую строку в име-  
 этого символа можно подавить, передав пустую строку в име-  
этого символа можно подавить, передав пустую строку в име-  
 символа можно подавить, передав пустую строку в име-  
символа можно подавить, передав пустую строку в име-  
 можно подавить, передав пустую строку в име-  
можно подавить, передав пустую строку в име-  
 подавить, передав пустую строку в име-  
подавить, передав пустую строку в име-  
, передав пустую строку в име-  
передав пустую строку в име-  
 пустую строку в име-  
пустую строку в име-  
 строку в име-  
строку в име-  
 в име-  
в име-  
 име-  
име-  
нованном аргументе end. Точно так же в этом аргументе можно передать иной   
разделитель строк (включая символ \n, чтобы вручную обеспечить перевод   
строки):  
>>> print(x, y, z, end=’’) # Подавление вывода символа конца строки  
spam 99 [‘eggs’]>>>  
>>>  
>>> print(x, y, z, end=’’); print(x, y, z) # Вывод двух строк в одной строке  
spam 99 [‘eggs’]spam 99 [‘eggs’]  
>>> print(x, y, z, end=’...\n’) # Нестандартный разделитель строк  
spam 99 [‘eggs’]...  
>>>  
Допускается одновременное использование именованных аргументов, опреде-  
ляющих разделители между объектами в строке и между строками, – они мо-  
гут указываться в любом порядке, но только после объектов, которые требует-  
ся вывести:  
>>> print(x, y, z, sep=’...’, end=’!\n’) # Несколько именованных аргументов  
spam...99...[‘eggs’]!  
>>> print(x, y, z, end=’!\n’, sep=’...’) # Порядок не имеет значения  
spam...99...[‘eggs’]!  
Ниже демонстрируется порядок использования именованного аргумента file –   
с его помощью можно перенаправить вывод текста в файл, открытый для за-  
писи, или в другой объект, совместимый с операцией вывода (в действительно-  
сти – это разновидность перенаправления потоков, о чем будет рассказываться   
ниже в этом разделе):  
>>> print(x, y, z, sep=’...’, file=open(‘data.txt’, ‘w’)) # Вывод в файл  
>>> print(x, y, z) # Вывод в поток stdout  
spam 99 [‘eggs’]  
>>> print(open(‘data.txt’).read()) # Вывод содержимого текстового файла  
spam...99...[‘eggs’]  
Наконец, имейте в виду, что возможность определить строку-разделитель объ-  
ектов в строке и разделитель строк, предоставляемая функцией print, – это   
всего лишь подручное средство. При необходимости получить более сложное   
форматирование совсем не обязательно использовать эту возможность. Вме-  
сто этого можно сконструировать строку заранее или прибегнуть к помощи   
инструментов форматирования, с которыми мы встречались в главе 7, внутри   
вызова функции print, и вывести сконструированную строку единственным   
вызовом print:  
>>> text = ‘%s: %-.4f, %05d’ % (‘Result’, 3.14159, 42)  
>>> print(text)  
Result: 3.1416, 00042

366   
Глава 11. Присваивание, выражения и print   
>>> print(‘%s: %-.4f, %05d’ % (‘Result’, 3.14159, 42))  
Result: 3.1416, 00042  
Как мы увидим в следующем разделе, практически все, что было сказано   
о функции print в Python 3.0, точно так же применимо и к инструкции print   
в версии 2.6, особенно если учесть, что функция задумывалась как дальней-  
шее развитие и улучшение поддержки операции вывода в 2.6.  
Инструкция print в Python 2.6  
Как уже упоминалось выше, операция вывода в Python 2.6 реализована в виде   
инструкции с уникальным и весьма специфическим синтаксисом, а не в виде   
встроенной функции. На практике же печать? организованная в 2.6, является   
вариацией на ту же тему – за исключением строк-разделителей (которые под-  
держиваются в версии 3.0 и не поддерживаются в версии 2.6), для всех осталь-  
ных возможностей функции print в версии 3.0 имеют прямые аналоги в ин-  
струкции print в версии 2.6.  
Формы инструкции  
В табл. 11.5 перечислены формы инструкции print в Python 2.6 и эквивалент-  
Python 2.6 и эквивалент-  
 2.6 и эквивалент-  
ные им вызовы функции print в Python 3.0. Обратите внимание, что запятая   
в инструкции print имеет важное значение – она отделяет объекты, которые   
требуется вывести, а завершающая запятая подавляет вывод символа конца   
строки, который обычно выводится в конце строки текста (не путайте с син-  
таксисом кортежей!). Синтаксическая конструкция >> обычно используется   
как оператор побитового сдвига вправо, однако он может использоваться и в   
инструкции print, чтобы определить поток вывода, отличный от sys.stdout, ко-  
торый используется по умолчанию.  
Таблица 11.5. Перечень форм инструкции print  
Инструкция print   
в Python 2.6  
Эквивалент   
в Python 3.0  
Интерпретация  
print x, y  
print(x, y)  
Вывод объектов в sys.stdout�   
добавляет пробел между   
объектами и символ конца   
строки  
print x, y,  
print(x, y, end=’’)  
То же самое, только на этот   
раз символ конца строки не   
добавляется  
print >> afile, x, y  
print(x, y, file=afile)  
Текст передается методу   
myfile.write, а не sys.stdout.  
write  
Инструкция print в действии  
Хотя синтаксис инструкции print в Python 2.6 отличается от синтаксиса функ-  
Python 2.6 отличается от синтаксиса функ-  
 2.6 отличается от синтаксиса функ-  
ции print в Python 3.0, тем не менее она точно так же проста в использовании.   
Давайте рассмотрим несколько простых примеров. По умолчанию инструкция

Операция print   
367  
print в версии 2.6 добавляет пробел между элементами, отделенными запяты-  
ми, и символ конца строки в конце текущей строки вывода:  
C:\misc> c:\python26\python  
>>>  
>>> x = ‘a’  
>>> y = ‘b’  
>>> print x, y  
a b  
Такое форматирование – всего лишь значение по умолчанию, которое можно   
использовать или нет. Чтобы подавить вывод символа конца строки (и позднее   
продолжить вывод текста в текущую строку), инструкцию print следует завер-  
шать символом запятой, как показано во второй строке табл. 11.5 (ниже приво-  
дятся две инструкции в одной строке, разделенные точкой с запятой):  
>>> print x, y,; print x, y  
a b a b  
Чтобы подавить вывод пробела между элементами, следует производить вывод   
не таким способом – вместо этого нужно самостоятельно собрать строку с по-  
мощью операции конкатенации и форматирования, о которых рассказывалось   
в главе 7, и вывести эту строку:  
>>> print x + y  
ab  
>>> print ‘%s...%s’ % (x, y)  
a...b  
Как видно из примеров, если не считать специального синтаксиса задания раз-  
личных режимов вывода, инструкция print в версии 2.6 почти так же проста   
в использовании, как и функция print в версии 3.0. В следующем разделе опи-  
сывается способ перенаправления вывода в файл с помощью инструкции print   
в 2.6.  
Перенаправление потока вывода в инструкции print  
В обеих версиях, Python 3.0 и 2.6, выводимый текст по умолчанию передает-  
Python 3.0 и 2.6, выводимый текст по умолчанию передает-  
 3.0 и 2.6, выводимый текст по умолчанию передает-  
ся в поток стандартного вывода. Однако достаточно часто возникает необхо-  
димость вывести текст в какое-нибудь другое место, например в файл, чтобы   
сохранить результаты для последующего использования или для нужд тести-  
рования. Подобное перенаправление вывода можно организовать на уровне   
системной командной оболочки, за пределами интерпретатора, однако перена-  
правление внутри сценария реализуется ничуть не сложнее.  
Программа «hello world» на языке Python  
Начнем с привычного (и обыденного) теста языка программирования – с про-  
граммы «hello world». Чтобы вывести сообщение «hello world» в программе на   
языке Python, достаточно просто напечатать строку:  
>>> print(‘hello world’) # Вызов строкового объекта в 3.0  
hello world   
   
>>> print ‘hello world’ # Вызов строкового объекта в 2.6  
hello world

368   
Глава 11. Присваивание, выражения и print   
Поскольку результаты выражений в интерактивной оболочке выводятся авто-  
матически, вы можете даже не использовать инструкцию print – просто введи-  
те выражение, которое требуется вывести, и результат его вычисления немед-  
ленно будет выведен:  
>>> ‘hello world’ # Интерактивная оболочка выводит результат автоматически  
‘hello world’  
Этот фрагмент трудно назвать примером программистского мастерства, тем   
не менее он наглядно демонстрирует особенности поведения операции вывода.   
В действительности инструкция print – это всего лишь эргономичная особен-  
ность языка Python, – она обеспечивает простой интерфейс к объекту sys.std-  
.std-  
std-  
out, добавляя незначительный объем форматирования. Фактически если вам   
нравится идти более трудным путем, вы можете запрограммировать вывод та-  
ким способом:  
>>> import sys # Вывод более сложным способом  
>>> sys.stdout.write(‘hello world\n’)  
hello world  
В этом фрагменте явно вызывается метод write объекта sys.stdout – атрибут,   
предустановленный интерпретатором Python во время открытия файла, свя-  
занного с потоком вывода. Инструкция print скрывает большую часть этих   
подробностей, предоставляя простой инструмент для решения простых задач   
вывода.  
Перенаправление потока вывода вручную  
Итак, зачем я показал более сложный способ вывода? Как оказывается, объект   
sys.stdout обеспечивает возможность вывода, эквивалентную базовой методи-  
ке, используемой в языке Python. Вообще говоря, инструкция print и объект   
sys.stdout связаны между собой следующим образом. Инструкция:  
print(X, Y) # Или print X, Y в версии 2.6  
является эквивалентом более длинной:  
import sys  
sys.stdout.write(str(X) + ‘ ‘ + str(Y) + ‘\n’)   
которая вручную выполняет преобразование объекта в строку с помощью   
функции str, добавляет символ конца строки с помощью оператора + и вызы-  
вает метод write потока вывода. А как бы вы выполнили ту же задачу? (Этим   
примером мне хотелось бы подчеркнуть дружественную природу инструкции   
print...)  
Очевидно, что более длинная форма сама по себе менее удобна в использова-  
нии. Однако полезно знать, что она является точным эквивалентом инструк-  
ции print, потому что существует возможность переназначить sys.stdout на   
что-то, отличное от стандартного потока вывода. Другими словами, эта экви-  
валентность обеспечивает возможность заставить инструкцию print выводить   
текст в другое место. Например:  
import sys  
sys.stdout = open(‘log.txt’, ‘a’) # Перенаправить вывод в файл  
...  
Print(x, y, x) # Текст появится в файле log.txt

Операция print   
369  
Здесь мы вручную перенаправили объект sys.stdout в файл, открытый вруч-  
ную в режиме добавления (так как мы добавляем новое содержимое). После   
этого все инструкции print в программе будут выводить текст в конец файла   
log.txt, а не в стандартный поток вывода. Инструкции print благополучно про-  
должают вызывать метод write объекта sys.stdout независимо от того, куда он   
ссылается. Поскольку в каждом процессе существует всего один модуль sys,   
перенаправление sys.stdout таким способом будет воздействовать на все ин-  
струкции print в программе.  
Фактически, как будет говориться в ближайшей врезке, описывающей ин-  
струкцию print и объект stdout, существует возможность перенаправить sys.  
stdout в объект, который даже не является файлом, при условии, что он под-  
держивает ожидаемый интерфейс: метод write, принимающий строковый ар-  
гумент. Этот объект может быть классом, способным обрабатывать и перена-  
правлять выводимый текст произвольным образом.  
Этот прием с перенаправлением потока вывода в первую очередь может ока-  
заться полезен в программах, изначально рассчитанных на использование ин-  
струкции print. Если известно, что весь вывод должен отправляться в файл, вы   
всегда сможете организовать вызов методов записи в файл. При перенаправле-  
нии потока вывода в программах, основанных на использовании инструкции   
print, настройка объекта sys.stdout обеспечивает удобную альтернативу изме-  
нению поведения всех инструкций print или применению перенаправления   
средствами командной оболочки системы.  
Автоматическое перенаправление потоков  
Прием перенаправления вывода текста за счет назначения файла в объекте   
sys.stdout очень часто используется на практике. Однако в программном коде   
предыдущего раздела имеется одна потенциальная проблема – отсутствует   
прямой способ восстановления первоначального потока вывода, если вдруг по-  
сле вывода данных в файл потребуется вернуться обратно к выводу на экран.   
Но поскольку sys.stdout является обычным объектом, вы всегда можете в слу-  
чае необходимости сохранить его и восстановить позднее:1  
C:\misc> c:\python30\python  
>>> import sys  
>>> temp = sys.stdout # Сохранить для последующего восстановления  
>>> sys.stdout = open(‘log.txt’, ‘a’) # Перенаправить вывод в файл  
>>> print(‘spam’) # Выведет в файл, а не на экран  
>>> print(1, 2, 3)  
>>> sys.stdout.close() # Вытолкнуть буферы на диск  
>>> sys.stdout = temp # Восстановить первоначальный поток  
   
>>> print(‘back here’) # Вывести на экран  
back here  
>>> print(open(‘log.txt’).read()) # Результаты более ранних обращений  
spam # к инструкции print  
1 2 3  
1   
В обеих версиях, 2.6 и 3.0, можно также использовать атрибут \_\_stdout\_\_ модуля sys,   
который ссылается на первоначальное значение sys.stdout, имевшееся на момент   
запуска программы. Но вам и в этом случае необходимо вручную восстановить sys.  
stdout в первоначальное значение sys.\_\_stdout\_\_, чтобы вернуться к оригинальному   
потоку вывода. За дополнительными подробностями обращайтесь к описанию моду-  
ля sys в руководстве по стандартной библиотеке.

370   
Глава 11. Присваивание, выражения и print   
Потребность в таком перенаправлении возникает на удивление часто, а ручное   
сохранение и восстановление оригинального потока вывода – процедура доста-  
точно сложная, что привело к появлению расширения для инструкции print,   
которое делает такое перенаправление ненужным.   
В версии 3.0 именованный аргумент file позволяет перенаправить в файл вы-  
вод единственного вызова функции print, не прибегая к переустановке значе-  
ния sys.stdout. Так как такое перенаправление носит временный характер,   
обычные вызовы функции print продолжают выводить текст в оригинальный   
поток вывода. В версии 2.6 того же эффекта можно добиться, указав в начале   
инструкции print символы >> и вслед за ними – объект файла (или другой со-  
вместимый объект). Например, следующий фрагмент выводит текст в файл log.  
txt:  
log = open(‘log.txt’, ‘a’) # 3.0  
print(x, y, z, file=log) # Вывести в объект, напоминающий файл  
print(a, b, c) # Вывести в оригинальный поток вывода   
   
log = open(‘log.txt’, ‘a’) # 2.6  
print >> log, x, y, z # Вывести в объект, напоминающий файл  
print a, b, c # Вывести в оригинальный поток вывода  
Эти способы перенаправления удобно использовать, когда в одной и той же про-  
грамме необходимо организовать вывод и в файл, и в стандартный поток выво-  
да. Однако если вы собираетесь использовать эти формы вывода, вам потребу-  
ется создать объект-файл (или объект, который имеет метод write, как и объект   
файла) и передавать инструкции этот объект, а не строку с именем файла:  
C:\misc> c:\python30\python  
>>> log = open(‘log.txt’, ‘w’)  
>>> print(1, 2, 3, file=log) # В 2.6: print >> log, 1, 2, 3  
>>> print(4, 5, 6, file=log)  
>>> log.close()  
>>> print(7, 8, 9) # В 2.6: print 7, 8, 9  
7 8 9  
>>> print(open(‘log.txt’).read())  
1 2 3  
4 5 6  
Эти расширенные формы инструкции print нередко используются для выво-  
да сообщений об ошибках в стандартный поток ошибок, sys.stderr. Вы можете   
либо использовать его метод write и форматировать выводимые строки вруч-  
ную, либо использовать синтаксис перенаправления:  
>>> import sys  
>>> sys.stderr.write((‘Bad!’ \* 8) + ‘\n’)  
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!  
   
>>> print(‘Bad!’ \* 8, file=sys.stderr) # В 2.6: print >> sys.stderr, ‘Bad’ \* 8  
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!  
Теперь, когда вы знаете все о перенаправлении вывода, эквивалентность функ-  
ции print и метода write файлов должна показаться вам очевидной. В следую-  
щем примере демонстрируется вывод текста обоими способами в версии 3.0,   
затем демонстрируется перенаправление вывода во внешний файл и выполня-  
ется проверка содержимого текстовых файлов:

Операция print   
371  
>>> X = 1; Y = 2  
>>> print(X, Y) # Вывод: простой способ  
1 2  
>>> import sys # Вывод: сложный способ  
>>> sys.stdout.write(str(X) + ‘ ‘ + str(Y) + ‘\n’)  
1 2  
4  
>>> print(X, Y, file=open(‘temp1’, ‘w’)) # Перенаправление в файл  
   
>>> open(‘temp2’, ‘w’).write(str(X) + ‘ ‘ + str(Y) + ‘\n’) # Запись в файл  
4  
>>> print(open(‘temp1’, ‘rb’).read()) # Двоичный режим  
b’1 2\r\n’  
>>> print(open(‘temp2’, ‘rb’).read())  
b’1 2\r\n’  
Как видите, для отображения текста лучше пользоваться операцией print,   
если только вы не получаете особое удовольствие от ввода с клавиатуры. Дру-  
гой пример, демонстрирующий эквивалентность использования операции   
print и метода write файлов, – пример имитации функции print для Python вер-  
сий 2.6 и более ранних, вы найдете в главе 18.  
Реализация вывода,   
не зависящая от версии интерпретатора  
Наконец, для тех, кто не может ограничиться только версией Python 3.0 и ищет   
способ организации вывода, совместимый с 3.0, имеется несколько вариантов   
на выбор. Первый состоит в том, чтобы использовать инструкции print для вер-  
сии 2.6 и затем автоматически преобразовывать их в эквивалентные вызовы   
функции print для версии 3.0 с помощью сценария 2to3. Дополнительные по-  
дробности об этом сценарии вы найдете в документации к Python 3.0 – он пыта-  
Python 3.0 – он пыта-  
 3.0 – он пыта-  
ется преобразовать программный код, написанный для версии 2.X, так, чтобы   
он мог выполняться под управлением Python 3.0.  
Второй заключается в том, чтобы использовать функцию print для версии 3.0   
в программах, выполняющихся под управлением Python 2.6, включив воз-  
Python 2.6, включив воз-  
 2.6, включив воз-  
можность использования этой функции с помощью инструкции:  
from \_\_future\_\_ import print\_function  
Эта инструкция включает поддержку разновидности инструкции print в ин-  
терпретаторе Python 2.6, которая точно соответствует функции print в вер-  
сии 3.0. Благодаря этому вы сможете использовать все возможности функции   
print в версии 2.6, и при этом позднее вам не придется менять операции вывода   
при переходе на новую версию 3.0.   
Кроме того, имейте в виду, что простые случаи использования операции вы-  
 того, имейте в виду, что простые случаи использования операции вы-  
того, имейте в виду, что простые случаи использования операции вы-  
, имейте в виду, что простые случаи использования операции вы-  
имейте в виду, что простые случаи использования операции вы-  
 в виду, что простые случаи использования операции вы-  
в виду, что простые случаи использования операции вы-  
 виду, что простые случаи использования операции вы-  
виду, что простые случаи использования операции вы-  
, что простые случаи использования операции вы-  
что простые случаи использования операции вы-  
 простые случаи использования операции вы-  
простые случаи использования операции вы-  
 случаи использования операции вы-  
случаи использования операции вы-  
 использования операции вы-  
использования операции вы-  
 операции вы-  
операции вы-  
 вы-  
вы-  
вода, как в первой строке табл. 11.5, будут действовать в любой версии Python.   
Благодаря тому, что синтаксис языка позволяет заключать в круглые скобки   
любые выражения, мы всегда можем имитировать вызов функции print вер-  
сии 3.0 в программном коде для версии 2.6, просто добавив скобки. Единствен-  
ный недостаток такого подхода состоит в том, что он образует кортеж из выво-  
димых объектов, когда их более одного – в выводимом тексте они будут заклю-  
чены в круглые скобки. В версии 3.0, например, в круглых скобках, в вызове   
функции, может быть указано произвольное количество объектов:

372   
Глава 11. Присваивание, выражения и print   
C:\misc> c:\python30\python  
>>> print(‘spam’) # Синтаксис вызова функции print в 3.0   
spam  
>>> print(‘spam’, ‘ham’, ‘eggs’) # Вызов с нескольким аргументами  
spam ham eggs  
В первом случае в версии 2.6 результат будет тот же самый, но во втором случае   
будет создан кортеж:  
C:\misc> c:\python26\python  
>>> print(‘spam’) # 2.6: использование скобок в инструкции  
spam  
>>> print(‘spam’, ‘ham’, ‘eggs’) # В действительности – это объект кортежа!  
(‘spam’, ‘ham’, ‘eggs’)  
Чтобы обеспечить настоящую независимость от версии интерпретатора, необ-  
ходимо сначала сконструировать строку, используя оператор форматирования   
строк, или метод format, или другие инструменты для работы со строками, с ко-  
торыми мы познакомились в главе 7:  
>>> print(‘%s %s %s’ % (‘spam’, ‘ham’, ‘eggs’))  
spam ham eggs  
>>> print(‘{0} {1} {2}’.format(‘spam’, ‘ham’, ‘eggs’))  
spam ham eggs  
Разумеется, если имеется возможность пользоваться исключительно версией   
Python 3.0, можно вообще позабыть о способах обеспечения совместимости   
версий. Однако многим программистам придется еще не раз столкнуться или   
писать программный код для версии Python 2.X.  
На протяжении всей книги я использую функцию print в вер-  
сии Python 3.0. Обычно я буду предупреждать, что при опробо-  
вании примеров в интерпретаторе версии 2.6 в выводе могут   
появиться дополнительные круглые скобки, потому что при на-  
личии нескольких элементов образуется кортеж, но иногда я не   
буду делать этого, поэтому считайте это примечание предупре-  
ждением – если вы увидите дополнительные круглые скобки   
в выводе при опробовании примеров в интерпретаторе вер-  
сии 2.6, то либо опустите скобки в инструкции print, либо пере-  
пишите инструкции вывода, используя приемы обеспечения   
совместимости, описанные здесь, либо попытайтесь полюбить   
эти лишние скобки.  
Придется держать в уме: print и stdout  
Эквивалентность инструкции print и метода write объекта sys.stdout   
имеет большое значение. Она позволяет перенаправить объект sys.std-  
out в определяемый пользователем объект, который поддерживает тот   
же самый метод write, что и файлы. Так как инструкция print всего   
лишь передает текст методу sys.stdout.write, вы можете перехватывать   
выводимый текст, перенаправив sys.stdout в объект, обладающий мето-  
дом write для обработки текста.

Операция print   
373  
Например, можно отправить текст в окно графического интерфейса или   
отправить его в несколько мест, определив объект с методом write, ко-  
торый выполнит все необходимые действия. Пример использования та-  
кой возможности вы увидите в шестой части этой книги, когда мы будем   
изучать классы, но в общих чертах это выглядит примерно так:  
class FileFaker:  
 def write(self, string):  
 # Выполнить какие-либо действия со строкой  
   
import sys  
sys.stdout = FileFaker()  
print someObjects # Передает строку методу write класса  
Этот прием возможен благодаря тому, что инструкция print являет-  
ся тем, что в следующей части книги мы назовем полиморфной опера-  
цией, – она не интересуется тем, что из себя представляет объект sys.  
stdout, ей нужен всего лишь метод (то есть интерфейс) с именем write.   
Такое перенаправление в объекты может быть реализовано еще проще   
с помощью именованного аргумента file в версии 3.0 и расширенной   
формы >> инструкции print в версии 2.6, благодаря чему нам не требу-  
ется явно перенаправлять объект sys.stdout – обычные операции вывода   
по-прежнему будут выводить текст в объект sys.stdout:  
myobj = FileFaker() # 3.0: Перенаправление вывода для одной инструкции  
print(someObjects, file=myobj) # Не влияет на объект sys.stdout  
   
myobj = FileFaker() # 2.6: Перенаправление вывода для одной инструкции  
print >> myobj, someObjects # Не влияет на объект sys.stdout  
Встроенная функция input в языке Python читает данные из файла sys.  
stdin, благодаря чему существует возможность перенаправить ввод ана-  
логичным образом, используя классы, реализующие метод read. Смо-  
трите пример использования функции input и цикла while, который при-  
водится в главе 10, чтобы лучше представлять себе, о чем речь.  
Обратите внимание, что вывод текста осуществляется в поток stdout, – это   
обеспечивает возможность вывода документов HTML в CGI-сценариях.   
Кроме того, это позволяет выполнить перенаправление ввода и вывода   
для сценария на языке Python обычными средствами командной строки   
операционной системы:  
python script.py < inputfile > outputfile  
python script.py | filterProgram  
Механизм перенаправления операций вывода в языке Python являет-  
ся альтернативой этим видам перенаправления средствами командной   
оболочки.

374   
Глава 11. Присваивание, выражения и print   
В заключение  
В этой главе мы начали детальный обзор инструкций языка Python, иссле-  
довав инструкции присваивания, выражения и операции print. Несмотря на   
всю простоту использования этих инструкций, у них имеются альтернативные   
формы, которые являются необязательными, но порой достаточно удобными   
в применении. Комбинированные инструкции присваивания и возможность   
перенаправления вывода в операциях print, например, позволяют уменьшить   
объем программного кода. Кроме того, мы изучили синтаксис имен перемен-  
ных, приемы перенаправления потоков и различные часто встречающиеся   
ошибки, которых следует избегать, такие как обратное присваивание перемен-  
ной значения, возвращаемого методом append.  
В следующей главе мы продолжим наше знакомство с инструкциями, подроб-  
но рассмотрев инструкцию if – основную инструкцию организации выбора   
в языке Python. В этой главе мы еще раз вернемся к синтаксической модели   
языка и рассмотрим поведение логических выражений. Но прежде чем дви-  
нуться дальше, проверьте знания, полученные здесь, с помощью контрольных   
вопросов.  
Закрепление пройденного  
Контрольные вопросы  
1. Назовите три способа, с помощью которых можно присвоить одно и то же   
значение трем переменным.  
2. Что требуется держать в уме, когда трем переменным присваивается один   
и тот же изменяемый объект?  
3. В чем заключается ошибка в инструкции L = L.sort()?  
4. Как с помощью инструкции print вывести текст во внешний файл?  
Ответы  
1. Можно выполнить групповое присваивание (A = B = C = 0), присваивание по-  
следовательности (A, B, C = 0, 0, 0) или несколько инструкций присваивания   
в отдельных строках (A = 0, B = 0 и C = 0). И последний способ, как было пока-  
зано в главе 10, заключается в том, чтобы объединить инструкции в одной   
строке, разделив их точкой с запятой (A = 0; B = 0; C = 0).  
2. Если выполнить присваивание следующим образом:  
A = B = C = []  
то все три имени будут ссылаться на один и тот же объект, поэтому непо-  
средственное изменение объекта с помощью одной переменной (например,   
A.append(99)) отразится на других. Это справедливо только для изменений,   
производимых непосредственно в изменяемых объектах, таких как списки   
и словари. Для неизменяемых объектов, таких как числа и строки, эта про-  
блема не проявляется.  
3. Метод списков sort, как и метод append, выполняет изменения непосредствен-  
но в объекте и возвращает значение None, а не измененный список. Обратное   
присваивание переменной L приведет к тому, что в нее запишется значение

Закрепление пройденного   
375  
None, а не отсортированный список. Как будет показано далее в этой части   
книги, недавно в языке появилась новая функция sorted, которая выполня-  
ет сортировку любых последовательностей и возвращает новый список с ре-  
зультатами сортировки. Поскольку в этом случае изменения производятся   
не в самом объекте, есть смысл сохранить возвращаемое значение.  
4. Чтобы реализовать вывод в файл в единственной операции print, в вер-  
сии 3.0 можно использовать вызов вида print(X, file=F), а в версии 2.6 ис-  
пользовать расширенную форму инструкции print >> file или вручную   
перенаправить объект sys.stdout, открыв файл перед вызовом инструкции   
print и восстановив оригинальное значение после вывода. Можно также   
перенаправить весь выводимый программой текст в файл с помощью син-  
таксических конструкций системной командной оболочки, но этот вариант   
к языку Python никакого отношения не имеет.

Глава 12.  
   
Условная инструкция if   
и синтаксические правила  
Эта глава представляет условную инструкцию if, которая является основной   
инструкцией, используемой для выбора среди альтернативных операций на   
основе результатов проверки. Поскольку в нашем обзоре это первая составная   
инструкция – инструкция, встроенная в другую инструкцию, – мы заодно бо-  
лее подробно, чем в главе 10, рассмотрим общие концепции, на которых поко-  
ится синтаксическая модель инструкций в языке Python. А так как условная   
инструкция вводит понятие проверки, мы также будем иметь дело с логиче-  
скими выражениями и остановимся на некоторых деталях относительно про-  
верок истинности вообще.   
Условные инструкции if  
Если говорить простым языком, в Python инструкция if выбирает, какое дей-  
ствие следует выполнить. Это основной инструмент выбора в Python, который   
отражает большую часть логики программы на языке Python. Кроме того, это   
наша первая составная инструкция. Как и все составные инструкции языка   
Python, инструкция if может содержать другие инструкции, в том числе дру-  
гие условные инструкции if. Фактически Python позволяет комбинировать ин-  
Python позволяет комбинировать ин-  
 позволяет комбинировать ин-  
струкции в программные последовательности (чтобы они выполнялись одна за   
другой) и в произвольно вложенные конструкции (которые выполняются толь-  
ко при соблюдении определенных условий).  
Общая форма  
Условная инструкция if в языке Python – это типичная условная инструкция,   
которая присутствует в большинстве процедурных языков программирова-  
ния. Синтаксически сначала записывается часть if с условным выражением,   
далее могут следовать одна или более необязательных частей elif («else if»)   
с условными выражениями и, наконец, необязательная часть else. Условные   
выражения и часть else имеют ассоциированные с ними блоки вложенных ин-

Условные инструкции if   
377  
струкций, с отступом относительно основной инструкции. Во время выполне-  
ния условной инструкции if интерпретатор выполняет блок инструкций, ас-  
социированный с первым условным выражением, только если оно возвращает   
истину, в противном случае выполняется блок инструкций else. Общая форма   
записи условной инструкции if выглядит следующим образом:  
if <test1>: # Инструкция if с условным выражением test1  
 <statements1> # Ассоциированный блок  
elif <test2>: # Необязательные части elif  
 <statements2>  
else: # Необязательный блок else  
 <statements3>  
Простые примеры  
C целью демонстрации инструкции if в действии рассмотрим несколько про-  
стых примеров. Все части этой инструкции, за исключением основной части   
if с условным выражением и связанных с ней инструкций, являются необяза-  
тельными. В простейшем случае остальные части инструкции опущены:  
>>> if 1:  
... print ‘true’  
...  
true  
Обратите внимание, что приглашение к вводу изменяется на ... для строк про-  
должения в базовом интерфейсе командной строки, используемом здесь (в IDLE   
текстовый курсор просто перемещается на следующую строку уже с отступом,   
а нажатие на клавишу Backspace возвращает на строку вверх). Ввод пустой стро-  
ки (двойным нажатием клавиши Enter) завершает инструкцию и приводит к ее   
выполнению. Вспомните, что число 1 является логической истиной, поэтому   
данная проверка всегда будет успешной. Чтобы обработать ложный результат,   
добавьте часть else:  
>>> if not 1:  
... print ‘true’  
... else:  
... print ‘false’  
...  
false  
Множественное ветвление  
Теперь рассмотрим пример более сложной условной инструкции if, в которой   
присутствуют все необязательные части:  
>>> x = ‘killer rabbit’  
>>> if x == ‘roger’:  
... print “how’s jessica?”  
... elif x == ‘bugs’:  
... print “what’s up doc?”  
... else:  
... print ‘Run away! Run away!’  
...  
Run away! Run away!

378   
Глава 12. Условная инструкция if и синтаксические правила   
Эта многострочная инструкция простирается от строки if до конца блока else.   
При выполнении этой инструкции интерпретатор выполнит вложенные ин-  
струкции после той проверки, которая даст в результате истину, или блок else,   
если все проверки дадут ложный результат (в этом примере так и происходит).   
На практике обе части elif и else могут быть опущены, и в каждой части может   
иметься более одной вложенной инструкции. Обратите внимание, что связь   
слов if, elif и else определена тем, что они находятся на одной вертикальной   
линии, с одним и тем же отступом.  
Если вы знакомы с такими языками, как C или Pascal, вам будет интересно   
узнать, что в языке Python отсутствует инструкция switch или case, которая   
позволяет выбирать производимое действие на основе значения переменной.   
Вместо этого множественное ветвление оформляется либо в виде последова-  
тельности проверок if/elif, как в предыдущем примере, либо индексировани-  
ем словарей, либо поиском в списках. Поскольку словари и списки могут созда-  
ваться во время выполнения, они иногда способны обеспечить более высокую   
гибкость, чем жестко заданная логика инструкции if:  
>>> choice = ‘ham’  
>>> print {‘spam’: 1.25, # Инструкция ‘switch’ на базе словаря  
... ‘ham’: 1.99, # Используйте has\_key или get для   
... ‘eggs’: 0.99, # значения по умолчанию  
... ‘bacon’: 1.10}[choice]  
1.99  
Тем, кто такой прием видит впервые, может потребоваться некоторое время,   
чтобы осознать его� и тем не менее данный словарь обеспечивает множествен-  
ное ветвление через индексирование по ключу choice для выбора одного из не-  
скольких значений, почти так же, как это делает инструкция switch в языке C.   
Эквивалентная, но менее компактная инструкция if в языке Python выглядит,   
как показано ниже:  
>>> if choice == ‘spam’:  
... print 1.25  
... elif choice == ‘ham’:  
... print 1.99  
... elif choice == ‘eggs’:  
... print 0.99  
... elif choice == ‘bacon’:  
... print 1.10  
... else:  
... print ‘Bad choice’  
...  
1.99  
Обратите внимание на часть else, которая предназначена для обработки ситуа-  
ции, когда не найдено ни одного совпадения. Как было показано в главе 8, зна-  
чение по умолчанию при использовании словарей может быть получено с по-  
мощью оператора in, метода get или при перехвате исключения. Те же самые   
приемы могут использоваться и здесь – для определения действия по умолча-  
нию в случае реализации множественного ветвления на базе словаря. Ниже   
приводится пример с использование метода get для получения значения по   
умолчанию:  
>>> branch = {‘spam’: 1.25,  
... ‘ham’: 1.99,

Синтаксические правила языка Python   
379  
... ‘eggs’: 0.99}  
   
>>> print branch.get(‘spam’, ‘Bad choice’)  
1.25  
>>> print branch.get(‘bacon’, ‘Bad choice’)  
Bad choice  
Применение оператора in проверки на вхождение в инструкции if может обе-  
спечить получение того же результата по умолчанию:  
>>> choice = ‘bacon’  
>>> if choice in branch:  
... print(branch[choice])  
... else:  
... print(‘Bad choice’)  
...  
Bad choice  
Словари хорошо подходят для выбора значений, ассоциированных с ключами,   
но как быть в случае более сложных действий, которые можно запрограммиро-  
вать в инструкциях if? В четвертой части книги вы узнаете, что словари также   
могут содержать функции, выполняющие сложные действия при ветвлении,   
реализуя обычные таблицы переходов. В этом случае функции, играющие   
роль значений в словаре, часто создаются как лямбда-функции и вызываются   
добавлением круглых скобок. Подробнее об этом вы прочитаете в главе 19.  
Множественное ветвление на базе словаря довольно удобно использовать в про-  
граммах, которые имеют дело с динамическими данными, однако большин-  
ство программистов согласятся, что использование инструкции if является   
наиболее простым способом организации множественного ветвления. Обычно   
при колебаниях при выборе того или иного подхода предпочтение следует от-  
давать более простому и более удобочитаемому способу.  
Синтаксические правила языка Python  
Первое знакомство с синтаксической моделью языка Python состоялось в гла-  
ве 10. Теперь, когда мы подошли к таким крупным инструкциям, как if, наста-  
ло время пересмотреть и дополнить сведения о синтаксисе, введенные ранее.   
Вообще язык программирования Python обладает простым синтаксисом, осно-  
ванным на применении инструкций. Однако он обладает некоторыми особен-  
ностями, о которых вам необходимо знать:  
 •  
Инструкции выполняются последовательно, одна за другой, пока не будет   
предусмотрено что-то другое. Обычно интерпретатор выполняет инструк-  
ции в файле или в блоке от начала до конца, но такие инструкции, как if (и,   
как будет показано далее, циклы), заставляют интерпретатор выполнять пе-  
реходы внутри программного кода. Так как путь интерпретатора Python че-  
рез текст программы называется потоком управления, такие инструкции,   
как if, часто называются инструкциями управления потоком выполнения.  
 •  
Границы блоков и инструкций определяются автоматически. Как мы уже   
видели, в языке Python отсутствуют фигурные скобки или разделители   
«begin/end», окружающие блоки программного кода. Вместо этого принад-  
лежность инструкций к вложенному блоку определяется по величине от-  
ступов. Точно так же инструкции в языке Python обычно не завершаются

380   
Глава 12. Условная инструкция if и синтаксические правила   
точкой с запятой� обычно признаком конца инструкции служит конец стро-  
ки с этой инструкцией.  
 •  
Составные инструкции = “заголовок + «:» + инструкции с отступами”.   
Все составные инструкции в языке Python оформляются одинаково: стро-  
ка с заголовком завершается двоеточием, далее следуют одна или более   
вложенных инструкций, обычно с отступом относительно заголовка. Эти   
инструкции с отступами называются блоком (или иногда набором). В ин-  
струкции if предложения elif и else являются не только частями инструк-  
ции if, но и заголовками с собственными вложенными блоками.  
 •  
Пустые строки, пробелы и комментарии обычно игнорируются. Пустые   
строки игнорируются в файлах (но не в интерактивной оболочке, когда   
они завершают составные инструкции). Пробелы внутри инструкций и вы-  
ражений игнорируются практически всегда (за исключением строковых   
литералов, а также когда они используются для оформления отступов).   
Комментарии игнорируются всегда: они начинаются с символа # (не внутри   
строковых литералов) и простираются до конца строки.  
 •  
Строки документирования игнорируются, но сохраняются и отображают-  
ся специализированными инструментами. В языке Python поддержива-  
ется дополнительная форма комментариев, которая называется строками   
документирования, которые, в отличие от комментариев, начинающихся   
с #, сохраняются и доступны для просмотра во время выполнения. Строки   
документирования – это обычные строки, которые располагаются в начале   
файлов с программами и в некоторых инструкциях. Интерпретатор игно-  
рирует их содержимое, но они автоматически присоединяются к объектам   
во время выполнения и могут отображаться инструментами доступа к до-  
кументации. Строки документирования являются частью стратегии доку-  
ментирования в языке Python и будут рассматриваться в последней главе   
этой части книги.  
Как уже говорилось ранее, в языке Python отсутствует необходимость объ-  
являть типы переменных – только один этот факт упрощает синтаксис язы-  
ка больше, чем любые другие особенности. Но для большинства новых поль-  
зователей самой необычной особенностью синтаксиса языка Python кажется   
отсутствие фигурных скобок и точек с запятой, используемых в качестве раз-  
делителей блоков и инструкций во многих других языках, поэтому давайте   
рассмотрим их поближе.  
Разделители блоков: правила оформления отступов  
Интерпретатор автоматически определяет границы блоков по величине от-  
ступов – то есть по ширине пустого пространства слева от программного кода.   
Все инструкции, смещенные вправо на одинаковое расстояние, принадлежат   
к одному и тому же блоку кода. Другими словами, инструкции в блоке выстро-  
ены по вертикальной линии. Блок заканчивается либо с концом файла, либо   
как только встретится строка с меньшим отступом� более глубоко вложенные   
блоки имеют более широкие отступы, чем инструкции в объемлющем блоке.  
Например, на рис. 12.1 показана структура блоков следующего фрагмента про-  
граммного кода:  
x = 1  
if x:

Синтаксические правила языка Python   
381  
 y = 2  
 if y:  
 print ‘block2’  
 print ‘block1’  
print ‘block0’  
Заголовок:  
Заголовок:  
Блок 0  
Блок 0  
Блок 2  
Блок 1  
Блок 1  
Рис. 12.1. Вложенные блоки программного кода: вложенный блок начинается   
с инструкции, имеющей больший отступ, и заканчивается либо когда будет   
встречена инструкция с меньшим отступом, либо в конце файла  
Данный фрагмент содержит три блока: первый (программный код верхнего   
уровня) вообще не имеет отступов, второй (внутри внешней инструкции if)   
имеет отступ из четырех пробелов и третий (инструкция print внутри вложен-  
ной инструкции if) имеет отступ из восьми пробелов.  
Вообще программный код верхнего уровня (не вложенный) должен начинаться   
в строках с позиции 1. Вложенные блоки могут начинаться с любой позиции –   
отступ может состоять из любого числа пробелов и символов табуляции, глав-  
ное, чтобы все инструкции в одном блоке имели одинаковые отступы. То есть   
для интерпретатора не важно, как будут оформляться отступы, главное, чтобы   
оформление было непротиворечивым. Наиболее часто для смещения на один   
уровень используются четыре пробела или один символ табуляции, но в этом   
отношении не существует никаких стандартов.  
Правила оформления отступов в программном коде являются вполне есте-  
ственными. Например, в следующем фрагменте демонстрируются типичные   
ошибки, связанные с оформлением отступов:  
 x = ‘SPAM’ # Ошибка: отступ в первой строке  
if ‘rubbery’ in ‘shrubbery’:  
 print(x \* 8)  
 x += ‘NI’ # Ошибка: неуместный отступ  
 if x.endswith(‘NI’):  
 x \*= 2  
 print(x) # Ошибка: непоследовательное оформление отступов  
Правильно оформленная версия этого фрагмента приводится ниже – даже в та-  
ком простом примере, как этот, правильное оформление отступов делает про-  
граммный код намного более удобочитаемым:

382   
Глава 12. Условная инструкция if и синтаксические правила   
x = ‘SPAM’  
if ‘rubbery’ in ‘shrubbery’:  
 print(x \* 8)  
 x += ‘NI’  
 if x.endswith(‘NI’):  
 x \*= 2  
 print(x) # Выведет “SPAMNISPAMNI”  
Единственное, где в языке Python пробелы имеют большое значение, – это ког-  
да они находятся левее программного кода� в большей части других случаев   
неважно, есть пробелы или нет. При этом отступы в действительности явля-  
ются частью синтаксиса языка, а не просто предложением по оформлению: все   
инструкции внутри любого заданного блока должны иметь одинаковые отсту-  
пы, в противном случае интерпретатор сообщит о синтаксической ошибке. Т.к.   
вам не требуется явно отмечать начало и конец вложенного блока, различные   
синтаксические элементы, которые можно найти в других языках, в языке   
Python не нужны.  
Как уже отмечалось в главе 10, отступы, как часть синтаксической модели, ста-  
ли важным компонентом обеспечения удобочитаемости в языках структурного   
программирования, таких как Python. Иногда синтаксис языка Python опи-  
Python опи-  
 опи-  
сывают фразой «what you see is what you get» (что видишь, то и получаешь) –   
отступ каждой строки однозначно говорит, к какому блоку она принадлежит.   
Такой непротиворечивый внешний вид программного кода на языке Python   
обеспечивает простоту его сопровождения и многократного использования.  
Отступы выглядят в программном коде куда более естественно, чем можно   
было бы подумать, и они обеспечивают отражение его логической структуры.   
Программный код, в котором отступы оформлены непротиворечивым образом,   
всегда будет соответствовать правилам языка Python. Кроме того, многие тек-  
стовые редакторы (включая IDLE) упрощают следование модели отступов, ав-  
томатически выравнивая программный код по мере его ввода.  
Не смешивайте пробелы и символы табуляции:   
новый механизм проверки ошибок в версии 3.0  
Для оформления отступов допускается использовать пробелы и символы та-  
буляции, но обычно не принято смешивать их внутри блока – используйте   
что-то одно – или пробелы, или символы табуляции. С технической точки   
зрения считается, что символ табуляции смещает текущую позицию в строке   
до ближайшей позиции с номером, кратным 8, и программный код будет ин-  
терпретироваться безошибочно, если пробелы и символы табуляции смеши-  
ваются непротиворечивым способом. Однако такой программный код будет   
достаточно сложно изменять. Хуже того, смешивание пробелов и символов   
табуляции ухудшают удобочитаемость программного кода – символы табуля-  
ции могут отображаться в текстовом редакторе другого программиста совсем   
не так, как у вас.   
В действительности, по только что описанным причинам, интерпретатор Py-  
Py-  
thon 3.0 считает ошибкой непоследовательное смешивание пробелов и симво-  
 3.0 считает ошибкой непоследовательное смешивание пробелов и симво-  
лов табуляции в пределах блока (то есть, когда величина отступов из смешан-  
ных символов в пределах блока может отличаться в зависимости от интерпре-  
тации ширины символов табуляции). В Python 2.6 допускается подобное сме-  
Python 2.6 допускается подобное сме-  
 2.6 допускается подобное сме-  
шивание. Однако эта версия интерпретатора имеет ключ -t командной строки,   
при использовании которого интерпретатор будет предупреждать о непосле-

Синтаксические правила языка Python   
383  
довательном использовании символов табуляции, а при запуске с ключом -tt   
он будет считать ошибкой такое оформление программного кода (вы можете   
указывать эти ключи в командной строке при запуске своего сценария, напри-  
мер: python –t main.py). Режим работы интерпретатора Python 3.0 эквивалентен   
запуску интерпретатора версии 2.6 с ключом -tt.  
Разделители инструкций:   
строки и многострочные инструкции   
В языке Python инструкция обычно заканчивается в конце строки. Однако   
если инструкция слишком велика, чтобы уместиться в одной строке, можно   
использовать следующие специальные правила размещения инструкции в не-  
скольких строках:  
 •  
Инструкции могут располагаться в нескольких строках, если они окруже-  
ны синтаксической парой скобок. Язык Python позволяет продолжить ввод   
инструкции в следующей строке, когда содержимое инструкции заключено   
в пару скобок (), {} или []. Примерами инструкций, которые могут распола-  
гаться в нескольких строках, могут служить выражения в круглых скоб-  
ках, литералы словарей и списков – инструкция не считается законченной,   
пока интерпретатор Python не встретит строку с закрывающей скобкой (), }   
или ]). Промежуточные строки (вторая и последующие строки инструкции)   
могут иметь любые отступы, но желательно, чтобы вы обеспечили одина-  
ковое выравнивание по вертикали для повышения удобочитаемости, если   
это возможно. Это правило относится также к генераторам словарей и мно-  
жеств в Python 3.0.  
 •  
Инструкции могут располагаться в нескольких строках, если они завер-  
шаются символом обратного слеша. Это несколько устаревшая особен-  
ность, но если необходимо разместить инструкцию в нескольких строках,   
можно в конец каждой предшествующей строки вставить символ обратного   
слеша, который будет служить признаком, что инструкция продолжается   
на следующей строке. Так как существует возможность использовать кру-  
глые скобки для заключения длинных конструкций, символы обратного   
слеша практически никогда не используются. При использовании такого   
подхода легко допустить ошибку: обычно интерпретатор замечает отсут-  
ствие символа \ и выводит сообщение об ошибке, но если текущая и сле-  
дующая строки могут интерпретироваться, как самостоятельные и незави-  
симые инструкции, ошибка не будет замечена, а результат может оказаться   
непредсказуемым.  
 •  
Литералы строк в тройных кавычках могут располагаться в нескольких   
строках. Очень длинные строковые литералы можно разместить в несколь-  
ких строках – блоки строк в тройных кавычках, с которыми мы встрети-  
лись в главе 7, предназначены именно для этих целей. Там же, в главе 7, мы   
узнали, что к строковым литералам применяется неявная операция кон-  
катенации – в соответствии с правилом скобок, упоминавшимся выше, ис-  
пользование круглых скобок позволит расположить несколько строковых   
литералов в разных строках.  
 •  
Другие правила. Существует еще несколько моментов, которые хотелось бы   
упомянуть. Хотя это и используется редко, инструкции можно завершать   
точкой с запятой – иногда это соглашение применяется, чтобы компактно   
разместить несколько инструкций в одной строке. Кроме того, в любом ме-

384   
Глава 12. Условная инструкция if и синтаксические правила   
сте в файле могут присутствовать пустые строки и комментарии. Коммента-  
рии (которые начинаются с символа #) простираются до конца строки.  
Несколько специальных случаев  
Ниже показано, как выглядит инструкция при использовании правила ис-  
пользования скобок. Конструкции, заключенные в скобки, могут занимать   
произвольное число строк:  
L = [“Good”,  
 “Bad”,  
 “Ugly”] # Пара скобок может охватывать несколько строк  
Этот прием также можно использовать с круглыми скобками (выражения, ар-  
гументы функций, заголовки функций, кортежи и выражения-генераторы)   
и с фигурными скобками (словари, литералы множеств в версии 3.0, а также   
генераторы множеств и словарей). С некоторыми из этих инструментов мы по-  
знакомимся в следующих главах, однако это правило охватывает большинство   
конструкций, которые могут располагаться в нескольких строках.  
При желании можно использовать символы обратного слеша, но этот прием   
редко используется на практике:  
if a == b and c == d and \  
 d == e and f == g:  
 print(‘olde’) # Символы обратного слеша позволяют продолжить...  
Поскольку любое выражение можно заключить в круглые скобки, то лучше   
использовать их, когда возникает необходимость расположить инструкцию   
в нескольких строках:  
if (a == b and c == d and  
 d == e and e == f):  
 print(‘new’) # Но круглые скобки позволяют то же самое  
На практике символы обратного слеша к использованию не рекомендуются,   
потому что они малозаметны и их легко пропустить по случайности. В следую-  
щем примере задача состояла в присваивании переменной x значения 10, если   
учесть наличие символа обратного слеша. Однако если обратный слеш случай-  
но опустить, переменной x будет присвоено значение 6� при этом никаких со-  
общений об ошибке не появится (+4 – это допустимая инструкция выражения).  
В действующих программах подобная случайность в более сложных инструк-  
циях присваивания могла бы привести к весьма неприятным проблемам:1  
x = 1 + 2 + 3 \ # Отсутствие \ существенно изменяет смысл выражения  
+4  
1   
Откровенно говоря, меня очень удивило, что возможность использования символа   
обратного слеша таким способом не была ликвидирована в Python 3.0, особенно если   
учесть, насколько грандиозны были некоторые из изменений! (В табл. П.2, в преди-  
словии, приводится список возможностей, которые были убраны в версии 3.0, – не-  
которые из них кажутся весьма безобидными по сравнению с опасностями, которые   
таит в себе такое использование символа обратного слеша.) С другой стороны, цель   
этой книги – изучение языка Python, а не популистские разглагольствования, поэто-  
му я могу только посоветовать: не используйте эту возможность.

Проверка истинности   
385  
Еще один специальный случай: в языке Python допускается записывать   
в одной строке несколько несоставных инструкций (то есть инструкций, кото-  
рые не имеют вложенных инструкций), разделяя их точками с запятой. Неко-  
торые программисты используют эту возможность для экономии пространства   
в файле, однако удобочитаемость будет выше, если в каждой строке размещать   
только одну инструкцию:  
x = 1; y = 2; print(x) # Несколько простых инструкций в одной строке  
Как мы узнали в главе 7, строковые литералы в тройных кавычках также мо-  
гут занимать несколько строк. Кроме того, если два строковых литерала следу-  
ют друг за другом, они объединяются, как если бы между ними стоял оператор   
конкатенации +. Согласно правилу скобок, если несколько строковых лите-  
ралов окружить круглыми скобками, их можно будет расположить в разных   
строках. В первом примере ниже между строками будет вставлен символ конца   
строки и переменной S будет присвоен результат ‘\naaaa\nbbbb \ncccc’, а во вто-  
ром примере будет выполнена неявная операция конкатенации и переменной   
S будет присвоена строка ‘aaaabbbbcccc’. Во втором случае комментарии будут   
игнорироваться, а в первом они станут частью строки S:  
S = “””  
aaaa  
bbbb  
cccc”””  
   
S = (‘aaaa’  
 ‘bbbb’ # Этот комментарий игнорируется  
 ‘cccc’)  
И наконец, Python позволяет располагать тело составной инструкции в одной   
строке с заголовком при условии, что тело образует простая (несоставная) ин-  
струкция. Вам часто придется видеть следующий вариант использования про-  
стых инструкций с единственным условием и действием:  
if 1: print(‘hello’) # Простая инструкция в строке заголовка  
Эти специальные случаи можно комбинировать между собой, чтобы писать   
программный код, который будет сложно читать, но я не рекомендую посту-  
пать так – старайтесь записывать по одной инструкции в строке и выравнивай-  
те все блоки� исключение могут составлять только простейшие случаи. Когда   
вам придется вернуться к своей программе спустя полгода, вы будете рады, что   
поступали именно так.  
Проверка истинности  
Понятия сравнения, равенства и значений истинности были введены в главе 9.   
Инструкция if – это первая инструкция на нашем пути, которая использует   
результаты проверки, поэтому здесь мы подробнее поговорим о некоторых из   
этих идей. В частности, о том, что логические операторы в языке Python не-  
сколько отличаются от аналогичных операторов в таких языках, как C. В язы-  
ке Python:  
 •  
Любое число, не равное нулю, или непустой объект интерпретируется как   
истина.

386   
Глава 12. Условная инструкция if и синтаксические правила   
 •  
Числа, равные нулю, пустые объекты и специальный объект None интерпре-  
тируются как ложь.  
 •  
Операции сравнения и проверки на равенство применяются к структурам   
данных рекурсивно.  
 •  
Операции сравнения и проверки на равенство возвращают значение True   
или False (которые представляют собой версии чисел 1 и 0).  
 •  
Логические операторы and и or возвращают истинный или ложный объект-  
операнд.   
В двух словах, логические операторы используются для объединения резуль-  
татов других проверок. В языке Python существует три логических оператора:  
X and Y  
Истина, если оба значения X и Y истинны.  
X or Y  
Истина, если любое из значений X или Y истинно.  
not X  
Истина, значение X ложно (выражение возвращает значение True или False)  
Здесь X и Y могут быть любыми значениями истинности или выражениями,   
которые возвращают значения истинности (например, выражение проверки   
на равенство, сравнение с диапазоном значений и так далее). В языке Python   
логические операторы имеют вид слов (вместо обозначений &&, || и !, как это   
реализовано в языке C). Кроме того, логические операторы and и or возвращают   
истинный или ложный объект, а не значение True или False. Рассмотрим не-  
сколько примеров, чтобы понять, как они работают:  
>>> 2 < 3, 3 < 2 # Меньше чем: возвращает True или False (1 или 0)  
(True, False)  
Операция сравнения величин, как в данном случае, возвращает в качестве ре-  
зультата значение True или False, которые, как мы узнали в главах 5 и 9, в дей-  
ствительности являются особыми версиями целых чисел 1 и 0 (выводятся они   
особым образом, а во всем остальном являются обычными числами).  
С другой стороны, операторы and и or всегда возвращают объект – объект либо   
слева от оператора, либо справа. Если действие этих операторов проверяется   
инструкцией if или другими инструкциями, они будут иметь ожидаемый ре-  
зультат (не забывайте, что каждый объект может интерпретироваться как ис-  
тина или как ложь), но это не будут простые значения True или False.  
В случае оператора or интерпретатор начинает вычислять значения объектов-  
операндов слева направо и возвращает первый, имеющий истинное значение.   
Кроме того, интерпретатор прекратит дальнейшие вычисления, как только бу-  
дет найден первый объект, имеющий истинное значение. Это обычно называют   
вычислением по короткой схеме, так как конечный результат становится из-  
вестен еще до вычисления остальной части выражения:  
>>> 2 or 3, 3 or 2 # Вернет левый операнд, если он имеет истинное значение  
(2, 3) # Иначе вернет правый операнд (истинный или ложный)  
>>> [] or 3  
3  
>>> [] or {}  
{}

Трехместное выражение if/else   
387  
В первой строке предыдущего примера оба операнда (2 и 3) имеют истинные (то   
есть ненулевые) значения, поэтому интерпретатор всегда будет останавливать   
вычисления и возвращать операнд слева. В других двух операциях левый опе-  
ранд имеет ложное значение (пустой объект), поэтому интерпретатор просто   
вычисляет и возвращает объект справа (который может иметь как истинное,   
так и ложное значение).  
Вычисление оператора and также останавливается, как только результат ста-  
нет известен, однако в этом случае интерпретатор вычисляет операнды слева   
направо и возвращает первый объект, имеющий ложное значение:   
>>> 2 and 3, 3 and 2 # Вернет левый операнд, если он имеет ложное значение  
(3, 2) # Иначе вернет правый операнд (истинный или ложный)  
>>> [] and {}  
[]  
>>> 3 and []  
[]  
Здесь в первой строке оба операнда имеют истинные значения, поэтому интер-  
претатор вычислит оба операнда и вернет объект справа. Во второй проверке   
левый операнд имеет ложное значение ([]), поэтому интерпретатор останавли-  
вает вычисления и возвращает его в качестве результата проверки. В последней   
проверке левый операнд имеет истинное значение (3), поэтому интерпретатор   
вычисляет и возвращает объект справа (который имеет ложное значение []).  
Конечный результат будет тот же, что и в языке C и во многих других языках, –   
вы получаете значение, которое логически интерпретируется как истина или   
ложь при использовании в инструкции if или while. Однако в языке Python   
логические операторы возвращают либо левый, либо правый объект, а не про-  
стое целочисленное значение.  
Такое поведение операторов and и or может показаться странным на первый   
взгляд, поэтому загляните во врезку «Придется держать в уме: логические   
значения», где вы найдете примеры, как иногда эта особенность может исполь-  
зоваться программистами на языке Python. Кроме того, в следующем разделе   
демонстрируются часто встречающиеся способы использования такого поведе-  
ния операторов и их замена в более свежих версиях Python.  
Трехместное выражение if/else  
Одна из основных ролей логических операторов в языке Python заключается   
в образовании выражений, которые выполняются так же, как условная ин-  
струкция if. Рассмотрим следующую инструкцию, которая записывает в A зна-  
чение Y или Z, в зависимости от истинности значения X:  
if X:  
 A = Y  
else:  
 A = Z  
Иногда, как в данном примере, элементы инструкции настолько просты, что   
кажется излишеством тратить на них четыре строки. В некоторых случаях   
у нас может появиться желание вложить такую конструкцию внутрь дру-  
гой инструкции вместо того, чтобы выполнять присваивание переменной. По   
этим причинам (и, откровенно говоря, потому что в языке C имеется похожая

388   
Глава 12. Условная инструкция if и синтаксические правила   
возможность)1 в версии Python 2.5 появилась новая конструкция, позволяю-  
щая записать те же действия в виде единственного выражения:  
A = Y if X else Z  
Данное выражение дает тот же результат, что и предыдущая четырехстроч-  
ная инструкция if, но выглядит она проще. Как и в предыдущей инструкции,   
интерпретатор выполняет выражение Y, только если объект X имеет истинное   
значение, а выражение Z выполняется, только если объект X имеет ложное зна-  
чение. То есть вычисления здесь также выполняются по сокращенной схеме.   
Ниже приводятся несколько примеров выражения в действии:   
>>> A = ‘t’ if ‘spam’ else ‘f’ # Непустая строка – это истина  
>>> A  
‘t’  
>>> A = ‘t’ if ‘’ else ‘f’  
>>> A  
‘f’  
До версии Python 2.5 (да и после) тот же эффект можно было получить за счет   
комбинирования операторов and и or благодаря тому, что они возвращают объ-  
ект слева или справа:  
A = ((X and Y) or Z)  
Этот прием работает, но он скрывает в себе ловушку – он предполагает, что   
Y будет иметь истинное значение. В этом случае эффект будет тот же самый:   
оператор and выполнится первым и вернет Y, если X имеет истинное значение.   
В противном случае оператор or просто вернет Z. Другими словами, мы получа-  
ем: «if X then Y else Z».  
Эта комбинация операторов and/or требует некоторого времени, чтобы осознать   
ее при первом знакомстве, но, начиная с версии 2.5, надобность в таком приеме   
отпала, так как существует более наглядная конструкция Y if X else Z, которую   
можно использовать в качестве выражения. Если же составляющие достаточно   
сложны, лучше использовать полноценную инструкцию if.  
В качестве дополнительного примечания: в языке Python следующее выраже-  
Python следующее выраже-  
 следующее выраже-  
ние дает похожий эффект – благодаря тому, что функция bool преобразует X   
в соответствующее целое число 1 или 0, которое затем используется для выбора   
требуемого значения из списка:  
A = [Z, Y][bool(X)]  
Например:  
>>> [‘f’, ‘t’][bool(‘’)]  
‘f’  
>>> [‘f’, ‘t’][bool(‘spam’)]  
‘t’  
1   
В действительности, порядок следования операндов в выражении X if Y else Z в языке   
Python несколько отличается от выражения Y ? X : Z в языке C. Такой порядок был   
выбран в соответствии с анализом наиболее распространенных шаблонов программи-  
рования в языке Python, а также отчасти, чтобы помочь бывшим программистам на   
языке C избавиться от привычки злоупотреблять этим выражением. Не забывайте,   
что в языке Python исповедуется принцип – чем проще, тем лучше.

Трехместное выражение if/else   
389  
Однако это далеко не то же самое, потому что в данном случае интерпретатор   
не использует сокращенную схему вычисления – он всегда будет вычислять оба   
значения Z и Y, независимо от значения X. Из-за всех этих сложностей лучше   
использовать более простое и более понятное выражение if/else, появившееся   
в версии Python 2.5. Кроме того, этим выражением не следует злоупотреблять   
и следует использовать его, только если части выражения достаточно просты,   
в противном случае лучше использовать обычную инструкцию if, что облегчит   
модификацию программного кода в будущем. Ваши коллеги будут благодарны   
вам за это.  
Однако вы по-прежнему можете встретить конструкцию на основе комбина-  
ции and/or в программном коде, написанном до появления версии Python 2.5 (и   
в программном коде, написанном программистами, использовавшими язык C,   
и не сумевшими избавиться от прошлых привычек).  
Придется держать в уме: логические значения  
В языке Python часто используется прием выбора одного объекта из   
множества, основанный на необычном поведении логических операто-  
ров. Следующая инструкция:  
X = A or B or C or None  
присвоит переменной X первый непустой (имеющий истинное значение)   
объект из множества объектов A, B и C или None, если все эти объекты ока-  
жутся пустыми. Этот прием стал возможен благодаря тому, что опера-  
тор or возвращает один из двух объектов и, как оказывается, это весьма   
распространенная парадигма программирования на языке Python: что-  
бы выбрать непустой объект из фиксированного множества, достаточно   
просто объединить их в выражение с помощью оператора or. В простей-  
шем случае эту особенность можно использовать для назначения зна-  
чения по умолчанию – следующая инструкция присвоит переменной   
X значение переменной A, если оно истинно (или непустое), и default –   
в противном случае:  
X = A or default  
Также важно осознать, как выполняются вычисления по сокращенной   
схеме, потому что справа от логического оператора может находиться   
функция, выполняющая важную работу или оказывающая побочное   
влияние, вызов которой не произойдет из-за действия правила вычисле-  
ния по сокращенной схеме:  
if f1() or f2(): ...  
В данном случае, если функция f1 вернет истинное (или непустое) зна-  
чение, интерпретатор никогда не вызовет функцию f2. Чтобы гаранти-  
ровать вызов обеих функций, можно вызвать их до применения опера-  
тора or:  
tmp1, tmp2 = f1(), f2()  
if tmp1 or tmp2: ...

390   
Глава 12. Условная инструкция if и синтаксические правила   
Вы уже видели другой вариант использования такого поведения: благо-  
даря особенностям работы логических операторов выражение ((A and B)   
or C) может использоваться для достаточно близкой (смотрите обсужде-  
ние этой инструкции выше) имитации инструкции if/else.  
Другие случаи логических значений мы уже видели в предыдущих гла-  
вах. Как мы видели в главе 9, вследствие того, что все объекты могут   
расцениваться как истинные или ложные значения, в языке Python лег-  
ко и просто выполнить проверку объекта напрямую (if X:) вместо того,   
чтобы сравнивать его с пустым значением (if X != ‘’:). В случае строк эти   
две проверки равнозначны. Мы также узнали в главе 5, что логические   
значения True и False являются обычными целыми числами 1 и 0 и могут   
использоваться для инициализации переменных (X = False), в условных   
выражениях циклов (while True:) и для отображения результатов в инте-  
рактивном сеансе.  
Кроме того, в шестой части, при обсуждении темы перегрузки операто-  
ров мы увидим, что в определениях новых типов объектов с помощью   
классов имеется возможность определять их логическую природу с по-  
мощью методов \_\_bool\_\_ и \_\_len\_\_ (метод \_\_bool\_\_ в версии 2.6 носит имя   
\_\_nonzero\_\_). Если первый метод отсутствует в определении класса, для   
проверки истинности объекта используется второй метод – если возвра-  
щаемая длина равна нулю, объект считается ложным, так как пустые   
объекты всегда считаются ложными.  
В заключение  
В этой главе мы познакомились с инструкцией if языка Python. Так как это   
была первая составная инструкция на нашем пути, мы попутно рассмотрели   
общие синтаксические правила языка Python и поближе познакомились с опе-  
Python и поближе познакомились с опе-  
 и поближе познакомились с опе-  
рациями проверки истинности. Кроме того, мы также узнали, как в языке   
Python реализуется множественное ветвление, и изучили форму выражений   
if/else, которая впервые появилась в версии Python 2.5.  
В следующей главе мы продолжим изучение процедурных инструкций и рас-  
смотрим циклы while и for. Там будет рассказано об альтернативных способах   
программирования циклов в языке Python, каждый из которых имеет свои   
преимущества. Но перед этим вас ждут обычные контрольные вопросы главы.  
Закрепление пройденного  
Контрольные вопросы  
1. Как в языке Python можно оформить множественное ветвление?  
2. Как в языке Python можно оформить инструкцию if/else в виде выраже-  
ния?  
3. Как можно разместить одну инструкцию в нескольких строках?  
4. Что означают слова True и False?

Закрепление пройденного   
391  
Ответы  
1. Самый простой, хотя и не самый краткий способ организации множествен-  
ного ветвления заключается в использовании инструкции if с нескольки-  
ми частями elif. Нередко того же результата можно добиться с помощью   
операции индексирования в словаре, особенно если учесть, что словари   
могут содержать функции, созданные с помощью инструкций def или вы-  
ражений lambda.  
2. В версии Python 2.5 выражение Y if X else Z возвращает Y, если X имеет ис-  
тинное значение, или Z – в противном случае. Это выражение эквивалентно   
инструкции if из четырех строк. Комбинация операторов and/or ((X and Y)   
or Z) может действовать точно так же, но она менее понятна и требует, чтобы   
часть Y имела истинное значение.  
3. Обернув инструкцию синтаксически уместной парой скобок ((), [], или {}),   
можно расположить ее на нескольких строках – инструкция будет считать-  
ся законченной, когда интерпретатор обнаружит правую закрывающую   
скобку. Строки инструкции, со второй и далее, могут иметь любые отступы.  
4. True и False – это всего лишь версии целых чисел 1 и 0 соответственно. В язы-  
ке Python они обозначают истинные и ложные значения. Они могут исполь-  
зоваться в операциях проверки истинности и для инициализации пере-  
менных, а также выводиться как результаты выражений в интерактивном   
сеансе.

Глава 13.  
   
Циклы while и for  
В этой главе мы встретимся с двумя основными конструкциями организации   
циклов в языке Python – инструкциями, которые выполняют одну и ту же по-  
следовательность действий снова и снова. Первая из них, инструкция while,   
обеспечивает способ организации универсальных циклов� вторая, инструкция   
for, предназначена для обхода элементов в последовательностях и выполнения   
блока программного кода для каждого из них.  
Ранее мы уже встречались с этими инструкциями, но в этой главе мы получим   
более полное представление о них. Кроме того, мы попутно рассмотрим неко-  
торые менее известные инструкции, используемые в циклах, такие как break   
и continue, и встроенные функции, такие как range, zip и map, которые исполь-  
зуются вместе с циклами.  
В языке Python существуют и другие способы организации циклов, но ин-  
струкции while и for, которые описываются здесь, являются основными син-  
таксическими элементами, предоставляющими возможность программирова-  
ния повторяющихся действий. Тема итераций будет продолжена в следующей   
главе, где мы исследуем родственные концепции протокола итераций в языке   
Python (используется инструкцией for). В последующих главах мы займемся   
исследованием еще более экзотических инструментов организации циклов,   
таких как генераторы, функции filter и reduce. Но пока начнем с самого про-  
стого.  
Циклы while  
Инструкция while является самой универсальной конструкцией организации   
итераций в языке Python. Проще говоря, она продолжает выполнять блок ин-  
струкций (обычно с отступами) до тех пор, пока условное выражение продол-  
жает возвращать истину. Она называется «циклом», потому что управление   
циклически возвращается к началу инструкции, пока условное выражение   
не вернет ложное значение. Как только в результате проверки будет получено   
ложное значение, управление будет передано первой инструкции, расположен-  
ной сразу же за вложенным блоком тела цикла while. В результате тело цикла   
продолжает выполняться снова и снова, пока условное выражение возвращает

Циклы while   
393  
истинное выражение, а если условное выражение сразу вернет ложное значе-  
ние, тело цикла никогда не будет выполнено.  
Общий формат  
В своей наиболее сложной форме инструкция while состоит из строки заголовка   
с условным выражением, тела цикла, содержащего одну или более инструк-  
ций с отступами, и необязательной части else, которая выполняется, когда   
управление передается за пределы цикла без использования инструкции break.   
Интерпретатор продолжает вычислять условное выражение в строке заголовка   
и выполнять вложенные инструкции в теле цикла, пока условное выражение   
не вернет ложное значение:  
while <test>: # Условное выражение test  
 <statements1> # Тело цикла  
else: # Необязательная часть else  
 <statements2> # Выполняется, если выход из цикла производится не   
 # инструкцией break  
Примеры  
Для иллюстрации рассмотрим несколько простых циклов while в действии.   
Первый, который содержит инструкцию print, вложенную в цикл while, просто   
выводит сообщение до бесконечности. Не забывайте, что True – это всего лишь   
особая версия целого числа 1, и оно обозначает истинное значение, поэтому   
результатом этого условного выражения всегда будет истина, и интерпретатор   
бесконечно будет выполнять тело цикла, пока вы не прервете его выполнение.   
Такие циклы обычно называются бесконечными:  
>>> while True:  
... print(‘Type Ctrl-C to stop me!’)  
Следующий фрагмент продолжает вырезать из строки первый символ, пока   
она не опустеет и в результате не превратится в ложное значение. Это обыч-  
ная практика – проверка истинности объектов осуществляется непосредствен-  
но вместо использования более растянутого эквивалента (while x != ‘’:). Далее   
в этой главе мы рассмотрим другие способы обхода элементов строки с помо-  
щью цикла for.   
>>> x = ‘spam’  
>>> while x: # Пока x не пустая строка  
... print(x, end=’ ‘)  
... x = x[1:] # Вырезать первый символ из x  
...  
spam pam am m  
Обратите внимание на именованный аргумент end=’ ‘, который обеспечивает   
вывод значений в одну строку через пробел. Если вы забыли, чем это объясня-  
ется, обращайтесь к главе 11. Следующий фрагмент перебирает значения от   
a до b, не включая значение b. Ниже мы рассмотрим более простой способ вы-  
полнения этих же действий с помощью цикла for и встроенной функции range:  
>>> a=0; b=10  
>>> while a < b: # Один из способов организации циклов перечисления  
... print(a, end=’ ‘)  
... a += 1 # Или, a = a + 1

394   
Глава 13. Циклы while и for   
...  
0 1 2 3 4 5 6 7 8 9  
Наконец, обратите внимание, что в языке Python отсутствует цикл «do until»,   
имеющийся в других языках программирования. Однако его можно имитиро-  
вать, добавив в конец тела цикла условную инструкцию и инструкцию break:  
while True:  
 ...тело цикла...  
 if exitTest(): break  
Чтобы окончательно понять, как эта структура работает, нам необходимо пе-  
рейти к следующему разделу и поближе познакомиться с инструкцией break.  
break, continue, pass и else  
Теперь, когда мы познакомились с несколькими циклами в действии, наста-  
ло время обратить внимание на две простые инструкции, которые могут ис-  
пользоваться только внутри циклов – инструкции break и continue. Раз уж мы   
занялись изучением необычных инструкций, заодно рассмотрим здесь часть   
else, потому что она некоторым образом связана с инструкцией break, и заодно   
пустую инструкцию-заполнитель pass (которая не имеет прямого отношения   
к циклам, а относится к категории простых инструкций, состоящих из одного   
слова). В языке Python:  
break  
Производит переход за пределы объемлющего цикла (всей инструкции цик-  
ла).  
continue  
Производит переход в начало цикла (в строку заголовка).  
pass  
Ничего не делает: это пустая инструкция, используемая как заполнитель.  
Блок else  
Выполняется, только если цикл завершился обычным образом (без исполь-  
зования инструкции break).  
Общий формат цикла  
С учетом инструкций break и continue цикл while в общем виде выглядит, как   
показано ниже:  
while <test1>:  
 <statements1>  
 if <test2>: break # Выйти из цикла, пропустив часть else  
 if <test3>: continue # Перейти в начало цикла, к выражению test1  
else:  
 <statements2> # Выполняется, если не была использована   
 # инструкция ‘break’  
Инструкции break и continue могут появляться в любом месте внутри тела цик-  
ла while (или for), но как правило, они используются в условных инструкциях   
if, чтобы выполнить необходимое действие в ответ на некоторое условие.

break, continue, pass и else   
395  
Обратимся к нескольким простым примерам, чтобы увидеть, как эти инструк-  
ции используются на практике.  
pass  
Инструкция pass не выполняет никаких действий и используется в случаях,   
когда синтаксис языка требует наличия инструкции, но никаких полезных   
действий в этой точке программы выполнить нельзя. Она часто используется   
в качестве пустого тела составной инструкции. Например, создать бесконеч-  
ный цикл, который ничего не делает, можно следующим образом:  
while 1: pass # Нажмите Ctrl-C, чтобы прервать цикл!  
Поскольку тело цикла – это всего лишь пустая инструкция, интерпретатор   
«застрянет» в этом цикле. Грубо говоря, pass в мире инструкций – это то же,   
что None в мире объектов, – явное ничто. Обратите внимание, что тело этого   
цикла while находится в той же строке, что и заголовок, после двоеточия. Как   
и в случае с инструкцией if, такой прием можно использовать только в случае,   
когда тело цикла образует несоставная инструкция.  
Этот пример вечно делает «ничто». Вероятно, это самая бесполезная програм-  
ма (если только вы не хотите погреться у своего ноутбука в холодный зимний   
день!), которая когда-либо была написана на языке Python, и, тем не менее,   
я не смог придумать лучший пример применения инструкции pass.   
Далее мы увидим, где эта инструкция может использоваться с большим смыс-  
лом, например, для того, чтобы игнорировать исключение в инструкции try,   
или для определения пустых классов, реализующих объекты, которые ведут   
себя подобно структурам и записям в других языках. Иногда инструкция pass   
используется как заполнитель, вместо того, «что будет написано позднее»,   
и в качестве временного фиктивного тела функций:  
def func1():  
 pass # Реализация функции будет добавлена позже  
   
def func2():  
 pass  
Задание пустого тела функции вызовет синтаксическую ошибку, поэтому в по-  
добных ситуациях можно использовать инструкцию pass.  
Примечание,  касающееся  различий  между  версиями: В версии   
Python 3.0 (но не в 2.6) вместо любого выражения допускается   
использовать многоточие ... (буквально, три точки, следующие   
друг за другом). Многоточие само по себе не выполняет никаких   
действий, поэтому его можно использовать как альтернативу   
инструкции pass, в частности вместо программного кода, кото-  
рый будет написан позднее, – своего рода примечание «TBD» (To   
Be Done – подлежит реализации) на языке Python:  
def func1():  
 ... # Альтернатива инструкции pass  
def func2():  
 ...  
   
func1() # При вызове не выполняет никаких действий

396   
Глава 13. Циклы while и for   
Многоточие может также присутствовать в одной строке с за-  
головком инструкции и использоваться для инициализации   
переменных, когда не требуется указывать значение какого-то   
определенного типа:   
def func1(): ... # Может также присутствовать в той же строке  
def func2(): ...  
   
>>> X = ... # Альтернатива объекту None  
>>> X  
Ellipsis  
Такая форма записи впервые появилась в версии Python 3.0   
(и может использоваться гораздо шире своего основного пред-  
назначения – в расширенных операциях извлечения среза).   
Возможно, со временем в подобных ситуациях многоточие по-  
лучит более широкое распространение, чем инструкция pass   
и объект None.  
continue  
Инструкция continue вызывает немедленный переход в начало цикла. Она ино-  
гда позволяет избежать использования вложенных инструкций. В следующем   
примере инструкция continue используется для пропуска нечетных чисел. Этот   
фрагмент выводит четные числа меньше 10 и больше или равные 0. Вспомним,   
что число 0 означает ложь, а оператор % вычисляет остаток от деления, поэтому   
данный цикл выводит числа в обратном порядке, пропуская значения, не крат-  
ные 2 (он выводит 8 6 4 2 0):  
x = 10  
while x:  
 x = x-1 # Или, x -= 1  
 if x % 2 != 0: continue # Нечетное? – пропустить вывод  
 print(x, end=’ ‘)  
Так как инструкция continue выполняет переход в начало цикла, нам не по-  
требовалось вкладывать инструкцию print в инструкцию if – она будет задей-  
ствована, только если инструкция continue не будет выполнена. Если она напо-  
минает вам инструкцию «goto», имеющуюся в других языках, то это справед-  
ливо. В языке Python нет инструкции «goto», но так как инструкция continue   
позволяет выполнять переходы внутри программы, большинство замечаний,   
касающихся удобочитаемости и простоты сопровождения, которые вы могли   
слышать в отношении инструкции «goto», применимы и к инструкции contin-  
ue. Не злоупотребляйте использованием этой инструкции, особенно когда вы   
только начинаете работать с языком Python. Например, последний пример вы-  
глядел бы понятнее, если бы инструкция print была вложена в инструкцию if:  
x = 10  
while x:  
 x = x-1  
 if x % 2 == 0: # Четное? - вывести  
 print(x, end=’ ‘)

break, continue, pass и else   
397  
break  
Инструкция break вызывает немедленный выход из цикла. Так как программ-  
ный код, следующий в цикле за этой инструкцией, не выполняется, если эта   
инструкция запущена, то ее также можно использовать, чтобы избежать вло-  
жения. Например, ниже приводится простой интерактивный цикл (вариант   
более крупного примера, рассматривавшегося в главе 10), где производится   
ввод данных с помощью функции input (raw\_input в Python 2.6) и производится   
выход из цикла, если в ответ на запрос имени будет введена строка «stop»:  
>>> while 1:  
... name = input(‘Enter name:’)  
... if name == ‘stop’: break  
... age = input(‘Enter age: ‘)  
... print(‘Hello’, name, ‘=>’, int(age) \*\* 2)  
...  
Enter name:mel  
Enter age: 40  
Hello mel => 1600  
Enter name:bob  
Enter age: 30  
Hello bob => 900  
Enter name:stop  
Обратите внимание, как в этом примере выполняется преобразование строки   
age в целое число с помощью функции int, перед тем как возвести его во вторую   
степень. Как вы помните, это совершенно необходимо, потому что функция in-  
put возвращает ввод пользователя в виде строки. В главе 35 вы увидите, что   
функция input также возбуждает исключение при получении символа конца   
файла (например, когда пользователь нажимает комбинацию клавиш Ctrl-Z   
или Ctrl-D). Если это может иметь влияние, оберните вызов функции input ин-  
струкцией try.  
else  
При объединении с частью else инструкция break часто позволяет избавиться   
от необходимости сохранять флаг состояния поиска, как это делается в дру-  
гих языках программирования. Например, следующий фрагмент определяет,   
является ли положительное целое число y простым числом, выполняя поиск   
делителей больше 1:  
x = y // 2 # Для значений y > 1  
while x > 1:  
 if y % x == 0: # Остаток  
 print(y, ‘has factor’, x)  
 break # Перешагнуть блок else  
 x -= 1  
else: # Нормальное завершение цикла  
 print(y, ‘is prime’)  
Вместо того чтобы устанавливать флаг, который будет проверен по окончании   
цикла, достаточно вставить инструкцию break в месте, где будет найден де-  
литель. При такой реализации управление будет передано блоку else, только   
если инструкция break не была выполнена, то есть когда с уверенностью можно   
сказать, что число является простым.

398   
Глава 13. Циклы while и for   
Блок else цикла выполняется также в том случае, когда тело цикла ни разу не   
выполнялось, поскольку в этой ситуации инструкция break также не выполня-  
ется. В циклах while это происходит, когда первая же проверка условия в за-  
головке дает ложное значение. Вследствие этого в предыдущем примере будет   
получено сообщение «is prime» (простое число), если изначально x меньше или   
равно 1 (то есть, когда y равно 2).  
Этот пример определяет простые числа, но недостаточно точно.   
Числа, меньшие 2, не считаются простыми в соответствии со   
строгим математическим определением. Если быть более точ-  
ным, этот программный код также будет терпеть неудачу при   
отрицательных значениях и выполняться успешно при исполь-  
зовании вещественных чисел без дробной части. Обратите так-  
же внимание, что в Python 3.0 вместо оператора деления / ис-  
пользуется оператор //, потому что теперь оператор / выполняет   
операцию «истинного деления», как описано в главе 5 (первона-  
чальное деление необходимо, чтобы отсечь остаток!). Если вы за-  
хотите поэкспериментировать с этим фрагментом, загляните   
в упражнения к четвертой части книги, где этот пример обернут   
в функцию.  
Еще о блоке else в цикле  
Так как блок else в цикле является уникальной особенностью языка Python,   
он нередко становится источником недопонимания для тех, кто только начи-  
нает осваивать его. В общих чертах, блок else в циклах обеспечивает явный   
синтаксис представления распространенной ситуации – эта программная   
структура позволяет обработать «другой» способ выхода из цикла, без необхо-  
димости устанавливать и проверять флаги или условия.  
Предположим, например, что вы создаете цикл поиска некоторого значения   
в списке и после выхода из цикла вам необходимо узнать, было ли найдено зна-  
чение. Эту задачу можно решить следующим способом:  
found = False  
while x and not found:  
 if match(x[0]): # Искомое значение является первым?  
 Print(‘Ni’)  
 found = True  
 else:  
 x = x[1:] # Вырезать первое значение и повторить  
if not found:  
 print(‘not found’)  
Здесь мы инициализируем, устанавливаем и проверяем флаг, чтобы опреде-  
лить, увенчался поиск успехом или нет. Это вполне корректный программный   
код для языка Python, и он работает, однако это именно тот случай, когда мож-  
но использовать блок else в цикле. Ниже приводится эквивалентный фрагмент:  
while x: # Выйти, когда x опустеет  
 if match(x[0]):  
 print(‘Ni’)  
 break # Выход, в обход блока else

break, continue, pass и else   
399  
 x = x[1:]  
else:  
 print(‘Not found’) # Этот блок отработает, только если строка x исчерпана  
Эта версия более компактна. Нам удалось избавиться от флага и заменить ин-  
струкцию if за циклом на блок else (по вертикали находится на одной линии со   
словом while). Так как выход из цикла while по инструкции break минует блок   
else, его можно рассматривать как более удобный способ обработки случая   
неудачного поиска.  
Некоторые из вас могли бы заметить, что в предыдущем примере блок else   
можно заменить проверкой строки x после выхода из цикла (например, if not   
x:). Для данного примера это вполне возможно, но часть else обеспечивает   
явный синтаксис реализации этого шаблона программирования (здесь – это   
более очевидный блок обработки ситуации неудачного поиска), и кроме того,   
подобная проверка не всегда возможна. Часть else в циклах становится еще   
более полезной, когда используется в сочетании с инструкцией цикла for – те-  
мой следующего раздела, потому что обход последовательностей выполняется   
неподконтрольно вам.  
Придется держать в уме: имитация циклов while языка C  
В разделе главы 11, где рассматривались инструкции выражений,   
утверждалось, что язык Python не предусматривает возможность вы-  
полнять присваивание там, где ожидается выражение. Это означает, что   
следующий, широко используемый, шаблон программирования языка   
C неприменим в языке Python:  
while ((x = next()) != NULL) {...обработка x...}  
Операции присваивания в языке C возвращают присвоенное значение,   
но в языке Python присваивание – это всего лишь инструкция, а не вы-  
ражение. Благодаря этому ликвидируется обширный класс ошибок,   
свойственных языку C (в языке Python невозможно по ошибке оставить   
знак = там, где подразумевается ==). Но в случае необходимости в ци-  
клах while языка Python подобное поведение можно реализовать как   
минимум тремя способами, без встраивания инструкции присваивания   
в условное выражение. Операцию присваивания можно переместить   
в тело цикла вместе с инструкцией break:  
while True:  
 x = next()  
 if not x: break  
 ...обработка x...  
или вместе с инструкцией if:  
x = True  
while x:  
 x = next()  
 if x:  
 ...обработка x...

400   
Глава 13. Циклы while и for   
или вынести первое присваивание за пределы цикла:  
x = next()  
while x:  
 ...обработка x...  
 x = next()  
Из этих трех вариантов первый, как могут полагать некоторые, – наи-  
менее структурированный, но он же представляется наиболее простым   
и наиболее часто используемым. Простейший цикл for в языке Python   
также может заменить некоторые циклы языка C.  
Циклы for  
Цикл for является универсальным итератором последовательностей в языке   
Python: он может выполнять обход элементов в любых упорядоченных объ-  
ектах последовательностей. Инструкция for способна работать со строками,   
списками, кортежами, с другими встроенными объектами, поддерживающи-  
ми возможность выполнения итераций, и с новыми объектами, которые соз-  
даются с помощью классов, как будет показано позже. Мы уже встречались   
с этой инструкцией, когда рассматривали типы последовательностей, а теперь   
познакомимся с ней поближе.  
Общий формат  
Циклы for в языке Python начинаются со строки заголовка, где указывается   
переменная для присваивания (или – цель), а также объект, обход которого бу-  
дет выполнен. Вслед за заголовком следует блок (обычно с отступами) инструк-  
ций, которые требуется выполнить:  
for <target> in <object>: # Связывает элементы объекта с переменной цикла  
 <statements> # Повторяющееся тело цикла: использует переменную цикла  
else:  
 <statements> # Если не попали на инструкцию ‘break’  
Когда интерпретатор выполняет цикл for, он поочередно, один за другим, при-  
сваивает элементы объекта последовательности переменной цикла и выполня-  
ет тело цикла для каждого из них. Для обращения к текущему элементу после-  
довательности в теле цикла обычно используется переменная цикла, как если   
бы это был курсор, шагающий от элемента к элементу.  
Имя, используемое в качестве переменной цикла (возможно, новой), которое   
указывается в заголовке цикла for, обычно находится в области видимости,   
где располагается сама инструкция for. О ней почти нечего сказать� хотя она   
может быть изменена в теле цикла, тем не менее ей автоматически будет при-  
своен следующий элемент последовательности, когда управление вернется   
в начало цикла. После выхода из цикла эта переменная обычно все еще ссы-  
лается на последний элемент последовательности, если цикл не был завершен   
инструкцией break.  
Инструкция for также поддерживает необязательную часть else, которая ра-  
ботает точно так же, как и в циклах while, – она выполняется, если выход из

Циклы for   
401  
цикла производится не инструкцией break (то есть, если в цикле был выполнен   
обход всех элементов последовательности). Инструкции break и continue, пред-  
ставленные выше, в циклах for работают точно так же, как и в циклах while.   
Полная форма цикла for имеет следующий вид:  
for <target> in <object>: # Присваивает элементы объекта с переменной цикла  
 <statements>  
 if <test>: break # Выход из цикла, минуя блок else  
 if <test>: continue # Переход в начало цикла  
else:  
 <statements> # Если не была вызвана инструкция ‘break’  
Примеры  
Рассмотрим несколько интерактивных циклов for, чтобы вы могли увидеть,   
как они используются на практике.  
Типичные варианты использования  
Как упоминалось ранее, цикл for может выполнять обход элементов в любых   
объектах последовательностей. В нашем первом примере, например, мы пооче-  
редно, слева направо, присвоим переменной x каждый из трех элементов спи-  
ска и выведем каждый из них с помощью инструкции print. Внутри инструк-  
ции print (в теле цикла) имя x ссылается на текущий элемент списка:  
>>> for x in [“spam”, “eggs”, “ham”]:  
... print(x, end=’ ‘)  
...  
spam eggs ham  
В следующих двух примерах вычисляется сумма и произведение всех элемен-  
тов в списке. В этой главе и далее в книге мы познакомимся с инструментами,   
которые применяют такие операции, как + и \*, к элементам списка автомати-  
чески, но обычно для этого используется цикл for:  
>>> sum = 0  
>>> for x in [1, 2, 3, 4]:  
... sum = sum + x  
...  
>>> sum  
10  
>>> prod = 1  
>>> for item in [1, 2, 3, 4]: prod \*= item  
...  
>>> prod  
24  
Другие типы данных  
Цикл for, будучи универсальным инструментом, может применяться к лю-  
бым последовательностям. Например, цикл for может применяться к строкам   
и кортежам:  
>>> S = “lumberjack”  
>>> T = (“and”, “I’m”, “okay”)  
   
>>> for x in S: print(x, end=’ ‘) # Обход строки  
...

402   
Глава 13. Циклы while и for   
l u m b e r j a c k  
   
>>> for x in T: print(x, end=’ ‘) # Обход элементов кортежа  
...  
and I’m okay  
Фактически, как будет показано чуть ниже, циклы for могут применяться   
даже к объектам, которые вообще не являются последовательностями, напри-  
мер к файлам и словарям!  
Присваивание кортежа в цикле for  
Если выполнить обход последовательности кортежей, переменная цикла сама   
фактически будет кортежем. Это лишь еще один случай операции присваива-  
ния кортежа. Не забывайте, что инструкция цикла for присваивает элементы   
объекта последовательности переменной цикла, а операция присваивания вез-  
де выполняется одинаково:  
>>> T = [(1, 2), (3, 4), (5, 6)]  
>>> for (a, b) in T: # Операция присваивания кортежа в действии  
... print(a, b)  
...  
1 2  
3 4  
5 6  
Здесь первый проход цикла действует подобно инструкции (a, b) = (1, 2), второй   
проход – инструкции (a, b) = (3, 4) и так далее. В результате в каждой итерации   
автоматически выполняется операция присваивания кортежа.  
Такая форма присваивания в цикле for часто выполняется с использовани-  
ем функции zip, с которой мы встретимся ниже в этой главе, – когда будем   
рассматривать возможность одновременного обхода сразу нескольких после-  
довательностей. Эта же форма цикла часто используется при работе с базами   
данных, когда ответ на запрос SQL возвращается в виде последовательности   
последовательностей, таких как список в данном примере, – внешний список   
представляет таблицу в базе данных, вложенные кортежи – строки в таблице,   
а инструкция присваивания кортежа извлекает значения полей.  
Кортежи в циклах for можно использовать для обхода ключей и значений сло-  
варей, применяя метод items, что гораздо удобнее, чем выполнять обход ключей   
и затем с помощью операции индексирования извлекать значения:  
>>> D = {‘a’: 1, ‘b’: 2, ‘c’: 3}  
>>> for key in D:  
... print(key, ‘=>’, D[key]) # Используется итератор словаря   
... # и операция индексирования  
a => 1  
c => 3  
b => 2  
   
>>> list(D.items())  
[(‘a’, 1), (‘c’, 3), (‘b’, 2)]  
   
>>> for (key, value) in D.items():  
... print(key, ‘=>’, value) # Обход ключей и значений одновременно  
...  
a => 1

Циклы for   
403  
c => 3  
b => 2  
Важно отметить, что операция присваивания кортежей в инструкции цикла   
for не является каким-то особым случаем – синтаксически после слова for вы-  
полняется присваивание переменной цикла любого вида. С другой стороны,   
мы всегда можем выполнить распаковывание кортежа внутри цикла:  
>>> T  
[(1, 2), (3, 4), (5, 6)]  
>>> for both in T:  
... a, b = both # Эквивалентный вариант с присваиванием вручную  
... print(a, b)  
...  
1 2  
3 4  
5 6  
Использование кортежей в заголовке цикла позволяет сэкономить одну опе-  
рацию в теле цикла при обходе последовательности последовательностей. Как   
уже упоминалось в главе 11, этот способ можно использовать даже для автома-  
тического распаковывания вложенных структур в цикле for:  
>>> ((a, b), c) = ((1, 2), 3) # Вложенные структуры также могут использоваться  
>>> a, b, c  
(1, 2, 3)  
>>> for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: print(a, b, c)  
...  
1 2 3  
4 5 6  
Это также не является каким-то специальным случаем – просто в каждой ите-  
рации инструкция for выполняет операцию присваивания, аналогичную той,   
что продемонстрирована чуть выше цикла. Таким способом может быть распа-  
кована последовательность любых структур, – просто благодаря универсаль-  
ности операции присваивания последовательностей:  
>>> for ((a, b), c) in [([1, 2], 3), [‘XY’, 6]]: print(a, b, c)  
...  
1 2 3  
X Y 6  
Расширенная операция присваивания последовательностей   
в циклах for в версии Python 3.0  
Фактически благодаря тому, что к переменной цикла в инструкции цикла for   
может применяться присваивание любого типа, в Python 3.0 мы можем ис-  
Python 3.0 мы можем ис-  
 3.0 мы можем ис-  
пользовать синтаксис присваивания расширенной операции распаковывания   
последовательностей, извлекая элементы и фрагменты последовательностей.   
В действительности это тоже не какой-то особый случай, а просто новая фор-  
ма присваивания, появившаяся в версии 3.0 (которая рассматривалась в гла-  
ве 11). Так как этот синтаксис может применяться в инструкциях присваива-  
ния, его также можно использовать в циклах for.   
Вернемся к форме присваивания кортежей, рассматривавшейся в предыду-  
щем разделе. В каждой итерации кортеж значений присваивается кортежу   
переменных, точно так же, как в обычной инструкции присваивания:

404   
Глава 13. Циклы while и for   
>>> a, b, c = (1, 2, 3) # Присваивание кортежа  
>>> a, b, c  
(1, 2, 3)  
>>> for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: # Используется в цикле for  
... print(a, b, c)  
...  
1 2 3  
4 5 6  
Благодаря тому, что в Python 3.0 появилась возможность выполнить присва-  
Python 3.0 появилась возможность выполнить присва-  
 3.0 появилась возможность выполнить присва-  
ивание последовательности множеству переменных, где имени со звездочкой   
может быть присвоено сразу множество элементов, мы можем использовать тот   
же прием для извлечения фрагментов вложенных последовательностей в ци-  
кле for:  
>>> a, \*b, c = (1, 2, 3, 4) # Расширенная инструкция   
>>> a, b, c # распаковывания последовательностей  
(1, [2, 3], 4)  
   
>>> for (a, \*b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:  
... print(a, b, c)  
...  
1 [2, 3] 4  
5 [6, 7] 8  
На практике этот подход можно использовать для выборки нескольких полей   
из записей с данными, представленными в виде вложенных последовательно-  
стей. В Python 2.X не поддерживается расширенная операция распаковыва-  
Python 2.X не поддерживается расширенная операция распаковыва-  
 2.X не поддерживается расширенная операция распаковыва-  
X не поддерживается расширенная операция распаковыва-  
 не поддерживается расширенная операция распаковыва-  
ния последовательностей, однако того же эффекта можно добиться с помощью   
операции извлечения среза. Единственное отличие состоит в том, что операция   
извлечения среза возвращает результат, зависящий от типа исходной после-  
довательности, тогда как расширенная операция распаковывания последова-  
тельностей всегда возвращает список:  
>>> for all in [(1, 2, 3, 4), (5, 6, 7, 8)]: # Извлечение среза в 2.6  
... a, b, c = all[0], all[1:3], all[3]  
... print(a, b, c)  
...  
1 (2, 3) 4  
5 (6, 7) 8  
Подробнее об этой форме присваивания рассказывается в главе 11.  
Вложенные циклы for  
Теперь рассмотрим более сложный вариант цикла for. Следующий пример ил-  
люстрирует использование блока else и вложенный цикл for. Имея список объ-  
ектов (items) и список ключей (tests), этот фрагмент пытается отыскать каж-  
дый ключ в списке объектов и сообщает о результатах поиска:  
>>> items = [“aaa”, 111, (4, 5), 2.01] # Множество объектов  
>>> tests = [(4, 5), 3.14] # Ключи, которые требуется отыскать  
>>>  
>>> for key in tests: # Для всех ключей  
... for item in items: # Для всех элементов  
... if item == key: # Проверить совпадение  
... print(key, “was found”)

Циклы for   
405  
... break  
... else:  
... print(key, “not found!”)  
...  
(4, 5) was found  
3.14 not found!  
Поскольку при обнаружении совпадения вложенная инструкция if вызывает   
инструкцию break, можно утверждать, что блок else будет выполняться только   
в случае, когда поиск завершится неудачей. Обратите внимание на вложение   
инструкций. Если запустить этот фрагмент, одновременно будут выполняться   
два цикла: внешний цикл будет выполнять обход списка ключей, а внутрен-  
ний будет выполнять обход списка элементов в поисках каждого ключа. Уро-  
вень вложенности блока else имеет большое значение – он находится на уровне   
строки заголовка внутреннего цикла for, поэтому он соответствует внутренне-  
му циклу (не инструкции if и не внешнему циклу for).  
Примечательно, что этот пример можно упростить, если использовать опера-  
тор in для проверки вхождения ключа. Поскольку оператор in неявно выпол-  
няет обход списка в поисках совпадения, он заменяет собой внутренний цикл:  
>>> for key in tests: # Для всех ключей  
... if key in items: # Позволить интерпретатору отыскать совпадение  
... print(key, “was found”)  
... else:  
... print(key, “not found!”)  
...  
(4, 5) was found  
3.14 not found!  
Вообще, ради компактности кода и скорости вычислений всегда правильнее   
будет переложить на плечи интерпретатора как можно больше работы, как это   
сделано в данном примере.  
Следующий пример с помощью цикла for решает типичную задачу обработ-  
ки данных – выборку одинаковых элементов из двух последовательностей (из   
строк). Это достаточно простая задача поиска пересечения двух множеств. По-  
сле того как цикл for выполнится, переменная res будет ссылаться на список,   
содержащий все одинаковые элементы, обнаруженные в seq1 и seq2:  
>>> seq1 = “spam”  
>>> seq2 = “scam”  
>>>  
>>> res = [] # Изначально список пуст  
>>> for x in seq1: # Выполнить обход первой последовательности  
... if x in seq2: # Общий элемент?  
... res.append(x) # Добавить в конец результата  
...  
>>> res  
[‘s’, ‘a’, ‘m’]  
К сожалению, этот фрагмент работает только с двумя определенными пере-  
менными: seq1 и seq2. Было бы замечательно, если бы этот цикл можно было   
привести к более универсальному виду, тогда его можно было бы использовать   
многократно. Эта простая идея ведет нас к функциям, теме следующей части   
книги.

406   
Глава 13. Циклы while и for   
Придется держать в уме: сканирование файлов  
Циклы удобно использовать там, где надо повторно выполнять некоторые   
действия или многократно обрабатывать данные. Файлы содержат мно-  
жество символов и строк, поэтому они могут рассматриваться как один   
из типичных объектов применения циклов. Чтобы загрузить содержимое   
файла в строку одной инструкцией, достаточно вызвать метод read:  
file = open(‘test.txt’, ‘r’) # Прочитать содержимое файла в строку  
print(file.read())  
Но для загрузки файла по частям обычно используется либо цикл while,   
завершающийся инструкцией break по достижении конца файла, либо   
цикл for. Чтобы выполнить посимвольное чтение, достаточно любого из   
следующих фрагментов:  
file = open(‘test.txt’)  
while True:  
 char = file.read(1) # Читать по одному символу  
 if not char: break  
 print(char)  
   
for char in open(‘test.txt’).read():  
 print(char)  
Тут цикл for выполняет обработку каждого отдельного символа, но за-  
грузка содержимого файла в память производится однократно. Чтение   
строками или блоками циклом while реализуется следующим образом:  
file = open(‘test.txt’)  
while True:  
 line = file.readline() # Читать строку за строкой  
 if not line: break  
 print(line, end=’ ‘) # Прочитанная строка уже содержит символ \n  
   
file = open(‘test.txt’, ‘rb’)  
while True:  
 chunk = file.read(10) # Читать блоками по 10 байтов  
 if not chunk: break  
 print(chunk)  
Двоичные данные обычно читаются блоками определенного размера.   
Однако в случае текстовых данных построчное чтение с помощью цикла   
for выглядит проще и работает быстрее:  
for line in open(‘test.txt’).readlines():  
 print(line, end=’’)  
   
for line in open(‘test.txt’): # Использование итератора: лучший способ   
 print(line, end=’’) # чтения текста  
Метод файлов readlines загружает файл целиком в список строк, тогда   
как при использовании итератора файла в каждой итерации загружа-  
ется только одна строка (итераторы подробно рассматриваются в гла-  
ве 14). Подробности об использованных здесь функциях вы найдете в ру-  
ководстве по стандартной библиотеке языка Python.

Приемы программирования циклов   
407  
Последний пример представляет наиболее предпочтительный способ ра-  
боты с текстовыми файлами. Он не только проще, но и способен работать   
с файлами любого размера, так как не загружает файл целиком в па-  
мять. Версия на базе итератора может оказаться самой быстрой, но пока   
остается неясным вопрос, связанный с производительностью операций   
ввода-вывода в Python3.0.  
В коде, написанном для версии Python 2.X, можно встретить вызов   
функции file вместо open, а также устаревший метод файлов xreadlines,   
который позволяет добиться того же эффекта, что с итератором файлов   
(он действует так же, как метод readlines, но не загружает файл в па-  
мять целиком). Функция file и метод xreadlines были ликвидированы   
в Python 3.0 вследствие их избыточности. Вам также не следует исполь-  
зовать их, если вы пользуетесь Python 2.6, но они все еще могут вам   
встретиться в старых программах. Подробнее об операциях чтения фай-  
лов рассказывается в главе 36, где вы узнаете, что работа с текстовыми   
и двоичными файлами в версии Python 3.0 имеет немного отличающий-  
ся смысл.  
Приемы программирования циклов  
Цикл for относится к категории счетных циклов. Обычно он выглядит проще   
и работает быстрее, чем цикл while, поэтому его нужно рассматривать в самую   
первую очередь, когда возникает необходимость выполнить обход последова-  
тельности. Однако существуют ситуации, когда необходимо выполнять обход   
каким-то особенным способом. Например, как быть, если необходимо выпол-  
нить обход каждого второго или каждого третьего элемента в списке или попут-  
но выполнить изменения в списке? Или если необходимо реализовать парал-  
лельный обход более чем одной последовательности в одном и том же цикле for?  
Такие уникальные ситуации всегда можно запрограммировать с помощью цик-  
ла while и извлечения элементов вручную, но Python предоставляет две встро-  
енные возможности, позволяющие управлять обходом элементов в цикле for:  
 •  
Встроенная функция range возвращает непрерывную последовательность   
увеличивающихся целых чисел, которые можно использовать в качестве   
индексов внутри цикла for.  
 •  
Встроенная функция zip возвращает список кортежей, составленных из   
элементов входных списков с одинаковыми индексами, который может ис-  
пользоваться для одновременного обхода нескольких последовательностей   
в цикле for.  
Обычно циклы for выполняются быстрее, чем аналогичные им счетные циклы   
на базе инструкции while, поэтому везде, где только возможно, лучше пользо-  
ваться такими инструментами, которые позволят использовать цикл for. Рас-  
смотрим каждый из этих встроенных инструментов по очереди.  
Счетные циклы: while и range  
Функция range является по-настоящему универсальным инструментом, ко-  
торый может использоваться в самых разных ситуациях. Чаще всего она ис-

408   
Глава 13. Циклы while и for   
пользуется для генерации индексов в цикле for, но вы можете использовать ее   
везде, где необходимы списки целых чисел. В Python 3.0 функция range возвра-  
щает итератор, который генерирует элементы по требованию, поэтому, чтобы   
отобразить результаты ее работы, мы должны обернуть вызов этой функции   
в вызов функции list (подробнее об итераторах рассказывается в главе 14):  
>>> list(range(5)), list(range(2, 5)), list(range(0, 10, 2))  
([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])  
Функция range с одним аргументом генерирует список целых чисел в диапазо-  
не от нуля до указанного в аргументе значения, не включая его. Если функции   
передать два аргумента, первый будет рассматриваться как нижняя граница   
диапазона. Необязательный третий аргумент определяет шаг – в этом случае   
интерпретатор будет добавлять величину шага при вычислении каждого по-  
следующего значения (по умолчанию шаг равен 1). Существует возможность   
воспроизводить последовательности чисел в диапазоне отрицательных значе-  
ний и в порядке убывания:  
>>> list(range(-5, 5))  
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]  
   
>>> list(range(5, -5, -1))  
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]  
Такое использование функции range само по себе может быть полезным, одна-  
ко чаще всего она используется в циклах for. Прежде всего, она обеспечивает   
простой способ повторить действие определенное число раз. Например, чтобы   
вывести три строки, можно использовать функцию range для создания соответ-  
ствующего количества целых чисел – в версии 3.0 инструкция for автоматиче-  
ски извлекает все значения из итератора range, поэтому нам не потребовалось   
использовать функцию list:  
>>> for i in range(3):  
... print(i, ‘Pythons’)  
...  
0 Pythons  
1 Pythons  
2 Pythons  
Функция range также часто используется для косвенного обхода последова-  
тельностей. Самый простой и самый быстрый способ выполнить обход после-  
довательности заключается в использовании цикла for, когда основную работу   
выполняет интерпретатор:  
>>> X = ‘spam’  
>>> for item in X: print(item, end=’ ‘) # Простейший цикл  
...  
s p a m  
При таком использовании все задачи, касающиеся выполнения итераций, ре-  
шаются внутренними механизмами цикла for. Если вам действительно необ-  
ходимо явно управлять логикой доступа к элементам, можно воспользоваться   
циклом while:  
>>> i = 0  
>>> while i < len(X): # Обход с помощью цикла while   
... print(X[i], end=’ ‘)

Приемы программирования циклов   
409  
... i += 1  
...  
s p a m  
Однако управлять индексами вручную можно и в цикле for, если использо-  
вать функцию range для воспроизведения списка индексов. Это многоэтапный   
процесс, но он вполне пригоден для генерирования смещений, а не элементов   
с этим смещениями:  
>>> X  
‘spam’  
>>> len(X) # Длина строки  
4  
>>> list(range(len(X))) # Все допустимые смещенияв X  
[0, 1, 2, 3]  
>>>  
>>> for i in range(len(X)): print(X[i],end=’ ‘) # Извлечение элементов вручную  
...  
s p a m  
В этом примере выполняется обход списка смещений в строке X, а не фактиче-  
ских элементов строки – нам пришлось внутри цикла обращаться к строке X,   
чтобы извлечь каждый элемент.  
Обход части последовательности: range и срезы  
Последний пример в предыдущем разделе вполне работоспособен, но он выпол-  
няется гораздо медленнее, чем мог бы. Кроме того, нам пришлось выполнить   
больше работы, чем требуется для решения такой задачи. Если вы не предъяв-  
ляете особых требований к индексам, всегда лучше использовать простейшую   
форму цикла for – используйте цикл for вместо while везде, где только возмож-  
но, и используйте функцию range в циклах for, только если это действительно   
необходимо. Следующее простое решение является лучшим:  
>>> for item in X: print(item) # Простейшая итерация  
...  
Однако прием, представленный в предшествующем примере, позволяет нам   
управлять порядком обхода последовательности, например пропускать эле-  
менты:  
>>> S = ‘abcdefghijk’  
>>> list(range(0, len(S), 2))  
[0, 2, 4, 6, 8, 10]  
   
>>> for i in range(0, len(S), 2): print(S[i], end=’ ‘)  
...  
a c e g i k  
Здесь в цикле выбирается каждый второй элемент строки S при обходе списка   
значений, сгенерированных функцией range. Чтобы извлечь каждый третий   
элемент, достаточно изменить третий аргумент функции range, передав в нем   
значение 3, и так далее. Таким образом, функция range позволяет пропускать   
элементы, сохраняя при этом простоту цикла for.  
Однако, на сегодняшний день это, пожалуй, не самый лучший способ. Если   
вам действительно необходимо пропустить элементы последовательности,

410   
Глава 13. Циклы while и for   
можно использовать расширенную форму операции извлечения среза с тре-  
мя пределами, представленную в главе 7, которая обеспечивает более простой   
путь к достижению цели. Чтобы получить каждый второй символ из строки S,   
можно извлечь срез с шагом 2:  
>>> S = ‘abcdefghijk’  
>>> for c in S[::2]: print(c, end=’ ‘)  
...  
a c e g i k  
Изменение списков: range  
Еще одно место, где можно использовать комбинацию функции range и цикла   
for, – это циклы, изменяющие список в процессе его обхода. Например, предпо-  
ложим, что по тем или иным причинам нам необходимо прибавить 1 к каждо-  
му элементу списка. Можно попытаться использовать для этой цели простей-  
шую форму цикла for, но скорее всего это не то, что нам нужно:  
>>> L = [1, 2, 3, 4, 5]  
>>> for x in L:  
... x += 1  
...  
>>> L  
[1, 2, 3, 4, 5]  
>>> x  
6  
Такое решение вообще ничего не дает – здесь изменяется переменная цикла x,   
а не список L. Причину такого поведения трудно заметить. Всякий раз, когда   
цикл выполняет очередную итерацию, переменная x ссылается на очередное   
целое число, которое уже было извлечено из списка. В первой итерации, напри-  
мер, переменная x является целым числом 1. На следующей итерации в пере-  
менную x будет записана ссылка на другой объект – целое число 2, но это никак   
не повлияет на список, откуда было взято число 1.  
Чтобы действительно изменить список во время его обхода, нам необходимо ис-  
пользовать операцию присваивания по индексу и изменить значения во всех   
позициях, по которым осуществляется цикл. Необходимые нам индексы мож-  
но воспроизвести с помощью комбинации функций len/range:  
>>> L = [1, 2, 3, 4, 5]  
   
>>> for i in range(len(L)): # Прибавить 1 к каждому элементу в списке L  
... L[i] += 1 # Или L[i] = L[i] + 1  
...  
>>> L  
[2, 3, 4, 5, 6]  
При такой реализации список изменяется в процессе обхода. Простой цикл for   
x in L: такого результата дать не может, потому что в таком цикле выполняется   
обход фактических элементов, а не позиций в списке. А возможно ли создать   
эквивалентный цикл while? Для этого нам потребуется приложить немного   
больше усилий, и такой цикл наверняка будет работать медленнее:  
>>> i = 0  
>>> while i < len(L):  
... L[i] += 1

Приемы программирования циклов   
411  
... i += 1  
...  
>>> L  
[3, 4, 5, 6, 7]  
В данном случае решение на базе функции range может быть неидеальным. Ге-  
нератор списка в виде  
[x+1 for x in L]   
также даст желаемый результат, но первоначальный список при этом не из-  
менится (мы могли бы присвоить получившийся новый список обратно пере-  
менной L, но это выражение не изменит другие ссылки на первоначальный   
список). Поскольку эта концепция циклов занимает такое важное положение,   
мы еще раз вернемся к ней, когда будем рассматривать генераторы списков   
ниже в этой главе.  
Параллельный обход: zip и map  
Как было показано выше, встроенная функция range позволяет выполнять об-  
ход отдельных частей последовательностей. В том же духе встроенная функция   
zip позволяет использовать цикл for для обхода нескольких последовательно-  
стей параллельно. Функция zip принимает одну или несколько последователь-  
ностей в качестве аргументов и возвращает список кортежей, составленных из   
соответствующих элементов этих последовательностей. Например, предполо-  
жим, что мы выполняем обработку двух списков:  
>>> L1 = [1,2,3,4]  
>>> L2 = [5,6,7,8]  
Для объединения элементов этих списков можно использовать функцию zip,   
которая создаст список кортежей из пар элементов (подобно функции range,   
в версии 3.0 функция zip возвращает итерируемый объект, поэтому, чтобы вы-  
вести все результаты, возвращаемые этой функцией, необходимо обернуть ее   
вызов вызовом функции list, – подробнее об итераторах рассказывается в сле-  
дующей главе):  
>>> zip(L1, L2)  
<zip object at 0x026523C8>  
>>> list(zip(L1,L2)) # Функция list необходима в 3.0, но не в 2.6  
[(1, 5), (2, 6), (3, 7), (4, 8)]  
Такой результат может пригодиться в самых разных ситуациях, но примени-  
тельно к циклу for он обеспечивает возможность выполнения параллельных   
итераций:  
>>> for (x, y) in zip(L1, L2):  
... print(x, y, ‘--’, x+y)  
...  
1 5 -- 6  
2 6 -- 8  
3 7 -- 10  
4 8 -- 12  
Здесь выполняется обход результата обращения к функции zip, то есть пар,   
составленных из элементов двух списков. Обратите внимание, что в этом ци-  
кле используется операция присваивания кортежей для получения элементов

412   
Глава 13. Циклы while и for   
каждого кортежа, полученного от функции zip. На первой итерации она будет   
выглядеть, как если бы была выполнена инструкция (x, y) = (1, 5).  
Благодаря этому мы можем сканировать оба списка L1 и L2 в одном цикле. Тот   
же эффект можно получить с помощью цикла while, в котором доступ к элемен-  
там производится вручную, но такой цикл будет сложнее в реализации и на-  
верняка медленнее, чем прием, основанный на использовании for/zip.  
Функция zip на самом деле более универсальна, чем можно было бы предста-  
вить на основе этого фрагмента. Например, она принимает последовательно-  
сти любого типа (в действительности – любые итерируемые объекты, включая   
и файлы) и позволяет указывать более двух аргументов. При вызове с тремя   
аргументами, как показано в следующем примере, она конструирует список   
кортежей, состоящих из трех элементов, выбирая элементы из каждой после-  
довательности с одним и тем же смещением (с технической точки зрения, из N   
аргументов функция zip создает N-мерный кортеж):  
>>> T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)  
>>> T3  
(7, 8, 9)  
>>> list(zip(T1,T2,T3))  
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]  
Длина списка, возвращаемого функцией zip, равна длине кратчайшей из по-  
следовательностей, если аргументы имеют разную длину. В следующем при-  
мере выполняется объединение двух строк с целью параллельной обработки   
их символов, при этом результат содержит столько кортежей, сколько было   
элементов в кратчайшей последовательности:  
>>> S1 = ‘abc’  
>>> S2 = ‘xyz123’  
>>>  
>>> list(zip(S1, S2))  
[(‘a’, ‘x’), (‘b’, ‘y’), (‘c’, ‘z’)]  
Эквивалентная функция map в Python 2.6  
В Python 2.X имеется родственная встроенная функция map, объединяющая   
элементы последовательностей похожим образом, но она не усекает результат   
по длине кратчайшей последовательности, а дополняет недостающие элементы   
значениями None, если аргументы имеют разную длину:  
>>> S1 = ‘abc’  
>>> S2 = ‘xyz123’  
   
>>> map(None, S1, S2) # Только в 2.X  
[(‘a’, ‘x’), (‘b’, ‘y’), (‘c’, ‘z’), (None, ‘1’), (None, ‘2’), (None,’3’)]  
В этом примере используется вырожденная форма обращения к встроенной   
функции map, которая больше не поддерживается в Python 3.0. Обычно она   
принимает функцию и одну или более последовательностей и собирает резуль-  
таты вызова функции с соответствующими элементами, извлеченными из по-  
следовательностей. Подробнее функция map будет рассматриваться в главах 19   
и 20. Ниже приводится короткий пример, где встроенная функция ord приме-  
няется к каждому символу в строке и собирает результаты в список (подобно   
функции zip, в версии 3.0 map возвращает генератор, и поэтому, чтобы получить

Приемы программирования циклов   
413  
все ее результаты в интерактивном сеансе, обращение к ней следует заключить   
в вызов функции list):  
>>> list(map(ord, ‘spam’))  
[115, 112, 97, 109]  
Тот же результат можно получить с помощью следующего цикла, но реализа-  
ция на основе функции map зачастую выполняется быстрее:  
>>> res = []  
>>> for c in ‘spam’: res.append(ord(c))  
>>> res  
[115, 112, 97, 109]  
Примечание,  касающееся  различий  между  версиями: Вырож-  
денная форма вызова функции map, когда в первом аргументе   
вместо функции передается объект None, больше не поддержива-  
ется в Python 3.0, потому что в этом случае она в значительной   
степени совпадает с функцией zip (и, честно говоря, несколько   
не соответствует основному назначению функции map). В вер-  
сии 3.0 вы можете либо использовать функцию zip, либо напи-  
сать цикл, который сам дополняет недостающие результаты.   
Как это сделать, будет показано в главе 20, после того как мы по-  
знакомимся с дополнительными концепциями итераций.  
Конструирование словаря с помощью функции zip  
В главе 8 я говорил, что функцию zip, используемую здесь, удобно применять   
для создания словарей, когда ключи и значения вычисляются во время выпол-  
нения программы. Теперь, когда мы поближе познакомились с этой функци-  
ей, я объясню, какое отношение она имеет к конструированию словарей. Как   
вы уже знаете, словарь всегда можно создать с помощью литерала словаря или   
присваивая значения ключам:  
>>> D1 = {‘spam’:1, ‘eggs’:3, ‘toast’:5}  
>>> D1  
{‘toast’: 5, ‘eggs’: 3, ‘spam’: 1}  
   
>>> D1 = {}  
>>> D1[‘spam’] = 1  
>>> D1[‘eggs’] = 3  
>>> D1[‘toast’] = 5  
Но как быть, если программа получает ключи и значения для словаря в виде   
списков во время выполнения, уже после того, как сценарий был написан? На-  
пример, предположим, что имеются следующие списки ключей и значений:  
>>> keys = [‘spam’, ‘eggs’, ‘toast’]  
>>> vals = [1, 3, 5]  
Один из способов превратить их в словарь состоит в том, чтобы передать списки   
функции zip и затем выполнить обход полученного результата в цикле for:  
>>> list(zip(keys, vals))  
[(‘spam’, 1), (‘eggs’, 3), (‘toast’, 5)]

414   
Глава 13. Циклы while и for   
>>> D2 = {}  
>>> for (k, v) in zip(keys, vals): D2[k] = v  
...  
>>> D2  
{‘toast’: 5, ‘eggs’: 3, ‘spam’: 1}  
Однако, начиная с версии Python 2.2, можно обойтись без цикла for и просто   
передать результат вызова функции zip встроенному конструктору dict:  
>>> keys = [‘spam’, ‘eggs’, ‘toast’]  
>>> vals = [1, 3, 5]  
>>> D3 = dict(zip(keys, vals))  
>>> D3  
{‘toast’: 5, ‘eggs’: 3, ‘spam’: 1}  
Встроенное имя dict в языке Python в действительности является именем   
типа (больше об именах типов и о создании подтипов вы узнаете в главе 31).   
Этот вызов производит преобразование списка в словарь, но в действительно-  
сти это вызов конструктора объекта. В следующей главе мы рассмотрим род-  
ственное, но более широкое понятие генераторов списков, которые позволяют   
создавать списки с помощью единственного выражения. Мы также вернемся   
еще раз к генераторам словарей, появившихся в версии 3.0, которые являются   
альтернативой вызову dict для пар ключ/значение, объединенных в последо-  
вательность.  
Генерирование индексов и элементов: enumerate  
Ранее мы рассматривали использование функции range для генерации индек-  
сов (смещений) элементов в строке вместо получения самих элементов с этими   
индексами. Однако в некоторых программах необходимо получить и то, и дру-  
гое: и элемент, и его индекс. При традиционном подходе можно было бы ис-  
пользовать простой цикл for, в котором вести счетчик текущего индекса:  
>>> S = ‘spam’  
>>> offset = 0  
>>> for item in S:  
... print(item, ‘appears at offset’, offset)  
... offset += 1  
...  
s appears at offset 0  
p appears at offset 1  
a appears at offset 2  
m appears at offset 3  
Этот способ вполне работоспособен, но в последних версиях языка Python те же   
самые действия можно выполнить с помощью встроенной функции с именем   
enumerate:  
>>> S = ‘spam’  
>>> for (offset, item) in enumerate(S):  
... print(item, ‘appears at offset’, offset)  
...  
s appears at offset 0  
p appears at offset 1  
a appears at offset 2  
m appears at offset 3

В заключение   
415  
Функция enumerate возвращает объект-генератор – разновидность объекта,   
который поддерживает протокол итераций, который мы будем рассматривать   
в следующей главе, и более подробно будем обсуждать в следующей части кни-  
ги. В двух словах: он имеет метод \_\_next\_\_, вызываемый встроенной функцией   
next и возвращающий кортеж (index, value) для каждого элемента списка. Мы   
можем использовать эти кортежи для присваивания в цикле for (точно так же,   
как и в случае с функцией zip):  
>>> E = enumerate(S)  
>>> E  
<enumerate object at 0x02765AA8>  
>>> next(E)  
(0, ‘s’)  
>>> next(E)  
(1, ‘p’)  
>>> next(E)  
(2, ‘a’)  
Обычно мы не видим всю эту механику, потому что во всех случаях (включая   
генераторы списков – тема главы 14) протокол итераций выполняется автома-  
тически:  
>>> [c \* i for (i, c) in enumerate(S)]  
[‘’, ‘p’, ‘aa’, ‘mmm’]  
Чтобы окончательно разобраться с такими понятиями итераций, как функции   
enumerate, zip и генераторы списков, нам необходимо перейти к следующей гла-  
ве, где производится разбор этих понятий с более формальной точки зрения.  
В заключение  
В этой главе мы исследовали инструкции циклов языка Python, а также неко-  
торые концепции, имеющие отношение к циклам. Мы рассмотрели инструк-  
ции циклов while и for во всех подробностях и узнали о связанных с ними   
блоках else. Мы также изучили инструкции break и continue, которые могут   
использоваться только внутри циклов, и дополнительно познакомились с не-  
которыми встроенными инструментами, часто используемыми в цикле for,   
включая функции range, zip, map и enumerate (хотя понимание их роли, как ите-  
раторов в Python 3.0, не может быть полным до прочтения следующей главы).  
В следующей главе мы продолжим исследование механизмов итераций и рас-  
смотрим генераторы списков и протокол итераций в языке Python – концеп-  
Python – концеп-  
 – концеп-  
ций, которые тесно связаны с циклами for. Там же будут объясняться некото-  
рые тонкости применения итерируемых инструментов, с которыми мы встре-  
тились здесь, таких как функции range и zip. Однако, как обычно, прежде чем   
двинуться дальше, попробуйте ответить на контрольные вопросы.  
Закрепление пройденного  
Контрольные вопросы  
1. В чем заключаются основные различия между циклами while и for?  
2. В чем заключаются основные различия между инструкциями break и con-  
tinue?

416   
Глава 13. Циклы while и for   
3. Когда выполняется блок else в циклах?  
4. Как в языке Python можно запрограммировать счетный цикл?  
5. Для чего может использоваться функция range в цикле for?  
Ответы  
1. Инструкция while обеспечивает способ организации универсальных ци-  
клов, а инструкция for предназначена для обхода элементов в последова-  
тельностях (в действительности – в итерируемых объектах). С помощью ин-  
струкции while можно сымитировать счетный цикл for, однако программ-  
ный код получится менее компактным и может выполняться медленнее.  
2. Инструкция break осуществляет немедленный выход из цикла (управление   
передается первой инструкции, следующей за телом цикла while или for),   
а инструкция continue выполняет переход в начало цикла (в позицию непо-  
средственно перед условным выражением в цикле while или перед извлече-  
нием очередного элемента в цикле for).  
3. Блок else в циклах while или for выполняется один раз после выхода из   
цикла при условии, что цикл завершается обычным образом (без исполь-  
зования инструкции break). Инструкция break осуществляет немедленный   
выход из цикла и пропускает блок else (если таковая присутствует).  
4. Счетные циклы могут быть реализованы на базе инструкции while при   
условии, что вычисление индексов будет производиться вручную, или на   
базе инструкции for, которая использует встроенную функцию range для   
генерирования последовательности целых чисел. Ни один из этих способов   
не является предпочтительным в языке Python. Если вам необходимо про-  
сто обойти все элементы в последовательности, везде, где только возможно,   
используйте простой цикл for, без функции range или счетчиков. Такая реа-  
лизация и выглядит проще, и обычно работает быстрее.  
5. Встроенная функция range может использоваться в циклах for для органи-  
зации фиксированного количества итераций, для реализации обхода сме-  
щений вместо самих элементов, для того, чтобы обеспечить пропуск эле-  
ментов и чтобы получить возможность изменять список во время его обхода.   
Однако ни в одном из перечисленных случаев использование функции range   
не является обязательным условием, и для большинства из них имеются   
альтернативные способы – сканирование фактических элементов, использо-  
вание операции извлечения среза с тремя пределами и генераторы списков   
зачастую являются более привлекательными решениями (несмотря на на-  
вязчивое стремление экс-C-программеров подсчитывать все подряд!).

Глава 14.  
   
Итерации и генераторы, часть 1  
В предыдущей главе мы познакомились с двумя инструкциями циклов, име-  
ющимися в языке Python, while и for. Они могут использоваться при решении   
самого широкого круга задач, где необходимо организовать многократное вы-  
полнение повторяющихся операций, однако необходимость перебора элемен-  
тов последовательностей возникает настолько часто, что в язык Python были   
введены дополнительные инструменты, делающие эту операцию более простой   
и эффективной. В данной главе мы начнем исследование этих инструментов.   
В частности, здесь будут рассматриваться родственные концепции протокола   
итераций в языке Python – модели, основанной на вызове методов, используе-  
Python – модели, основанной на вызове методов, используе-  
 – модели, основанной на вызове методов, используе-  
мой в циклах for, а кроме того, будут представлены некоторые подробности, ка-  
сающиеся генераторов списков – близкого родственника цикла for, который по-  
зволяет применять выражение к элементам итерируемой последовательности.  
Оба эти инструмента тесно связаны с циклами for и с функциями, поэтому мы   
будем рассматривать их в два захода: в этой главе, являющейся своего рода   
продолжением предыдущей главы, основные понятия об этих инструментах   
представлены с точки зрения циклического выполнения операций, а в главе 20   
эти инструменты будут рассмотрены с точки зрения вызова функций. В этой   
главе мы также познакомимся с дополнительными инструментами выпол-  
нения итераций в языке Python и коснемся новых итераторов, появившихся   
в Python 3.0.  
Небольшое предварительное примечание: некоторые из концепций, представ-  
ленных в этих главах, на первый взгляд могут показаться избыточными. Одна-  
ко по мере приобретения опыта вы поймете, насколько эти инструменты удоб-  
ны и эффективны. Строго говоря, знание их не является обязательным, однако   
они настолько часто используются в программах, что понимание основ сможет   
помочь вам разобраться в чужих программах, если вдруг возникнет такая не-  
обходимость.  
Итераторы: первое знакомство  
В предыдущей главе упоминалось, что цикл for может работать с последова-  
тельностями любого типа, включая списки, кортежи и строки, например:

418   
Глава 14. Итерации и генераторы, часть 1   
>>> for x in [1, 2, 3, 4]: print(x \*\* 2, end=’ ‘)  
...  
1 4 9 16  
   
>>> for x in (1, 2, 3, 4): print(x \*\* 3, end=’ ‘)  
...  
1 8 27 64  
   
>>> for x in ‘spam’: print(x \* 2, end=’ ‘)  
...  
ss pp aa mm  
Фактически цикл for имеет еще более универсальную природу, чем было по-  
казано, – он способен работать с любыми итерируемыми объектами, поддер-  
живающими возможность выполнения итераций. На самом деле это верно для   
всех средств выполнения итераций, которые выполняют сканирование объек-  
тов слева направо, включая циклы for, генераторы списков, оператор in про-  
верки на вхождение, встроенную функцию map и другие.  
Понятие «итерируемого объекта» является относительно новым в языке Py-  
Py-  
thon, но оно успело прочно внедриться в модель языка. По существу оно яв-  
, но оно успело прочно внедриться в модель языка. По существу оно яв-  
ляется обобщением понятия последовательности – объект считается итери-  
руемым, либо если он физически является последовательностью, либо если он   
является объектом, который воспроизводит по одному результату за раз в кон-  
тексте инструментов выполнения итераций, таких как цикл for. В некотором   
смысле в категорию итерируемых объектов входят как физические последова-  
тельности, так и последовательности виртуальные, которые вычисляются по   
требованию.1  
Протокол итераций: итераторы файлов  
Один из самых простых способов понять, что такое итераторы, – это посмо-  
треть, как они работают со встроенными типами, такими как файлы. В гла-  
ве 9 говорилось, что объекты открытых файлов имеют метод с именем readline,   
который читает по одной строке текста из файла за одно обращение – каждый   
раз, вызывая метод readline, мы перемещаемся к следующей строке. По дости-  
жении конца файла возвращается пустая строка, что может служить сигналом   
для выхода из цикла:  
>>> f = open(‘script1.py’) # Прочитать 4 строки из файла сценария   
>>> f.readline() # Метод readline загружает одну строку  
‘import sys\n’  
>>> f.readline()  
‘print(sys.path)\n’  
>>> f.readline()  
‘x = 2\n’  
>>> f.readline()  
1   
Терминология в этой теме носит немного расплывчатый характер. Термины «ите-  
рируемый» и «итератор» в этой книге используются как взаимозаменяемые и обо-  
значают объекты, которые поддерживают возможность итераций. Иногда термин   
«итерируемый» используется для обозначения объектов, реализующих метод iter,   
а термин «итератор» – для обозначения объектов, возвращаемых функцией iter   
и поддерживающих функцию next(I), однако это соглашение не является универ-  
сальным ни в языке Python, ни в этой книге.

Итераторы: первое знакомство   
419  
‘print(2 \*\* 33)\n’  
>>> f.readline() # Вернет пустую строку по достижении конца файла  
‘’  
Кроме того, файлы имеют также метод \_\_next\_\_, который производит практи-  
чески тот же эффект, – всякий раз, когда его вызывают, он возвращает следую-  
щую строку. Единственное значимое различие состоит в том, что по достиже-  
нии конца файла метод \_\_next\_\_ возбуждает встроенное исключение StopItera-  
tion вместо того, чтобы возвращать пустую строку:  
>>> f = open(‘script1.py’) # Метод \_\_next\_\_ загружает одну строку  
>>> f.\_\_next\_\_() # и возбуждает исключение по достижении конца файла  
‘import sys\n’  
>>> f.\_\_next\_\_()  
‘print(sys.path)\n’  
>>> f.\_\_next\_\_()  
‘x = 2\n’  
>>> f.\_\_next\_\_()  
‘print(2 \*\* 33)\n’  
>>> f.\_\_next\_\_()  
Traceback (most recent call last):  
 ...текст сообщения об ошибке опущен...  
StopIteration  
Такое поведение в точности соответствует тому, что мы в языке Python назы-  
ваем протоколом итераций, – объект реализует метод \_\_next\_\_, который воз-  
вращает следующее значение и возбуждает исключение StopIteration в конце   
серии результатов. Подобные объекты в языке Python считаются итерируемы-  
ми. Любой такой объект доступен для сканирования с помощью цикла for или   
других итерационных инструментов, потому что все инструменты выполнения   
итераций вызывают метод \_\_next\_\_ в каждой итерации и определяют момент   
выхода по исключению StopIteration.  
Следствие всего вышесказанного: лучший способ построчного чтения тексто-  
вого файла, как уже упоминалось в главе 9, состоит не в том, чтобы прочитать   
его целиком, а в том, чтобы позволить циклу for автоматически вызывать ме-  
тод \_\_next\_\_ для перемещения к следующей строке в каждой итерации. Напри-  
мер, следующий фрагмент читает содержимое файла строку за строкой (попут-  
но приводит символы к верхнему регистру и выводит их) без явного обращения   
к методам файла:  
>>> for line in open(‘script1.py’): # Использовать итератор файла  
... print(line.upper(), end=’’) # Вызывает метод \_\_next\_\_,   
... # перехватывает исключение StopIteration  
IMPORT SYS  
PRINT(SYS.PATH)  
X = 2  
PRINT(2 \*\* 33)  
Обратите внимание на аргумент end=’’ в вызове функции print� он подавляет   
вывод символа \n, потому что строки, прочитанные из файла, уже содержат   
этот символ (если этот аргумент опустить, выводимые строки будут переме-  
жаться пустыми строками). Такой способ построчного чтения текстовых фай-  
лов считается лучшим по трем причинам: программный код выглядит проще,   
он выполняется быстрее и более экономно использует память. Более старый   
способ достижения того же эффекта с помощью цикла for состоит в том, что-

420   
Глава 14. Итерации и генераторы, часть 1   
бы вызвать метод readlines для загрузки содержимого файла в память в виде   
списка строк:  
>>> for line in open(‘script1.py’).readlines():  
... print(line.upper(), end=’’)  
...  
IMPORT SYS  
PRINT SYS.PATH  
X = 2  
PRINT 2 \*\* 33  
Способ, основанный на использовании метода readlines, по-прежнему может   
использоваться, но на сегодня он проигрывает из-за неэкономного использова-  
ния памяти. Так как в этом случае файл загружается целиком, данный способ   
не позволит работать с файлами, слишком большими, чтобы поместиться в па-  
мять компьютера. При этом версия, основанная на применении итераторов, не   
подвержена таким проблемам с памятью, так как содержимое файла считы-  
вается по одной строке за раз. Кроме того, способ на базе итераторов должен   
иметь более высокую производительность, хотя это во многом может зависеть   
от версии (в Python 3.0 это преимущество становится менее очевидным из-за   
того, что библиотека ввода-вывода была полностью переписана с целью обеспе-  
чить поддержку Юникода и сделать ее еще более платформонезависимой).  
Как упоминалось во врезке «Придется держать в уме: сканирование файлов»   
в предыдущей главе, существует возможность построчного чтения файлов   
с помощью цикла while:  
>>> f = open(‘script1.py’)  
>>> while True:  
... line = f.readline()  
... if not line: break  
... print(line.upper(), end=’’)  
...  
...вывод тот же самый...  
Однако такой вариант наверняка будет работать медленнее версии, основанной   
на использовании итератора в цикле for, потому что итераторы внутри интер-  
претатора выполняются со скоростью, присущей программам, написанным   
на языке C, тогда как версия на базе цикла while работает со скоростью интер-  
претации байт-кода виртуальной машиной Python. Всякий раз, когда код на   
языке Python подменяется кодом на языке C, скорость его выполнения обычно   
увеличивается. Однако это не всегда так, особенно в Python 3.0� далее в книге   
будут представлены приемы оценки времени выполнения, позволяющие изме-  
рить относительную скорость выполнения вариантов, подобных этим.  
Выполнение итераций вручную: iter и next  
Для поддержки возможности выполнения итераций вручную (и уменьшения   
объема ввода с клавиатуры) в Python 3.0 имеется встроенная функция next,   
которая автоматически вызывает метод \_\_next\_\_ объекта. Допустим, что у нас   
имеется итерируемый объект X, тогда вызов next(X) будет равносилен вызову   
X.\_\_next\_\_(), но выглядит намного проще. Применительно к файлам, напри-  
мер, можно использовать любую из этих двух форм:  
>>> f = open(‘script1.py’)  
>>> f.\_\_next\_\_() # Непосредственный вызов метода

Итераторы: первое знакомство   
421  
‘import sys\n’  
>>> f.\_\_next\_\_()  
‘print(sys.path)\n’  
   
>>> f = open(‘script1.py’)  
>>> next(f) # Встроенная функция next вызовет метод \_\_next\_\_  
‘import sys\n’  
>>> next(f)  
‘print(sys.path)\n’  
С технической точки зрения итерационный протокол имеет еще одну сторону.   
В самом начале цикл for получает итератор из итерируемого объекта, переда-  
вая его встроенной функции iter, которая возвращает объект, имеющий тре-  
буемый метод \_\_next\_\_. Это станет более очевидным, если посмотреть, на то,   
как внутренние механизмы циклов for обрабатывают такие встроенные типы   
последовательностей, как списки:  
>>> L = [1, 2, 3]  
>>> I = iter(L) # Получить объект-итератор  
>>> I.\_\_next\_\_() # Вызвать \_\_next\_\_, чтобы перейти к следующему элементу  
1  
>>> I.\_\_next\_\_()  
2  
>>> I.\_\_next\_\_()  
3  
>>> I.\_\_next\_\_()  
Traceback (most recent call last):  
 ...текст сообщения об ошибке опущен...  
StopIteration  
При работе с файлами этот начальный этап не нужен, потому что объект фай-  
ла имеет собственный итератор. То есть объекты файлов имеют собственный   
метод \_\_next\_\_, и потому для работы с файлами не требуется получать другой   
объект:  
>>> f = open(‘script1.py’)  
>>> iter(f) is f  
True  
>>> f.\_\_next\_\_()  
‘import sys\n’  
Списки и многие другие встроенные объекты не имеют собственных итерато-  
ров, потому что они поддерживают возможность участия сразу в нескольких   
итерациях. Чтобы начать итерации по таким объектам, необходимо предвари-  
тельно вызвать функцию iter:  
>>> L = [1, 2, 3]  
>>> iter(L) is L  
False  
>>> L.\_\_next\_\_()  
AttributeError: ‘list’ object has no attribute ‘\_\_next\_\_’  
   
>>> I = iter(L)  
>>> I.\_\_next\_\_()  
1  
>>> next(I) # То же, что и вызов метода I.\_\_next\_\_()  
2

422   
Глава 14. Итерации и генераторы, часть 1   
Инструменты итераций в языке Python вызывают эти функции автоматиче-  
Python вызывают эти функции автоматиче-  
 вызывают эти функции автоматиче-  
ски, однако мы также можем пользоваться ими при выполнении итераций   
вручную. Следующий фрагмент наглядно демонстрирует эквивалентность ав-  
томатического и ручного способов организации итераций.1  
>>> L = [1, 2, 3]  
>>>  
>>> for X in L: # Автоматический способ выполнения итераций  
... print(X \*\* 2, end=’ ‘) # Получает итератор, вызывает \_\_next\_\_,   
... # обрабатывает исключение  
1 4 9  
   
>>> I = iter(L) # Ручной способ итераций: имитация цикла for  
>>> while True:  
... try: # Инструкция try обрабатывает исключения  
... X = next(I) # Или I.\_\_next\_\_  
... except StopIteration:  
... break  
... print(X \*\* 2, end=’ ‘)  
...  
1 4 9  
Чтобы понять, как действует этот программный код, необходимо знать, что   
инструкция try выполняет операцию и перехватывает исключения, которые   
могут возникнуть в ходе этой операции (мы будем исследовать исключения   
в седьмой части книги). Следует также отметить, что цикл for и другие сред-  
ства выполнения итераций иногда могут действовать иначе, при работе с поль-  
зовательскими классами, многократно производя операцию индексирования   
объекта вместо использования протокола итераций. Обсуждение этой особен-  
ности мы отложим до тех пор, пока в главе 29 не познакомимся с возможно-  
стью перегрузки операторов.  
Примечание,  касающееся  различий  между  версиями: В версии   
Python 2.6 метод итераций называется X.next(), а не X.\_\_next\_\_().   
Для обеспечения переносимости можно использовать встроен-  
ную функцию next(X), доступную также в Python 2.6 (но не в бо-  
лее ранних версиях), которая в версии 2.6 вызывает метод   
1   
Строго говоря, цикл for вызывает внутренний эквивалент метода I.\_\_next\_\_, а не   
функцию next(I), которая используется в примере. Различия между ними весьма не-  
значительны, но, как мы увидим в следующем разделе, в версии 3.0 имеются встроен-  
ные объекты (такие как возвращаемое значение функции os.popen), поддерживающие   
первый вариант и не поддерживающие второй, и при этом они обеспечивают возмож-  
ность обхода с помощью цикла for. Вообще говоря, при реализации итераций вруч-  
ную вы можете использовать любую схему вызова. Если вам интересно, в версии 3.0   
объект, возвращаемый функцией os.popen, был переписан с использованием модуля   
subprocess и класса-обертки. Его метод \_\_getattr\_\_ больше не вызывается в версии 3.0   
при неявном обращении к методу \_\_next\_\_ из встроенной функции next, но он вызы-  
вается при явном обращении по имени – это изменение в версии 3.0, которое будет   
рассматриваться в главах 37 и 38, затронуло даже программный код в стандартной   
библиотеке! Кроме того, в Python 3.0 были ликвидированы родственные функции   
os.popen2/3/4, имевшиеся в Python 2.6, – теперь вместо них следует использовать   
функцию subprocess.Popen с соответствующими аргументами (за дополнительными   
сведениями обращайтесь к руководству по стандартной библиотеке Python 3.0).

Итераторы: первое знакомство   
423  
X.next() вместо X.\_\_next\_\_(). Во всех остальных отношениях   
в версии 2.6 итерации действуют точно так же, как и в вер-  
сии 3.0, – просто используйте метод X.next() или функцию   
next(X) при выполнении итераций вручную. В более ранних вер-  
сиях Python, предшествующих версии 2.6, вместо функции   
next(X) следует вручную вызывать метод X.next().  
Другие итераторы встроенных типов  
Кроме файлов и фактических последовательностей, таких как списки, удоб-  
ные итераторы также имеют и другие типы. Классический способ выполнить   
обход всех ключей словаря, например, состоит в том, чтобы явно запросить   
список ключей:  
>>> D = {‘a’:1, ‘b’:2, ‘c’:3}  
>>> for key in D.keys():  
... print(key, D[key])  
...  
a 1  
c 3  
b 2  
В последних версиях Python словари имеют итератор, который автоматически   
возвращает по одному ключу за раз в контексте итераций:  
>>> I = iter(D)  
>>> next(I)  
‘a’  
>>> next(I)  
‘c’  
>>> next(I)  
‘b’  
>>> next(I)  
Traceback (most recent call last):  
 ...текст сообщения об ошибке опущен...  
StopIteration  
Благодаря этому больше не требуется вызывать метод keys, чтобы выполнить   
обход ключей словаря, – цикл for автоматически будет использовать протокол   
итераций, извлекая ключи по одному в каждой итерации:  
>>> for key in D:  
... print(key, D[key])  
...  
a 1  
c 3  
b 2  
Мы пока не можем углубляться в дальнейшее обсуждение этой темы, но замечу,   
что объекты других типов в языке Python также поддерживают протокол ите-  
Python также поддерживают протокол ите-  
 также поддерживают протокол ите-  
раций и поэтому могут использоваться в циклах for. Например, хранилища объ-  
ектов (объекты файлов с доступом по ключу, используемые для хранения объ-  
ектов Python) и объекты, возвращаемые функцией os.popen (используется для   
чтения вывода команд системной оболочки), также являются итерируемыми:

424   
Глава 14. Итерации и генераторы, часть 1   
>>> import os  
>>> P = os.popen(‘dir’)  
>>> P.\_\_next\_\_()  
‘ Volume in drive C is SQ004828V03\n’  
>>> P.\_\_next\_\_()  
‘ Volume Serial Number is 08BE-3CD4\n’  
>>> next(P)  
TypeError: \_wrap\_close object is not an iterator  
Обратите внимание, что объекты, возвращаемые функцией popen, в Python 2.6   
поддерживают метод P.next(). В версии 3.0 они поддерживают метод P.\_\_next\_\_  
(), но не поддерживают встроенную функцию next(P). Последняя определена   
так, что вызывает метод P.next(), и пока не совсем ясно, будет ли изменено такое   
ее поведение в будущих версиях (как описывалось в сноске выше, это явная   
проблема реализации). Впрочем, данная проблема возникает только при ор-  
ганизации итераций вручную – если используется механизм автоматической   
итерации через эти объекты, с помощью цикла for и других инструментов ите-  
раций (которые описываются в следующих разделах), они возвращают после-  
довательности строк в любых версиях Python.  
Кроме того, поддержка протокола итераций является причиной, по которой   
мы вынуждены были обертывать некоторые результаты в вызов функции list,   
чтобы увидеть все значения сразу. Итерируемые объекты возвращают элемен-  
ты не в виде списка, а по одному элементу за раз:  
>>> R = range(5)  
>>> R # Диапазоны в версии 3.0 – это итерируемые объекты  
range(0, 5)  
>>> I = iter(R) # Используйте протокол итераций для обхода элементов  
>>> next(I)  
0  
>>> next(I)  
1  
>>> list(range(5)) # Или функцию list для получения всех элементов сразу  
[0, 1, 2, 3, 4]  
Теперь, когда вы получили неплохое представление об этом протоколе, вы   
должны суметь объяснить, почему функция enumerate, представленная в преды-  
дущей главе, действует, как показано ниже:  
>>> E = enumerate(‘spam’) # enumerate возвращает итерируемый объект  
>>> E  
<enumerate object at 0x0253F508>  
>>> I = iter(E)  
>>> next(I) # Получить результаты с помощью протокола итераций  
(0, ‘s’)  
>>> next(I)   
(1, ‘p’)  
>>> list(enumerate(‘spam’)) # или с помощью функции list   
[(0, ‘s’), (1, ‘p’), (2, ‘a’), (3, ‘m’)]  
Обычно весь этот механизм скрыт от нас, потому что он автоматически исполь-  
зуется циклом for. Фактически любые инструменты, выполняющие обход объ-  
ектов слева направо, используют протокол итераций таким же точно способом,   
включая инструменты, которые рассматриваются в следующем разделе.

Генераторы списков: первое знакомство   
425  
Генераторы списков: первое знакомство  
Теперь, когда мы узнали, как действует протокол итераций, обратимся к наи-  
более частому случаю его использования. Наряду с циклом for, генераторы   
списков представляют один из самых заметных инструментов, где использует-  
ся протокол итераций.  
В предыдущей главе мы узнали о возможности использовать функцию range   
для изменения списков в ходе выполнения итераций:  
>>> L = [1, 2, 3, 4, 5]  
   
>>> for i in range(len(L)):  
... L[i] += 10  
...  
>>> L  
[11, 12, 13, 14, 15]  
Этот способ работает, но, как я уже упоминал, он может быть далеко не самым   
оптимальным в языке Python. В наши дни генераторы списков переводят мно-  
Python. В наши дни генераторы списков переводят мно-  
. В наши дни генераторы списков переводят мно-  
гое из того, что использовалось раньше, в разряд устаревших приемов. Напри-  
мер, в следующем фрагменте цикл был заменен единственным выражением,   
которое в результате воспроизводит требуемый список:  
>>> L = [x + 10 for x in L]  
>>> L  
[21, 22, 23, 24, 25]  
Конечный результат получается тем же самым, но от нас потребовалось мень-  
ше усилий и, скорее всего, этот вариант работает быстрее. Выражения гене-  
раторов списков нельзя считать равнозначной заменой инструкции цикла for,   
потому что они создают новые объекты списков (что может иметь значение при   
наличии нескольких ссылок на первоначальный список), но это подходящая   
замена для большинства применений, к тому же распространенная и достаточ-  
но удобная, чтобы заслужить внимательного изучения здесь.  
Основы генераторов списков  
Впервые с генераторами списков мы встретились в главе 4. Синтаксис генера-  
торов списков происходит от конструкций, используемых в теории множеств   
для описания операций над каждым элементом множества, но вам совсем не   
обязательно знать теорию множеств, чтобы использовать их. Многие считают,   
что генераторы списков в языке Python напоминают цикл for, записанный за-  
дом наперед.  
Чтобы получить представление о синтаксисе, рассмотрим пример из предыду-  
щего раздела более подробно:  
>>> L = [x + 10 for x in L]  
Генераторы списков записываются в квадратных скобках, потому что это,   
в конечном счете, способ создания нового списка. Генератор списка начинается   
с некоторого составленного нами выражения, которое использует введенную   
нами переменную цикла (x + 10). Далее следует то, что вы без труда опознаете   
как заголовок цикла for, в котором объявляется переменная цикла и итерируе-  
мый объект (for x in L).

426   
Глава 14. Итерации и генераторы, часть 1   
Чтобы найти значение выражения, Python выполняет обход списка L, присваи-  
вая переменной x каждый очередной элемент, и собирает результаты, пропу-  
ская все элементы через выражение слева. Полученный в результате список   
является точным отражением того, что «говорит» генератор списков, – новый   
список, содержащий x+10 для каждого x в L.  
С технической точки зрения всегда можно обойтись без генераторов списков,   
потому что существует возможность создавать список результатов выражения   
вручную, с помощью цикла for:  
>>> res = []  
>>> for x in L:  
... res.append(x + 10)  
...  
>>> res  
[21, 22, 23, 24, 25]  
Фактически это и есть точное представление внутреннего механизма генерато-  
ра списков.  
Но генераторы списков записываются компактнее, и данный способ сборки   
списков получил широкое распространение в языке Python, поэтому они ока-  
Python, поэтому они ока-  
, поэтому они ока-  
зываются очень удобными во многих ситуациях. Более того, генераторы спи-  
сков могут выполняться значительно быстрее (зачастую почти в два раза), чем   
инструкции циклов for, потому что итерации выполняются со скоростью язы-  
ка C, а не со скоростью программного кода на языке Python. Такое преимуще-  
C, а не со скоростью программного кода на языке Python. Такое преимуще-  
, а не со скоростью программного кода на языке Python. Такое преимуще-  
Python. Такое преимуще-  
. Такое преимуще-  
ство в скорости особенно важно для больших объемов данных.  
Использование генераторов списков   
для работы с файлами  
Рассмотрим еще один распространенный случай использования генераторов   
списков, чтобы подробнее исследовать их работу. Вспомним, что у объекта фай-  
ла имеется метод readlines, который загружает файл целиком в список строк:  
>>> f = open(‘script1.py’)  
>>> lines = f.readlines()  
>>> lines  
[‘import sys\n’, ‘print(sys.path)\n’, ‘x = 2\n’, ‘print(2 \*\* 33)\n’]  
Этот фрагмент работает, но все строки в списке оканчиваются символом но-  
вой строки (\n). Символ новой строки является препятствием для многих про-  
грамм – приходится быть осторожным, чтобы избежать появления пустых   
строк при выводе и так далее. Было бы совсем неплохо, если бы мы могли од-  
ним махом избавиться от этих символов новой строки.   
Всякий раз, когда мы заговариваем о выполнении операций над каждым эле-  
ментом последовательности, мы попадаем в сферу действий генераторов спи-  
сков. Например, предположим, что переменная lines находится в том же состо-  
янии, в каком она была оставлена в предыдущем примере. Тогда следующий   
фрагмент обработает каждую строку в списке функцией rstrip, чтобы удалить   
завершающие пробельные символы (также можно было бы использовать вы-  
ражение извлечения среза line[:-1], но только если бы мы были абсолютно уве-  
рены, что все строки завершаются символом новой строки):

Генераторы списков: первое знакомство   
427  
>>> lines = [line.rstrip() for line in lines]  
>>> lines  
[‘import sys’, ‘print(sys.path)’, ‘x = 2’, ‘print(2 \*\* 33)’]  
Этот метод действует, как ожидалось, генераторы списков – это другой итера-  
ционный контекст, но точно так же, как и в простом цикле for, нам не требуется   
даже открывать файл заранее. Если открыть его внутри выражения, генератор   
списков автоматически будет использовать итерационный протокол, с кото-  
рым мы познакомились выше в этой главе. То есть он будет читать из файла по   
одной строке за раз – вызовом метода \_\_next\_\_ файла, пропускать строку через   
функцию rstrip и добавлять результат в список. И снова мы получаем имен-  
но то, что запрашиваем, – результат работы метода rstrip для каждой строки   
в файле:  
>>> lines = [line.rstrip() for line in open(‘script1.py’)]  
>>> lines  
[‘import sys’, ‘print(sys.path)’, ‘x = 2’, ‘print(2 \*\* 33)’]  
Это выражение значительную часть работы выполняет неявно – интерпрета-  
тор сканирует файл и автоматически собирает список результатов выполнения   
операции. Кроме того, это наиболее эффективный способ, потому что большая   
часть действий выполняется внутри интерпретатора Python, который работает   
наверняка быстрее, чем эквивалентная инструкция for. Напомню еще раз, что   
при работе с большими файлами выигрыш в скорости от применения генерато-  
ров списков может оказаться весьма существенным.  
Генераторы списков не только обладают высокой эффективностью, они еще   
весьма выразительны. В этом примере мы могли бы выполнить над строками   
в файле любые строковые операции. Ниже приводится генератор списков, эк-  
вивалентный примеру, рассматривавшемуся выше, в котором использовался   
итератор файла и все символы преобразовывались в верхний регистр, а также   
несколько других (составление цепочки из вызовов методов во втором примере   
стало возможным благодаря тому, что строковые методы возвращают новую   
строку, к которой и применяется другой строковый метод):  
>>> [line.upper() for line in open(‘script1.py’)]  
[‘IMPORT SYS\n’, ‘PRINT(SYS.PATH)\n’, ‘X = 2\n’, ‘PRINT(2 \*\* 33)\n’]  
   
>>> [line.rstrip().upper() for line in open(‘script1.py’)]  
[‘IMPORT SYS’, ‘PRINT(SYS.PATH)’, ‘X = 2’, ‘PRINT(2 \*\* 33)’]  
   
>>> [line.split() for line in open(‘script1.py’)]  
[[‘import’,’sys’], [‘print(sys.path)’], [‘x’,’=’,’2’], [‘print(2’,’\*\*’,’33)’]]  
   
>>> [line.replace(‘ ‘, ‘!’) for line in open(‘script1.py’)]  
[‘import!sys\n’, ‘print(sys.path)\n’, ‘x!=!2\n’, ‘print(2!\*\*!33)\n’]  
   
>>> [(‘sys’ in line, line[0]) for line in open(‘script1.py’)]  
[(True, ‘i’), (True, ‘p’), (False, ‘x’), (False, ‘p’)]  
Расширенный синтаксис генераторов списков  
В действительности генераторы списков могут иметь еще более сложный вид.   
Например, в цикл for, вложенный в выражение, можно добавить оператор if,   
чтобы отобрать результаты, для которых условное выражение дает истинное   
значение.

428   
Глава 14. Итерации и генераторы, часть 1   
Например, предположим, что нам требуется повторить предыдущий пример,   
но при этом необходимо отобрать только строки, начинающиеся с символа p   
(возможно, первый символ в каждой строке – код действия некоторого вида).   
Достичь поставленной цели можно, если добавить фильтрующий оператор if:  
>>> lines = [line.rstrip() for line in open(‘script1.py’) if line[0] == ‘p’]  
>>> lines  
[‘print(sys.path)’, ‘print(2 \*\* 33)’]  
В этом примере оператор if проверяет, является ли первый символ в строке   
символом p. Если это не так, строка не включается в список результатов. Это   
достаточно длинное выражение, но его легко понять, если преобразовать в эк-  
вивалентный простой цикл for (вообще любой генератор списков можно пере-  
вести в эквивалентную реализацию на базе инструкции for, добавляя отступы   
к каждой последующей части):  
>>> res = []  
>>> for line in open(‘script1.py’):  
... if line[0] == ‘p’:  
... res.append(line.rstrip())  
...  
>>> res  
[‘print(sys.path)’, ‘print(2 \*\* 33)’]  
Эта инструкция for выполняет эквивалентные действия, но занимает четыре   
строки вместо одной и работает существенно медленнее.  
В случае необходимости генераторы списков могут иметь еще более сложный   
вид. Например, они могут содержать вложенные циклы, оформленные в виде   
серии операторов for. На самом деле полный синтаксис допускает указывать   
любое число операторов for, каждый из которых может иметь ассоциирован-  
ный с ним оператор if (подробнее о синтаксисе генераторов выражений расска-  
зывается в главе 20).  
Например, следующий фрагмент создает список результатов операции конка-  
тенации x+y для всех x в одной строке и для всех y – в другой. В результате по-  
лучаются сочетания символов в двух строках:  
>>> [x + y for x in ‘abc’ for y in ‘lmn’]  
[‘al’, ‘am’, ‘an’, ‘bl’, ‘bm’, ‘bn’, ‘cl’, ‘cm’, ‘cn’]  
Чтобы проще было понять это выражение, его также можно преобразовать   
в форму инструкции, добавляя отступы к каждой последующей части. Сле-  
дующий фрагмент представляет собой эквивалентную, но более медленную   
реализацию:  
>>> res = []  
>>> for x in ‘abc’:  
... for y in ‘lmn’:  
... res.append(x + y)  
...  
>>> res  
[‘al’, ‘am’, ‘an’, ‘bl’, ‘bm’, ‘bn’, ‘cl’, ‘cm’, ‘cn’]  
Даже с повышением уровня сложности выражения генераторов списков могут   
иметь очень компактный вид. Вообще они предназначены для реализации про-  
стых итераций – для реализации сложных действий более простая инструк-  
ция for наверняка будет проще и для понимания, и для изменения в будущем.

Генераторы списков: первое знакомство   
429  
Обычно если что-то в программировании для вас оказывается слишком слож-  
ным, возможно, это не самое лучшее решение.  
Мы еще вернемся к итераторам и генераторам списков в главе 20, где будем рас-  
сматривать их в контексте функций, где вы увидите, что они связаны с функ-  
циями не менее тесно, чем с инструкциями циклов.  
Другие контексты итераций  
Далее в этой книге мы увидим, что в своих классах также можно реализовать   
поддержку протокола итераций. Поэтому иногда бывает важно знать, какие   
встроенные инструменты могут использоваться для этого, – любой инструмент,   
использующий протокол итераций, автоматически сможет работать с любыми   
встроенными и пользовательскими классами, реализующими его поддержку.  
До настоящего момента я демонстрировал итераторы в контексте инструкции   
цикла for, потому что в этой части книги основное внимание уделяется ин-  
струкциям. Однако имейте в виду, что каждый инструмент, который выполня-  
ет обход объектов слева направо, использует итерационный протокол. В число   
этих инструментов входят и циклы for, как уже было показано выше:  
>>> for line in open(‘script1.py’): # Использовать итератор файла  
... print(line.upper(), end=’’)  
...  
IMPORT SYS  
PRINT(SYS.PATH)  
X = 2  
PRINT(2 \*\* 33)  
Генераторы списков, оператор in, встроенная функция map и другие встроенные   
средства, такие как функции sorted и zip, также основаны на применении ите-  
рационного протокола. Когда все эти инструменты применяются к файлу, они   
автоматически используют итератор объекта файла для построчного чтения:  
>>> uppers = [line.upper() for line in open(‘script1.py’)]  
>>> uppers  
[‘IMPORT SYS\n’, ‘PRINT(SYS.PATH)\n’, ‘X = 2\n’, ‘PRINT(2 \*\* 33)\n’]  
   
>>> map(str.upper, open(‘script1.py’)) # в 3.0 функция map возвращает итератор  
<map object at 0x02660710>  
   
>>> list( map(str.upper, open(‘script1.py’)) )  
[‘IMPORT SYS\n’, ‘PRINT(SYS.PATH)\n’, ‘X = 2\n’, ‘PRINT(2 \*\* 33)\n’]  
   
>>> ‘y = 2\n’ in open(‘script1.py’)  
False  
>>> ‘x = 2\n’ in open(‘script1.py’)  
True  
Используемая здесь функция map, с которой мы познакомились в предыдущей   
главе, представляет собой инструмент, вызывающий заданную функцию для   
каждого элемента итерируемого объекта. Функция map напоминает генераторы   
списков, хотя и с ограничениями, потому что ей можно передать только функ-  
цию, а не произвольное выражение. Кроме того, в Python 3.0 она возвращает   
итерируемый объект, поэтому нам пришлось обернуть обращение к этой функ-  
ции в вызов функции list, чтобы получить сразу все значения, – подробнее об   
этом изменении рассказывается ниже в этой главе. Так как функция map, как

430   
Глава 14. Итерации и генераторы, часть 1   
и генераторы списков, связана с циклами for и с функциями, мы снова вернем-  
ся к их исследованию в главах 19 и 20.  
В языке Python имеются различные встроенные функции, позволяющие об-  
Python имеются различные встроенные функции, позволяющие об-  
 имеются различные встроенные функции, позволяющие об-  
рабатывать итерируемые объекты: функция sorted сортирует элементы итери-  
руемого объекта, функция zip объединяет элементы итерируемых объектов,   
функция enumerate создает пары из элементов итерируемых объектов и их по-  
зиций, функция filter отбирает элементы, для которых указанная функция   
возвращает истинное значение, а функция reduce выполняет указанную опе-  
рацию, объединяя все элементы в итерируемом объекте. Все эти функции при-  
нимают итерируемые объекты, при этом в Python 3.0 функции zip, enumerate   
и filter еще и возвращают итерируемые объекты, подобно функции map. Сле-  
дующий пример демонстрирует эти функции в действии, где они применяются   
к итератору файла и автоматически сканируют его строка за строкой:  
>>> sorted(open(‘script1.py’))  
[‘import sys\n’, ‘print(2 \*\* 33)\n’, ‘print(sys.path)\n’, ‘x = 2\n’]  
   
>>> list(zip(open(‘script1.py’), open(‘script1.py’)))  
[(‘import sys\n’, ‘import sys\n’), (‘print(sys.path)\n’, ‘print(sys.path)\n’),  
(‘x = 2\n’, ‘x = 2\n’), (‘print(2 \*\* 33)\n’, ‘print(2 \*\* 33)\n’)]  
   
>>> list(enumerate(open(‘script1.py’)))  
[(0, ‘import sys\n’), (1, ‘print(sys.path)\n’), (2, ‘x = 2\n’),  
(3, ‘print(2 \*\* 33)\n’)]  
   
>>> list(filter(bool, open(‘script1.py’)))  
[‘import sys\n’, ‘print(sys.path)\n’, ‘x = 2\n’, ‘print(2 \*\* 33)\n’]  
   
>>> import functools, operator  
>>> functools.reduce(operator.add, open(‘script1.py’))  
‘import sys\nprint(sys.path)\nx = 2\nprint(2 \*\* 33)\n’  
Все они являются инструментами итераций, но каждая из них играет свою,   
уникальную роль. С функциями zip и enumerate мы встречались в предыдущей   
главе. С функциями filter и reduce мы познакомимся в главе 19, когда будем   
рассматривать тему функционального программирования, поэтому пока мы не   
будем погружаться в их особенности.  
С функцией sorted мы встретились в первый раз в главе 4 и использовали ее   
при работе со словарями в главе 8. sorted – это встроенная функция, использу-  
ющая протокол итераций, – она похожа на метод sort списков, но возвращает   
новый отсортированный список и принимает любые итерируемые объекты. Об-  
ратите внимание, что в отличие от функции map и других, в версии Python 3.0   
функция sorted возвращает фактический список, а не итерируемый объект.  
Другие встроенные функции также поддерживают протокол итераций (но,   
если честно, для них сложнее придумать достаточно интересный пример об-  
работки файлов). Например, функция sum вычисляет сумму всех чисел в любом   
итерируемом объекте. Встроенные функции any и all возвращают True, если   
любой (any) или все (all) элементы итерируемого объекта являются истинны-  
ми значениями соответственно. Функции max и min возвращают наибольший   
и наименьший элемент итерируемого объекта соответственно. Как и функция   
reduce, все эти инструменты принимают любые итерируемые объекты и ис-  
пользуют протокол итераций для их обхода, но возвращают единственное зна-  
чение, как показано в следующем примере:

Генераторы списков: первое знакомство   
431  
>>> sum([3, 2, 4, 1, 5, 0]) # sum работает только с числами  
15  
>>> any([‘spam’, ‘’, ‘ni’])  
True  
>>> all([‘spam’, ‘’, ‘ni’])  
False  
>>> max([3, 2, 5, 1, 4])  
5  
>>> min([3, 2, 5, 1, 4])  
1  
Строго говоря, функции max и min могут применяться и к файлам – они автома-  
тически используют протокол итераций для обхода содержимого файла и вы-  
бора строк с наибольшим и наименьшим строковым значением соответственно   
(однако я оставлю вам самим определить правильные случаи использования):  
>>> max(open(‘script1.py’)) # Поиск строк с максимальным и минимальным   
‘x = 2\n’ # строковым значением  
>>> min(open(‘script1.py’))  
‘import sys\n’  
Интересно, что область влияния итерационного протокола в языке Python в на-  
стоящее время гораздо шире, чем было продемонстрировано в примерах, – лю-  
бые встроенные инструменты в языке Python, которые выполняют обход объ-  
ектов слева направо, по определению используют итерационный протокол при   
работе с объектами. Сюда относятся даже такие замысловатые инструменты,   
как встроенные функции list и tuple (которые создают новые объекты из ите-  
рируемых объектов), строковый метод join (который вставляет подстроку меж-  
ду строками, содержащимися в итерируемом объекте) и даже операция при-  
сваивания последовательностей. Благодаря этому все они могут применяться   
к открытому файлу и автоматически выполнять чтение по одной строке за раз:  
>>> list(open(‘script1.py’))  
[‘import sys\n’, ‘print(sys.path)\n’, ‘x = 2\n’, ‘print(2 \*\* 33)\n’]  
   
>>> tuple(open(‘script1.py’))  
(‘import sys\n’, ‘print(sys.path)\n’, ‘x = 2\n’, ‘print(2 \*\* 33)\n’)  
   
>>> ‘&&’.join(open(‘script1.py’))  
‘import sys\n&&print(sys.path)\n&&x = 2\n&&print(2 \*\* 33)\n’  
   
>>> a, b, c, d = open(‘script1.py’)  
>>> a, d  
(‘import sys\n’, ‘print(2 \*\* 33)\n’)  
   
>>> a, \*b = open(‘script1.py’) # Расширенная форма в 3.0  
>>> a, b  
(‘import sys\n’, [‘print(sys.path)\n’, ‘x = 2\n’, ‘print(2 \*\* 33)\n’])  
Ранее мы видели, что встроенная функция dict может принимать итерируе-  
мые объекты, возвращаемые функцией zip. То же самое относится к встроен-  
ной функции set и к новым генераторам множеств и словарей, появившимся   
в Python 3.0, с которыми мы встречались в главах 4, 5 и 8:  
>>> set(open(‘script1.py’))  
{‘print(sys.path)\n’, ‘x = 2\n’, ‘print(2 \*\* 33)\n’, ‘import sys\n’}  
   
>>> {line for line in open(‘script1.py’)}  
{‘print(sys.path)\n’, ‘x = 2\n’, ‘print(2 \*\* 33)\n’, ‘import sys\n’}

432   
Глава 14. Итерации и генераторы, часть 1   
>>> {ix: line for ix, line in enumerate(open(‘script1.py’))}  
{0:’import sys\n’, 1:’print(sys.path)\n’, 2:’x = 2\n’, 3:’print(2 \*\* 33)\n’}  
Фактически оба типа генераторов – множеств и словарей – поддерживают   
расширенный синтаксис генераторов списков, с которым мы познакомились   
в этой главе, включая проверку условия if:  
>>> {line for line in open(‘script1.py’) if line[0] == ‘p’}  
{‘print(sys.path)\n’, ‘print(2 \*\* 33)\n’}  
   
>>> {ix: line for (ix, line) in enumerate(open(‘script1.py’)) if line[0]==’p’}  
{1: ‘print(sys.path)\n’, 3: ‘print(2 \*\* 33)\n’}  
Подобно генераторам списков, обе эти конструкции просматривают содержи-  
мое файла строку за строкой и отбирают строки, начинающиеся с символа «p».   
В результате они создают множество и словарь соответственно, выполняя всю   
необходимую нам работу.  
В качестве предварительного знакомства рассмотрим еще один, последний   
итерационный контекст, заслуживающий внимания: в главе 18 мы будем рас-  
сматривать специальную форму аргументов \*arg, которая используется в вызо-  
вах функций для распаковывания значений коллекций в отдельные аргумен-  
ты. Как вы уже наверняка поняли, в виде этой синтаксической конструкции   
можно передать любой итерируемый объект, включая и файлы (более подробно   
синтаксис вызова функций рассматривается в главе 18):  
>>> def f(a, b, c, d): print(a, b, c, d, sep=’&’)  
...  
>>> f(1, 2, 3, 4)  
1&2&3&4  
>>> f(\*[1, 2, 3, 4]) # Распаковывание списка в аргументы  
1&2&3&4  
   
>>> f(\*open(‘script1.py’)) # Можно даже выполнить обход строк в файле!  
import sys  
&print(sys.path)  
&x = 2  
&print(2 \*\* 33)  
Фактически в виде синтаксической конструкции распаковывания аргументов   
в вызовах функций допускается передавать любые итерируемые объекты. Бла-  
годаря этому существует возможность использовать встроенную функцию zip   
для распаковывания упакованных кортежей, передавая ей результаты другого   
вызова функции zip (внимание: возможно, вам не следует читать следующий   
пример, если в ближайшее время вам предстоит управлять тяжелой техни-  
кой!):  
>>> X = (1, 2)  
>>> Y = (3, 4)  
>>>  
>>> list(zip(X, Y)) # Упаковать кортежи: возвратит итерируемый объект  
[(1, 3), (2, 4)]  
>>>  
>>> A, B = zip(\*zip(X, Y)) # Распаковать упакованные кортежи!  
>>> A  
(1, 2)  
>>> B  
(3, 4)

Новые итерируемые объекты в Python 3.0   
433  
Кроме того, в языке Python существуют и другие инструменты, такие как   
встроенная функция range и объекты представлений словарей, возвращающие   
итерируемые объекты вместо списков. Чтобы увидеть, как они используют   
протокол итераций в Python 3.0, нам следует перейти к следующему разделу.  
Новые итерируемые объекты в Python 3.0  
Одно из фундаментальных отличий Python 3.0 от 2.X заключается в том, что   
в версии 3.0 делается сильный акцент на использование итераторов. Вдобавок   
к тому, что итераторы были ассоциированы со многими встроенными типами   
данных, такими как файлы и словари, в Python 3.0 методы keys, values и items   
словарей также возвращают итерируемые объекты, как и встроенные функ-  
ции range, map, zip и filter. Как было показано в предыдущем разделе, три по-  
следние функции не только возвращают итераторы, но и выполняют обработку   
данных в них. Все эти инструменты в Python 3.0 не создают списки с результа-  
Python 3.0 не создают списки с результа-  
 3.0 не создают списки с результа-  
тами, как в версии 2.6, а возвращают результаты по требованию.  
Итераторы позволяют экономнее расходовать память, однако в некоторых слу-  
чаях они могут оказать существенное влияние на стиль программирования.   
Например, ранее в этой книге вам уже приходилось видеть, как мы оберты-  
вали вызовы различных функций и методов в вызов функции list(...), чтобы   
сразу получить все результаты, воспроизводимые итерируемым объектом:  
>>> zip(‘abc’, ‘xyz’) # Итерируемый объект в Python 3.0 (список в 2.6)  
<zip object at 0x02E66710>  
   
>>> list(zip(‘abc’, ‘xyz’)) # Принудительное создание списка   
[(‘a’, ‘x’), (‘b’, ‘y’), (‘c’, ‘z’)] # результатов для отображения в 3.0  
Этого не требуется в версии 2.6, потому что функции, такие как zip, возвра-  
щают список результатов. Но в версии 3.0 они возвращают итерируемые объ-  
екты, которые воспроизводят результаты по требованию. Это означает необхо-  
димость ввода дополнительного программного кода для отображения резуль-  
татов в интерактивной оболочке (и, возможно, в некоторых других случаях),   
однако в крупных программах подобные отложенные вычисления позволяют   
экономить память и ликвидировать паузы, необходимые на вычисление длин-  
ных списков результатов. Давайте теперь рассмотрим некоторые новые, поя-  
вившиеся в версии 3.0, итерируемые объекты в действии.  
Итератор range  
В предыдущей главе мы уже познакомились с основами встроенной функции   
range. В версии 3.0 она возвращает итератор, который не создает сразу весь спи-  
сок целых чисел в заданном диапазоне, а генерирует их по требованию. Она   
действует точно так же, как прежняя функция xrange в версии 2.X (смотрите   
примечание, касающееся различий между версиями, ниже), и в случае, когда   
необходимо получить сразу весь список с результатами (например, для отобра-  
жения), обращение к ней следует обернуть в вызов функции list(range(...)):  
C:\\misc> c:\python30\python  
>>> R = range(10) # range возвращает итератор, а не список  
>>> R  
range(0, 10)

434   
Глава 14. Итерации и генераторы, часть 1   
>>> I = iter(R) # Вернет итератор для диапазона  
>>> next(I) # Переход к следующему результату  
0 # То же происходит в циклах for, генераторах списков и пр.   
>>> next(I)  
1  
>>> next(I)  
2  
>>> list(range(10)) # При необходимости можно принудительно   
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] # сгенерировать список  
В отличие от списков, возвращаемых вызовом этой функции в версии 2.X, объ-  
X, объ-  
, объ-  
екты, возвращаемые функцией range в версии 3.0, поддерживают только воз-  
можность итераций, операцию обращения к элементам по индексам и функ-  
цию len. Они не поддерживают никакие другие операции над последователь-  
ностями (если вам потребуется выполнить другие операции над списками, ис-  
пользуйте конструкцию list(...)):  
>>> len(R) # Диапазоны поддерживают функцию len и операцию индексирования  
10 # Другие операции над последовательностями не поддерживаются  
>>> R[0]  
0  
>>> R[-1]  
9  
   
>>> next(I) # Выборка данных будет продолжена с того места,   
3 # где она была прекращена  
>>> I.\_\_next\_\_() # Метод .next() был переименован в .\_\_next\_\_(),   
4 # но лучше использовать новую функцию next()  
Примечание,  касающееся  различий  между  версиями: В версии   
Python 2.X имеется встроенная функция с именем xrange, кото-  
рая действует так же, как и функция range, но возвращает не   
весь список с результатами, а воспроизводит элементы по требо-  
ванию. Поскольку по своей функциональности она в точности   
соответствует новой функции range, возвращающей итератор,   
функция xrange была ликвидирована в Python 3.0. В програм-  
мах, написанных для работы под управлением интерпретатора   
версии 2.X, все еще можно встретить функцию range, особенно   
там, где требуется получить сразу весь список результатов, но   
вы должны понимать, что она использует память неэффективно.   
Как отмечалось во врезке в предыдущей главе, в Python 3.0 по   
похожим причинам в пользу итераторов файлов, был ликвиди-  
рован метод file.xreadlines(), использовавшийся в версии 2.X   
для уменьшения потребления памяти.  
Итераторы map, zip и filter  
Подобно функции range, встроенные функции map, zip и filter в версии 3.0 так-  
же возвращают итераторы вместо того, чтобы воспроизводить сразу весь спи-  
сок с результатами. В версии 3.0 все три функции не только принимают ите-  
рируемые объекты в виде аргументов, но и возвращают итерируемые объекты   
в виде результатов. Однако, в отличие от функции range, их результаты сами   
и являются этими итераторами, – после однократного получения отдельного

Новые итерируемые объекты в Python 3.0   
435  
результата этот результат исчезает. Другими словами, вы не сможете на осно-  
ве результатов организовать проход по этим результатам нескольких итера-  
торов, который обеспечил бы возможность извлечения результатов в разных   
позициях.  
Ниже приводится пример использования встроенной функции map, с которой   
мы встречались в предыдущей главе. Как и при использовании других итера-  
торов, имеется возможность с помощью функции list(...) получить сразу весь   
список с результатами, но поведение функции map по умолчанию ориентиро-  
вано на экономию существенных объемов памяти при работе с большими на-  
борами результатов:  
>>> M = map(abs, (-1, 0, 1)) # map возвращает итератор, а не список  
>>> M  
<map object at 0x0276B890>  
>>> next(M) # Непосредственное использование итератора:   
1 # результаты исчерпываются безвозвратно.   
>>> next(M) # Они не поддерживают функцию len()   
0 # и операцию индексирования  
>>> next(M)  
1  
>>> next(M)  
StopIteration  
   
>>> for x in M: print(x) # Теперь итератор map пуст:   
... # возможен только один проход  
   
>>> M = map(abs, (-1, 0, 1)) # Чтобы выполнить второй проход, необходимо   
 # снова создать итератор  
>>> for x in M: print(x) # В контексте итераций функция next()  
... # вызывается автоматически  
1  
0  
1  
>>> list(map(abs, (-1, 0, 1))) # При необходимости можно получить сразу   
[1, 0, 1] # весь список с результатами  
Встроенная функция zip, представленная в предыдущей главе, возвращает   
итератор, который действует точно так же:  
>>> Z = zip((1, 2, 3), (10, 20, 30)) # zip также возвращает итератор, который   
>>> Z # позволяет выполнить только один проход  
<zip object at 0x02770EE0>  
   
>>> list(Z)  
[(1, 10), (2, 20), (3, 30)]  
   
>>> for pair in Z: print(pair) # Результаты исчерпываются после   
... # первого прохода  
   
>>> Z = zip((1, 2, 3), (10, 20, 30))  
>>> for pair in Z: print(pair) # Итератор можно использовать   
... # вручную или автоматически  
(1, 10)  
(2, 20)  
(3, 30)  
   
>>> Z = zip((1, 2, 3), (10, 20, 30))

436   
Глава 14. Итерации и генераторы, часть 1   
>>> next(Z)  
(1, 10)  
>>> next(Z)  
(2, 20)  
Встроенная функция filter, которую мы детально будем изучать в следующей   
части книги, действует аналогичным образом. Она возвращает только те эле-  
менты итерируемых объектов, для которых указанная функция возвращает   
значение True (как мы уже знаем, в языке Python значение True в логическом   
контексте получают любые непустые объекты):  
>>> filter(bool, [‘spam’, ‘’, ‘ni’])  
<filter object at 0x0269C6D0>  
   
>>> list(filter(bool, [‘spam’, ‘’, ‘ni’]))  
[‘spam’, ‘ni’]  
Подобно большинству других инструментов, обсуждавшихся в этом разделе,   
в версии 3.0 функция filter не только принимает итерируемые объекты для об-  
работки, но и возвращает итерируемый объект, воспроизводящий результаты   
по требованию.  
Поддержка множественных и единственных   
итераторов  
Многим интересно будет увидеть, в чем заключаются различия между объек-  
том диапазона, возвращаемым функцией range, и объектами, возвращаемыми   
другими встроенными функциями, которые описываются в этом разделе, – ре-  
зультат функции range поддерживает функцию len и операцию доступа к эле-  
ментам по индексу, но сам не является итератором (итератор можно получить   
вызовом функции iter и использовать для выполнения итераций вручную)�   
кроме того, он поддерживает возможность применения к результату несколь-  
ких итераторов, которые отслеживают свое положение в последовательности   
целых чисел независимо друг от друга:  
>>> R = range(3) # Объект диапазона позволяет получить множество итераторов  
>>> next(R)  
TypeError: range object is not an iterator  
   
>>> I1 = iter(R)  
>>> next(I1)  
0  
>>> next(I1)  
1  
>>> I2 = iter(R) # Два итератора для одного диапазона  
>>> next(I2)  
0  
>>> next(I1) # I1 находится в другой позиции, не совпадающей с позицией I2  
2  
Функции zip, map и filter, напротив, не поддерживают возможность получения   
различных активных итераторов для одного и того же результата:  
>>> Z = zip((1, 2, 3), (10, 11, 12))  
>>> I1 = iter(Z)  
>>> I2 = iter(Z) # Два итератора для одного и того же результата zip

Новые итерируемые объекты в Python 3.0   
437  
>>> next(I1)  
(1, 10)  
>>> next(I1)  
(2, 11)  
>>> next(I2) # Позиции итераторов I2 и I1 совпадают!  
(3, 12)  
   
>>> M = map(abs, (-1, 0, 1)) # То же относится к функции map (и filter)  
>>> I1 = iter(M); I2 = iter(M)  
>>> print(next(I1), next(I1), next(I1))  
1 0 1  
>>> next(I2)  
StopIteration  
   
>>> R = range(3) # А для объекта диапазона можно получить   
>>> I1, I2 = iter(R), iter(R) # множество итераторов  
>>> [next(I1), next(I1), next(I1)]  
[0 1 2]  
>>> next(I2)  
0  
Когда мы будем учиться создавать собственные итерируемые объекты с помо-  
щью классов (глава 29), мы увидим, что поддержка множественных итераторов   
обычно обеспечивается за счет создания новых объектов, которые могут пере-  
даваться функции iter. Если поддерживается единственный итератор, функ-  
ция iter обычно возвращает тот же объект, который она получила. В главе 20   
мы также узнаем, что функции-генераторы и выражения-генераторы своим   
поведением напоминают скорее функции map и zip, чем range, поддерживая   
единственный активный итератор. В этой главе мы также познакомимся с не-  
которыми тонкостями использования единственных итераторов в циклах, ко-  
торые пытаются использовать их многократно.  
Итераторы представлений словарей  
Как мы уже видели в главе 8, в Python 3.0 методы keys, values и items слова-  
рей возвращают итерируемые объекты представлений, которые возвращают   
результаты по одному за раз вместо того, чтобы сразу создавать списки с ре-  
зультатами. Элементы представлений сохраняют физический порядок следо-  
вания в словаре, и все изменения, выполняемые в словаре, отражаются на его   
представлениях. Теперь мы знаем об итераторах достаточно много и готовы   
двигаться дальше:  
>>> D = dict(a=1, b=2, c=3)  
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2}  
   
>>> K = D.keys() # Объект представления в версии 3.0 не является списком  
>>> K  
<dict\_keys object at 0x026D83C0>  
   
>>> next(K) # Представления не являются итераторами  
TypeError: dict\_keys object is not an iterator  
   
>>> I = iter(K) # Из представлений можно получить итераторы  
>>> next(I) # и с их помощью выполнять итерации вручную,  
‘a’ # но они не поддерживают функцию len()

438   
Глава 14. Итерации и генераторы, часть 1   
>>> next(I) # и операцию доступа к элементам по индексу  
‘c’  
>>> for k in D.keys(): print(k, end=’ ‘) # Во всех итерационных контекстах  
... # итераторы используются   
a c b # автоматически  
Как и при работе с любыми другими итераторами, в версии 3.0 вы всегда мо-  
жете получить полный список элементов представления словаря, передав его   
встроенной функции list. Однако на практике этого обычно не требуется, за   
исключением случаев, когда требуется вывести все элементы в интерактивном   
сеансе или применить операции над списками, такие как доступ к элементам   
по индексу:  
>>> K = D.keys()  
>>> list(K) # При необходимости всегда можно получить полный список  
[‘a’, ‘c’, ‘b’]  
   
>>> V = D.values() # То же относится к представлениям values() и items()  
>>> V  
<dict\_values object at 0x026D8260>  
>>> list(V)  
[1, 3, 2]  
   
>>> list(D.items())  
[(‘a’, 1), (‘c’, 3), (‘b’, 2)]  
   
>>> for (k, v) in D.items(): print(k, v, end=’ ‘)  
...  
a 1 c 3 b 2  
Кроме того, в версии 3.0 словари по-прежнему поддерживают собственные ите-  
раторы, которые возвращают последовательности ключей. То есть в следую-  
щем контексте нет никакой необходимости вызывать метод keys:  
>>> D # Словари поддерживают собственные итераторы,  
{‘a’: 1, ‘c’: 3, ‘b’: 2} # возвращающие следующий ключ в каждой итерации  
>>> I = iter(D)  
>>> next(I)  
‘a’  
>>> next(I)  
‘c’  
>>> for key in D: print(key, end=’ ‘) # Нет никакой необходимости вызывать  
... # метод keys(), однако этот метод   
a c b # в версии 3.0 также возвращает итератор!  
В заключение напомню еще раз: теперь метод keys больше не возвращает спи-  
сок, поэтому традиционный способ обхода словарей по отсортированному спи-  
ску ключей больше не работает в версии 3.0. Вместо этого нужно преобразовать   
представление ключей в список с помощью функции list или вызвать функ-  
цию sorted, предав ей либо представление ключей, либо сам словарь, как по-  
казано ниже:  
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2}  
>>> for k in sorted(D.keys())): print(k, D[k], end=’ ‘)  
...  
a 1 b 2 c 3

Другие темы, связанные с итераторами   
439  
>>> D  
{‘a’: 1, ‘c’: 3, ‘b’: 2}  
>>> for k in sorted(D): print(k, D[k], end=’ ‘) # Лучший способ   
... # сортировки ключей  
a 1 b 2 c 3  
Другие темы, связанные с итераторами  
Еще больше о генераторах списков и об итераторах мы узнаем в главе 20, когда   
будем рассматривать их в контексте функций, и еще раз вернемся к ним в гла-  
ве 29, когда будем знакомиться с классами. Позднее вы узнаете, что:  
 •  
С помощью инструкции yield пользовательские функции можно превра-  
тить в итерируемые функции-генераторы.  
 •  
Генераторы списков можно трансформировать в итерируемые выражения-  
генераторы, заключив их в круглые скобки.  
 •  
В пользовательские классы можно добавить поддержку итераций с помо-  
щью методов перегрузки операторов \_\_iter\_\_ или \_\_getitem\_\_.  
В частности, реализация пользовательских итераторов с помощью классов по-  
зволяет обеспечить возможность использования произвольных объектов в лю-  
бых итерацонных контекстах, с которыми мы познакомились здесь.  
В заключение  
В этой главе мы исследовали некоторые концепции, имеющие отношение к ци-  
клам в языке Python. Мы впервые достаточно близко рассмотрели протокол   
итераций – способ, позволяющий использовать в циклах объекты, не являю-  
щиеся последовательностями, – и генераторы  списков. Мы узнали, что вы-  
ражения генераторов списков похожи на циклы for, которые применяют не-  
которое выражение ко всем элементам итерируемого объекта. Попутно мы по-  
знакомились с некоторыми другими инструментами итераций в действии и с   
новейшими изменениями, появившимися в версии 3.0.  
На этом мы заканчиваем обзор характерных процедурных инструкций и род-  
ственных им инструментов. Следующая глава завершает эту часть книги об-  
суждением возможностей документирования программного кода, имеющихся   
в языке Python, – документация также является частью общей синтаксиче-  
Python, – документация также является частью общей синтаксиче-  
, – документация также является частью общей синтаксиче-  
ской модели и важным компонентом правильно оформленной программы.   
Кроме того, в следующей главе приводится набор упражнений для этой части   
книги, которые желательно выполнить, прежде чем переходить к изучению   
к более крупным компонентам программ, таким как функции. Однако, как   
обычно, прежде чем двинуться дальше, вспомним, что мы узнали в этой главе.  
Закрепление пройденного  
Контрольные вопросы  
1. Как взаимосвязаны циклы for и итераторы?  
2. Как взаимосвязаны циклы for и генераторы списков?  
3. Назовите четыре разных контекста итераций в языке Python.

440   
Глава 14. Итерации и генераторы, часть 1   
4. Какой способ построчного чтения файлов считается наиболее оптималь-  
ным?  
5. Какое оружие вы ожидали бы увидеть в руках испанской инквизиции?  
Ответы  
1. Цикл for использует итерационный протокол для обхода элементов объ-  
екта. На каждой итерации он вызывает метод \_\_next\_\_ объекта (вызывается   
встроенной функцией next) и перехватывает исключение StopIteration, по   
которому определяет момент окончания итераций. Любой объект, поддер-  
живающий эту модель, может использоваться в циклах for и в других кон-  
текстах итераций.  
2. Оба они являются инструментами итераций. Генераторы списков представ-  
ляют простой и эффективный способ выполнения задачи, типичной для   
цик лов for: сбор результатов применения выражения ко всем элементам   
итерируемого объекта. Генераторы списков всегда можно преобразовать   
в цикл for, а кроме того, генераторы списков по своему внешнему виду напо-  
минают заголовок инструкции for.  
3. В число итерационных контекстов языка Python входят: цикл for, генера-  
торы списков, встроенная функция map, оператор in проверки вхождения,   
а также встроенные функции sorted, sum, any и all. В эту категорию также   
входят встроенные функции list и tuple, строковый метод join и операции   
присваивания последовательностей – все они следуют итерационному про-  
токолу (метод \_\_next\_\_) для обхода итерируемых объектов.  
4. Рекомендуемый в настоящее время способ чтения строк из текстового фай-  
ла – не читать файл явно вообще. Вместо этого предлагается открыть файл   
в итерационном контексте, например в цикле for или в генераторе списков   
и позволить итерационному инструменту на каждой итерации автоматиче-  
ски извлекать по одной строке из файла с помощью метода \_\_next\_\_. Такой   
подход считается более оптимальным в смысле простоты программирова-  
ния, скорости выполнения и использования памяти.  
5. Любой из следующих вариантов я приму как правильный ответ: устраше-  
ние, шантаж, хорошие красные униформы, удобная кушетка и мягкие по-  
душки.

Глава 15.  
   
Документация  
Эта часть книги завершается изучением приемов и инструментов, используе-  
мых для документирования программного кода на языке Python. Несмотря   
на то что программный код Python изначально обладает высокой удобочитае-  
мостью, некоторый объем уместно расположенных, внятных комментариев   
может существенно облегчить другим людям понимание принципа действия   
ваших программ. Язык Python включает синтаксические конструкции и ин-  
струменты, облегчающие документирование программ.  
Эта тема в большей степени связана с инструментальными средствами и, тем не   
менее, она рассматривается здесь, отчасти потому что она имеет некоторое от-  
ношение к синтаксической модели языка Python, а отчасти как источник све-  
дений для тех, кто изо всех сил пытается понять возможности языка Python.   
Преследуя эту последнюю цель, я дополню указания о документировании, ко-  
торые были даны в главе 4. Как обычно, эта глава завершается предупрежде-  
ниями о наиболее часто встречающихся ловушках, контрольными вопросами   
к главе и упражнениями к этой части книги.   
Источники документации в языке Python  
К настоящему моменту вы уже наверняка начинаете понимать, что Python из-  
начально включает в себя удивительно широкие функциональные возможно-  
сти – встроенные функции и исключения, предопределенные атрибуты и мето-  
ды объектов, модули стандартной библиотеки и многое другое. Более того, на   
самом деле мы лишь слегка коснулись каждой из этих категорий.  
Один из первых вопросов, который часто задают удивленные новички: «Как   
мне найти информацию обо всех встроенных возможностях?» Этот раздел рас-  
сказывает о различных источниках документации, доступных в языке Python.   
Здесь также будут представлены строки документирования (docstrings) и си-  
s) и си-  
) и си-  
стема PyDoc, которая использует их. Эти темы мало связаны с самим языком   
программирования, но они будут иметь большое значение, как только вы по-  
дойдете к примерам и упражнениям в этой части книги.  
Как показано в табл. 15.1, существует множество мест, где можно отыскать ин-  
формацию о Python, и объем этой информации все увеличивается. Поскольку

442   
Глава 15. Документация   
документация играет важную роль в практическом программировании, мы   
исследуем каждую из этих категорий в следующих разделах.  
Таблица 15.1. Источники документации в языке Python  
Форма  
Назначение  
Комментарии #  
Документация внутри файла  
Функция dir  
Получение списка атрибутов объектов  
Строки документирования: \_\_doc\_\_  
Документация внутри файла,   
присоединяемая к объектам  
PyDoc: функция help  
Интерактивная справка по объектам  
PyDoc: отчеты в формате HTML  
Документация к модулям для просмо-  
тра в броузере  
Стандартный набор руководств  
Официальное описание языка и биб-  
лиотеки  
Веб-ресурсы  
Интерактивные учебные руководства,   
примеры и так далее  
Печатные издания  
Руководства, распространяемые на   
коммерческой основе  
Комментарии #  
Комментарии, начинающиеся с символа решетки, представляют собой самый   
элементарный способ документирования программного кода. Интерпретатор   
просто игнорирует весь текст, который следует за символом # (при условии, что   
он находится не внутри строкового литерала), поэтому вы можете помещать   
вслед за этими символами слова и описания, предназначенные для програм-  
мистов. Впрочем, такие комментарии доступны только в файлах с исходными   
текстами – для записи комментариев, которые будут доступны более широко,   
следует использовать строки документирования.  
В настоящее время считается, что строки документирования лучше подходят   
для создания функционального описания (например, «мой файл делает то-то   
и то-то»), а комментарии, начинающиеся с символа #, лучше подходят для опи-  
сания некоторых особенностей программного кода (например, «это странное   
выражение делает то-то и то-то»). О строках документирования мы поговорим   
чуть ниже.  
Функция dir  
Функция dir – это простой способ получить список всех атрибутов объекта (то   
есть методов и элементов данных). Она может быть вызвана для любого объ-  
екта, который имеет атрибуты. Например, чтобы узнать, что имеется в стан-  
дартном библиотечном модуле sys, просто импортируйте его и передайте имя   
модуля функции dir (следующие результаты получены в Python 3.0, в версии   
Python 2.6 они немного отличаются):  
>>> import sys  
>>> dir(sys)

Источники документации в языке Python   
443  
[‘\_\_displayhook\_\_’, ‘\_\_doc\_\_’, ‘\_\_excepthook\_\_’, ‘\_\_name\_\_’, ‘\_\_package\_\_’,  
‘\_\_stderr\_\_’, ‘\_\_stdin\_\_’, ‘\_\_stdout\_\_’, ‘\_clear\_type\_cache’, ‘\_current\_frames’,   
‘\_getframe’, ‘api\_version’, ‘argv’, ‘builtin\_module\_names’, ‘byteorder’, ‘call\_  
tracing’, ‘callstats’, ‘copyright’, ‘displayhook’, ‘dllhandle’, ‘dont\_write\_  
bytecode’, ‘exc\_info’, ‘excepthook’, ‘exec\_prefix’, ‘executable’, ‘exit’, ‘flags’,   
‘float\_info’, ‘getcheckinterval’, ‘getdefaultencoding’, ...остальные имена   
опущены...]  
Здесь показаны только некоторые из имен� чтобы получить полный список,   
выполните эти инструкции на своей машине.  
Чтобы узнать, какие атрибуты содержат объекты встроенных типов, передайте   
функции dir литерал (или существующий объект) требуемого типа. Например,   
чтобы увидеть атрибуты списков и строк, можно передать функции пустой   
объект:  
>>> dir([])  
[‘\_\_add\_\_’, ‘\_\_class\_\_’, ...остальные имена опущены...  
‘append’, ‘count’, ‘extend’, ‘index’, ‘insert’, ‘pop’, ‘remove’,  
‘reverse’, ‘sort’]  
   
>>> dir(‘’)  
[‘\_\_add\_\_’, ‘\_\_class\_\_’, ‘\_\_contains\_\_’, ...остальные имена опущены...  
‘capitalize’, ‘center’, ‘count’, ‘encode’, ‘endswith’, ‘expandtabs’,  
‘find’, ‘format’, ‘index’, ‘isalnum’, ‘isalpha’, ‘isdecimal’, ‘isdigit’,   
‘isidentifier’, ‘islower’, ‘isnumeric’, ‘isprintable’, ‘isspace’, ‘istitle’,   
‘isupper’, ‘join’, ‘ljust’, ‘lower’, ‘lstrip’, ‘maketrans’, ‘partition’, ‘replace’,   
‘rfind’, ‘rindex’, ‘rjust’,...остальные имена опущены...]  
Результаты работы функции dir для любого встроенного типа включают набор   
атрибутов, которые имеют отношение к реализации этого типа (методы пере-  
груженных операторов)� все они начинаются и заканчиваются двумя символа-  
ми подчеркивания, чтобы сделать их отличными от обычных имен, и вы може-  
те пока просто игнорировать их.  
Того же эффекта можно добиться, передав функции dir имя типа вместо лите-  
рала:  
>>> dir(str) == dir(‘’) # Результат тот же, что и в предыдущем примере  
True  
>>> dir(list) == dir([])  
True  
Такой прием работает по той простой причине, что имена функций преобра-  
зования, такие как str и list, в языке Python фактически являются именами   
типов – вызов любого из этих конструкторов приводит к созданию экземпляра   
этого типа. Подробнее о конструкторах и о перегрузке операторов мы будем го-  
ворить в шестой части книги, когда будем обсуждать классы.  
Функция dir служит своего рода «кратким напоминанием» – она предостав-  
ляет список имен атрибутов, но ничего не сообщает о том, что эти имена озна-  
чают. За этой информацией необходимо обращаться к следующему источнику   
документации.  
Строки документирования: \_\_doc\_\_  
Помимо комментариев, начинающихся с символа #, язык Python поддерживает   
возможность создания документации, которая автоматически присоединяется

444   
Глава 15. Документация   
к объектам и доступна во время выполнения. Синтаксически такие строки рас-  
полагаются в начале файлов модулей, функций и классов, перед исполняемым   
программным кодом (перед ними вполне могут располагаться комментарии #).   
Интерпретатор автоматически помещает строки документирования в атрибут   
\_\_doc\_\_ соответствующего объекта.  
Строки документирования, определяемые пользователем  
В качестве примера рассмотрим следующий файл – docstrings.py. Строки до-  
кументирования в нем располагаются в самом начале файла, а также в на-  
чале функции и класса. Здесь для создания многострочных описаний файла   
и функции я использовал строки в тройных кавычках, но допускается исполь-  
зовать строки любого типа. Мы еще не познакомились с инструкциями def   
и class, поэтому вы можете просто игнорировать все, что находится после них,   
за исключением строк в самом начале:  
“””  
Module documentation  
Words Go Here  
“””  
   
spam = 40  
   
def square(x):  
 “””  
 function documentation  
 can we have your liver then?  
 “””  
 return x \*\*2  
   
class Employee:  
 “class documentation”  
 pass  
   
print(square(4))  
print(square.\_\_doc\_\_)  
Самое важное в протоколе документирования заключается в том, что ваши   
комментарии становятся доступны для просмотра в виде атрибутов \_\_doc\_\_   
после того, как файл будет импортирован. Поэтому, чтобы отобразить строки   
документирования, связанные с модулем и его объектами, достаточно просто   
импортировать файл и вывести значения их атрибутов \_\_doc\_\_, где интерпре-  
татор сохраняет текст:  
>>> import docstrings  
16  
 function documentation  
 can we have your liver then?  
>>> print(docstrings.\_\_doc\_\_)  
Module documentation  
Words Go Here  
>>> print(docstrings.square.\_\_doc\_\_)  
 function documentation

Источники документации в языке Python   
445  
 can we have your liver then?  
>>> print(docstrings.employee.\_\_doc\_\_)  
 class documentation  
Обратите внимание, что для вывода строк документирования необходимо явно   
использовать функцию print, в противном случае будет выводиться единствен-  
ная строка со встроенными символами новой строки.  
Кроме того, существует возможность присоединять строки документирования   
к методам классов (эта возможность описывается ниже), но так как они пред-  
ставлены инструкциями def, вложенными в классы, это не является особым   
случаем. Чтобы извлечь строку с описанием метода класса, определяемого вну-  
три модуля, необходимо указать имя модуля, класса и метода: module.class.meth-  
.class.meth-  
class.meth-  
.meth-  
meth-  
od.\_\_doc\_\_ (примеры строк документирования методов приводятся в главе 28).  
Стандарты оформления строк документирования  
Не существует какого-то общепринятого стандарта, который регламентировал   
бы, что должно входить в строки документирования (хотя в некоторых ком-  
паниях существуют свои внутренние стандарты). В свое время предлагались   
различные шаблоны и языки разметки (например, HTML или XML), но они не   
завоевали популярность в мире Python. И, положа руку на сердце, едва ли мы   
дождемся появления программистов, которые захотят писать документацию   
на языке разметки HTML!  
Вообще, среди программистов документация обычно отходит на задний план.   
Если вы увидите хоть какие-то комментарии в файле, считайте, что вам повез-  
ло. Однако я настоятельно рекомендую тщательно документировать свой про-  
граммный код – это действительно очень важная часть хорошо написанного   
программного кода. Замечу, что нет никаких стандартов на структуру строк   
документирования, поэтому, если вы хотите использовать их, чувствуйте себя   
свободными.  
Встроенные строки документирования  
Как оказывается, во встроенных модулях и объектах языка Python исполь-  
Python исполь-  
 исполь-  
зуется сходная методика присоединения документации – до и после списка   
атрибутов, возвращаемых функцией dir. Например, чтобы увидеть удобочита-  
емое описание встроенного модуля, его надо импортировать и вывести строку   
\_\_doc\_\_:  
>>> import sys  
>>> print(sys.\_\_doc\_\_)  
This module provides access to some objects used or maintained by the  
interpreter and to functions that interact strongly with the interpreter.  
   
Dynamic objects:  
   
argv -- command line arguments; argv[0] is the script pathname if known  
path -- module search path; path[0] is the script directory, else ‘’  
modules -- dictionary of loaded modules  
...остальной текст опущен...  
Описание функций, классов и методов внутри встроенных модулей присоеди-  
нено к их атрибутам \_\_doc\_\_:

446   
Глава 15. Документация   
>>> print(sys.getrefcount.\_\_doc\_\_)  
getrefcount(object) -> integer  
   
Return the reference count of object. The count returned is generally  
one higher than you might expect, because it includes the (temporary)  
...остальной текст опущен...  
Кроме того, можно прочитать описание встроенных функций, находящееся   
в их строках документирования:  
>>> print(int.\_\_doc\_\_)  
int(x[, base]) -> integer  
   
Convert a string or number to an integer, if possible. A floating  
point argument will be truncated towards zero (this does not include a  
...остальной текст опущен...  
   
>>> print(map.\_\_doc\_\_)  
map(func, \*iterables) --> map object  
   
Make an iterator that computes the function using arguments from  
each of the iterables. Stops when the shortest iterable is exhausted.  
Просматривая таким способом строки документирования встроенных инстру-  
ментов, вы можете получить богатый объем информации, однако вам не требу-  
ется этого делать – эту информацию функция help, тема следующего раздела,   
предоставляет вам автоматически.  
PyDoc: функция help  
Методика использования строк документирования оказалась настолько удоб-  
ной, что теперь в состав Python входит инструмент, который упрощает их ото-  
Python входит инструмент, который упрощает их ото-  
 входит инструмент, который упрощает их ото-  
бражение. Стандартный инструмент PyDoc написан на языке Python, он умеет   
извлекать строки документирования вместе с информацией о структуре про-  
граммных компонентов и формировать из них удобно отформатированные   
отчеты различных типов. Существуют также дополнительные, свободно рас-  
пространяемые программные инструменты, позволяющие извлекать и форма-  
тировать строки документирования (включая инструменты, обеспечивающие   
поддержку разметки типа «structured text», – дополнительную информацию   
ищите в Сети), однако Python распространяется вместе с пакетом PyDoc, со-  
Python распространяется вместе с пакетом PyDoc, со-  
 распространяется вместе с пакетом PyDoc, со-  
PyDoc, со-  
, со-  
держащимся в стандартной библиотеке.  
Существуют различные способы запуска PyDoc, включая сценарий командной   
строки (за дополнительной информацией обращайтесь к руководству по библи-  
отеке Python). Два, пожалуй, самых заметных интерфейса к PyDoc – это встро-  
Python). Два, пожалуй, самых заметных интерфейса к PyDoc – это встро-  
). Два, пожалуй, самых заметных интерфейса к PyDoc – это встро-  
PyDoc – это встро-  
 – это встро-  
енная функция help и графический интерфейс к PyDoc для воспроизводства   
отчетов в формате HTML. Функция help вызывает PyDoc для создания про-  
PyDoc для создания про-  
 для создания про-  
стых текстовых отчетов (которые выглядят как страницы руководства в UNIX-  
подобных системах):  
>>> import sys  
>>> help(sys.getrefcount)  
Help on built-in function getrefcount:  
   
getrefcount(...)  
 getrefcount(object) -> integer  
   
 Return the reference count of object. The count returned is generally

Источники документации в языке Python   
447  
 one higher than you might expect, because it includes the (temporary)  
 ...остальной текст опущен...  
Обратите внимание: чтобы вызывать функцию help, не обязательно импорти-  
ровать модуль sys, но его необходимо импортировать, чтобы получить справку   
по модулю sys, – функция ожидает получить ссылку на объект. Для крупных   
объектов, таких как модули и классы, функция help делит выводимую инфор-  
мацию на множество разделов, часть из которых показана здесь. Запустите   
следующую команду в интерактивном сеансе, чтобы получить полный отчет:  
>>> help(sys)  
Help on built-in module sys:  
   
NAME  
 sys  
   
FILE  
 (built-in)  
   
MODULE DOCS  
 http://docs.python.org/library/sys  
   
DESCRIPTION  
 This module provides access to some objects used or maintained by the  
 interpreter and to functions that interact strongly with the interpreter.  
 ...остальной текст опущен...  
   
FUNCTIONS  
 \_\_displayhook\_\_ = displayhook(...)  
 displayhook(object) -> None  
   
 Print an object to sys.stdout and also save it in builtins.  
 ...остальной текст опущен...  
   
DATA  
 \_\_stderr\_\_ = <io.TextIOWrapper object at 0x0236E950>  
 \_\_stdin\_\_ = <io.TextIOWrapper object at 0x02366550>  
 \_\_stdout\_\_ = <io.TextIOWrapper object at 0x02366E30>  
 ...остальной текст опущен...  
Часть информации в этом отчете извлечена из строк документирования, а часть   
этих сведений (например, сигнатуры функций) – это информация о струк-  
туре программных компонентов, которую PyDoc извлекает автоматически,   
в результате анализа внутреннего устройства объектов. Кроме того, функция   
help может использоваться для получения сведений о встроенных функциях,   
методах и типах. Чтобы получить справку о встроенном типе, нужно просто   
передать функции имя типа (например, dict – для словарей, str – для строк,   
list – для списков). Вам будет предоставлен большой объем информации с опи-  
саниями всех методов, доступных для этого типа:  
>>> help(dict)  
Help on class dict in module builtins:  
   
class dict(object)  
 | dict() -> new empty dictionary.  
 | dict(mapping) -> new dictionary initialized from a mapping object’s  
 ...остальной текст опущен...  
   
>>> help(str.replace)

448   
Глава 15. Документация   
Help on method\_descriptor:  
   
replace(...)  
 S.replace (old, new[, count]) -> str  
   
 Return a copy of S with all occurrences of substring  
 ...остальной текст опущен...  
   
>>> help(ord)  
Help on built-in function ord in module builtins:  
   
ord(...)  
 ord(c) -> integer  
 Return the integer ordinal of a one-character string.  
Наконец функция help может извлекать информацию не только из встроен-  
ных, но и из любых других модулей. Ниже приводится отчет, полученный для   
файла docstrings.py, представленного выше. Здесь снова часть информации   
представлена строками документирования, а часть была получена автомати-  
чески, в результате исследования структуры объектов:  
>>> import docstrings  
>>> help(docstrings.square)  
Help on function square in module docstrings:  
   
square(x)  
 function documentation  
 can we have your liver then?  
   
>>> help(docstrings.Employee)  
Help on class Employee in module docstrings:  
   
class Employee(builtins.object)  
 | class documentation  
 |  
 | Data descriptors defined here:  
 ...остальной текст опущен...  
   
>>> help(docstrings)  
Help on module docstrings:  
   
NAME  
 docstrings  
   
FILE  
 c:\misc\docstrings.py  
   
DESCRIPTION  
 Module documentation  
 Words Go Here  
   
CLASSES  
 builtins.object  
 Employee  
   
 class Employee(builtins.object)  
 | class documentation  
 |  
 | Data descriptors defined here:  
 ...остальной текст опущен...

Источники документации в языке Python   
449  
FUNCTIONS  
 square(x)  
 function documentation  
 can we have your liver then?  
   
DATA  
 spam = 40  
PyDoc: отчеты в формате HTML  
Функция help прекрасно подходит для извлечения информации при работе   
в интерактивной оболочке. Однако для PyDoc существует и графический ин-  
PyDoc существует и графический ин-  
 существует и графический ин-  
терфейс (простой и переносимый сценарий Python/tkinter), с помощью которо-  
Python/tkinter), с помощью которо-  
/tkinter), с помощью которо-  
tkinter), с помощью которо-  
), с помощью которо-  
го можно создавать отчеты в формате HTML, доступные для просмотра в лю-  
HTML, доступные для просмотра в лю-  
, доступные для просмотра в лю-  
бом веб-броузере. В этом случае PyDoc может выполняться как локально, так   
и удаленно, в режиме клиент/сервер. Внутри отчетов автоматически создаются   
гиперссылки, которые позволят щелчком мыши перемещаться к описаниям   
взаимосвязанных компонентов в вашем приложении.  
Чтобы запустить PyDoc в этом режиме, сначала необходимо запустить поиско-  
PyDoc в этом режиме, сначала необходимо запустить поиско-  
 в этом режиме, сначала необходимо запустить поиско-  
вый механизм, графический интерфейс которого представлен на рис. 15.1. Сде-  
лать это можно, выбрав пункт меню Module Docs (Документация к модулям) в меню   
Python кнопки Пуск в �indows или запустив сценарий pydocgui.pyw в каталоге   
Tools/Scripts, где был установлен Python (также можно запустить сценарий   
pydoc.py с ключом –g, находящийся в подкаталоге Lib). Введите имя интере-  
сующего вас модуля и нажмите клавишу Enter – PyDoc обойдет каталоги в пути   
поиска модулей (sys.path) и отыщет ссылки на указанный модуль.  
Рис. 15.1. Главное окно графического интерфейса PyDoc: введите имя требуе-  
PyDoc: введите имя требуе-  
: введите имя требуе-  
мого модуля, нажмите клавишу Enter, выберите модуль и затем щелкните   
на кнопке «go to selected» (или, не вводя имя модуля, щелкните на кнопке   
«open browser», чтобы увидеть список всех доступных модулей)  
Отыскав нужную запись, выберите ее и щелкните на кнопке «go to selected»   
(перейти к выбранному элементу). PyDoc откроет веб-броузер и отобразит отчет   
в формате HTML. На рис. 15.2 показано, как выглядит информация, представ-  
HTML. На рис. 15.2 показано, как выглядит информация, представ-  
. На рис. 15.2 показано, как выглядит информация, представ-  
ленная PyDoc, для встроенного модуля glob.  
Обратите внимание на гиперссылки в разделе «Modules» (модули) на этой стра-  
Modules» (модули) на этой стра-  
» (модули) на этой стра-  
нице – вы можете щелкать на них мышью и перемещаться на страницы с опи-

450   
Глава 15. Документация   
саниями этих (импортированных) модулей. Для больших страниц PyDoc так-  
PyDoc так-  
 так-  
же генерирует гиперссылки на различные разделы на этой странице.  
Подобно функции help, графический интерфейс может извлекать информацию   
и из пользовательских модулей. На рис. 15.3 показана страница с информаци-  
ей, извлеченной из нашего файла docstrings.py.  
PyDoc можно настраивать и запускать разными способами, но мы не будем   
рассматривать эти возможности здесь – за дополнительной информацией об-  
ращайтесь к руководству по стандартной библиотеке языка Python. Главное,   
что вы должны запомнить, – PyDoc по сути создает отчеты о реализации на   
основе той информации, что имеется, – если вы использовали строки докумен-  
тирования в своих файлах, PyDoc сделает все необходимое, чтобы собрать и от-  
PyDoc сделает все необходимое, чтобы собрать и от-  
 сделает все необходимое, чтобы собрать и от-  
форматировать их соответствующим образом. PyDoc – это всего лишь средство   
получения справки об объектах, таких как функции и модули, но он обеспечи-  
вает простой доступ к документации с описанием этих компонентов. Его отче-  
ты более полезны, чем просто списки атрибутов, хотя и менее исчерпывающи,   
чем стандартные руководства.  
Рис. 15.2. Отыскав требуемый модуль в окне, представленном на рис. 15.1,   
щелкните на кнопке «go to selected», и описание модуля в формате HTML бу-  
go to selected», и описание модуля в формате HTML бу-  
 to selected», и описание модуля в формате HTML бу-  
to selected», и описание модуля в формате HTML бу-  
 selected», и описание модуля в формате HTML бу-  
selected», и описание модуля в формате HTML бу-  
», и описание модуля в формате HTML бу-  
HTML бу-  
 бу-  
дет отображено в веб-броузере, как в данном случае, где отображено описание   
модуля из стандартной библиотеки

Источники документации в языке Python   
451  
Рис. 15.3. PyDoc может служить источником документации как для встро-  
енных, так и для пользовательских модулей. Здесь приводится страница   
с описанием пользовательского модуля, где можно видеть все строки доку-  
ментирования (docstrings), извлеченные из файла с исходными текстами  
Совет  дня: если поле ввода имени модуля в окне на рис. 15.1   
оставить пустым и щелкнуть на кнопке «open browser» (открыть   
броузер), PyDoc воспроизведет веб-страницу с гиперссылками   
на все модули, доступные для импорта на данном компьютере.   
Сюда входят модули стандартной библиотеки, расширения сто-  
ронних производителей, пользовательские модули, располо-  
женные в пути поиска импортируемых модулей, и даже модули,   
написанные на языке C, скомпонованные статически или дина-  
мически. Такую информацию сложно получить иными путями,   
если не писать свой программный код, который будет занимать-  
ся исследованием исходных текстов набора модулей.  
Кроме того, PyDoc может сохранять документацию в формате   
HTML для последующего просмотра или вывода на печать� ука-  
зания о том, как это сделать, вы найдете в документации. Сле-  
дует отметить, что PyDoc может не совсем корректно работать   
со сценариями, которые читают данные из потока стандартного

452   
Глава 15. Документация   
ввода, – PyDoc импортирует целевой модуль для последующе-  
го исследования, но при работе в режиме с графическим ин-  
терфейсом может отсутствовать связь с потоком стандартного   
ввода. Однако модули, которые не требуют немедленного ввода   
информации в момент импортирования, будут обслуживаться   
корректно.  
Стандартный набор руководств  
Стандартные руководства играют роль наиболее полного и самого свежего опи-  
сания языка Python и набора инструментальных средств. Руководства распро-  
страняются в формате HTML и в других форматах и в �indows устанавлива-  
�indows устанавлива-  
 устанавлива-  
ются вместе системой Python – они доступны в виде пунктов подменю Python,   
в меню кнопки Пуск (Start), а также в меню Help (Справка) среды разработки IDLE.   
Набор руководств можно как получить отдельно, в различных форматах, по   
адресу http://www.python.org, так и читать непосредственно на сайте (следуй-  
те по ссылке Documentation (документация)). Руководства в системе �indows   
оформлены в виде файлов справки, поддерживающих возможность поиска�   
электронная версия на сайте проекта Python также имеет страницу поиска.  
После открытия руководства в операционной системе �indows оно отобража-  
�indows оно отобража-  
 оно отобража-  
ет начальную страницу, как показано на рис. 15.4. Двумя самыми важными,   
пожалуй, здесь являются ссылки Library Reference (справочное руководство по   
библиотеке, где описываются встроенные типы, функции, исключения и мо-  
дули стандартной библиотеки) и Language Reference (справочное руководство по   
языку, где приводится формальное описание языковых конструкций). На этой   
странице имеется также ссылка Tutorial (самоучитель), которая ведет к кратко-  
му введению для начинающих изучение языка.  
Веб-ресурсы  
На официальном веб-сайте проекта Python (http://www.python.org) вы найдете   
ссылки на различные ресурсы, посвященные этому языку программирования,   
часть которых охватывает специализированные темы и области применения   
языка. Щелкнув на ссылке Documentation (Документация), можно получить до-  
ступ к электронному учебнику и к руководству «Beginners Guide to Python»   
(руководство по языку Python для начинающих). На сайте также имеются   
ссылки на другие ресурсы на других языках.  
Массу информации о языке Python можно отыскать в интернет-энциклопедии,   
в блогах, на веб-сайтах и других ресурсах в Сети. Чтобы получить перечень ссы-  
лок на такие ресурсы, попробуйте поискать по строке «Python programming»   
в поисковой системе Google.  
Печатные издания  
Последний источник информации – это огромная коллекция печатных спра-  
вочных пособий по языку Python. Однако учтите, что обычно книги немного   
отстают от развития языка Python, частично из-за того, что для написания   
книги необходимо время, частично из-за естественных задержек, свойствен-  
ных самому процессу издания. Обычно книга выходит в свет с отставанием на

Типичные ошибки программирования   
453  
три или более месяцев от текущего состояния дел. В отличие от стандартных   
руководств, книги редко распространяются бесплатно.  
Однако для многих удобство и качество профессионального издания с лихвой   
окупает потраченные деньги. Более того, язык Python изменяется настолько   
медленно, что книги сохраняют свою актуальность в течение еще нескольких   
лет после их издания, особенно если их авторы публикуют дополнения в Сети.   
Ссылки на другие книги о Python вы найдете в предисловии.  
Типичные ошибки программирования  
Прежде чем перейти к упражнениям этой части книги, рассмотрим некоторые   
наиболее распространенные ошибки, которые допускают начинающие про-  
граммисты в инструкциях и программах на языке Python. Многие из этих   
ошибок, уже упоминавшиеся ранее в этой части книги, я привел здесь для   
полноты картины. С ростом опыта использования языка Python вы научитесь   
избегать их, но несколько слов, сказанных сейчас, помогут вам избегать их   
с самого начала:  
Рис. 15.4. Стандартный набор руководств по языку Python, доступный   
на сайте www.python.org в меню «Help» (справка) среды разработки IDLE   
и в меню кнопки «Пуск» («Start») в операционной системе Windows.    
В Windows набор руководств оформлен в виде файла справки, поддерживаю-  
Windows набор руководств оформлен в виде файла справки, поддерживаю-  
 набор руководств оформлен в виде файла справки, поддерживаю-  
щего возможность поиска; электронная версия на веб-сайте также имеет   
страницу поиска. Из всех предлагаемых руководств самым востребованным   
является «Library reference» (справочное руководство по библиотеке)

454   
Глава 15. Документация   
 •  
Не забывайте про двоеточия. Никогда не забывайте вводить символ двое-  
точия в конце заголовков составных инструкций (первая строка таких ин-  
струкций, как if, while, for и других). Сначала вы наверняка будете забы-  
вать об этом (как я и большинство из 3000 моих студентов), но вскоре это   
превратится для вас в привычку.  
 •  
Начинайте с первой позиции в строке. Программный код верхнего уровня   
(не вложенный) должен начинаться с первой позиции в строке. Сюда отно-  
сится как не вложенный программный код в модулях, так и программный   
код, который вводится в интерактивной оболочке.  
 •  
Пустые строки имеют особый смысл в интерактивной оболочке. Пустые   
строки в теле составных инструкций внутри файлов модулей всегда игно-  
рируются, но когда программный код вводится в интерактивной оболочке,   
они завершают составные инструкции. Другими словами, ввод пустой стро-  
ки сообщает интерактивной командной оболочке, что вы закончили ввод   
составной инструкции, – если вам необходимо продолжить ввод такой ин-  
струкции, не нажимайте клавишу Enter, когда отображается строка пригла-  
шения к вводу ... (или в IDLE), пока вы действительно не закончите ее ввод.  
 •  
Используйте отступы непротиворечивым способом. Старайтесь не смеши-  
вать символы табуляции и пробелы при оформлении отступов в блоке, если   
вы не знаете точно, как текстовый редактор интерпретирует символы табу-  
ляции. В противном случае интерпретатор Python будет видеть совсем не   
то, что вы видите на экране, когда он будет выполнять преобразование сим-  
волов табуляции в пробелы. Это справедливо не только для Python, но и для   
любого другого языка программирования с блочно-структурированным   
оформлением программного кода – если у другого программиста в тексто-  
вом редакторе ширина символов табуляции настроена иначе, он не сможет   
понять структуру вашего программного кода. Для оформления отступов   
лучше использовать что-то одно – или символы табуляции, или пробелы.  
 •  
Не пишите на языке C. Напоминаю программистам, использующим C/  
C++: нет никакой необходимости заключать условные выражения в кру-  
глые скобки в инструкциях if и while (например, if (X == 1):). Это допустимо   
(любое выражение можно заключить в круглые скобки), но в данном кон-  
тексте они совершенно излишни. Кроме того, не заканчивайте все инструк-  
ции точками с запятой – это также вполне допустимо в языке Python, но   
они совершенно бесполезны, если в каждой строке находится всего одна   
инструкция (конец строки обычно обозначает конец инструкции). И пом-  
ните – не встраивайте инструкции присваивания в условные выражения   
циклов while и не заключайте блоки в фигурные скобки {} (вложенные бло-  
ки оформляются с помощью отступов).  
 •  
Вместо циклов while и функции range старайтесь использовать простые   
циклы for. Еще одно напоминание: простые циклы for (например, for x in   
seq:) практически всегда проще и выполняются быстрее, чем счетные ци-  
клы while или основанные на использовании функции range. Так как в про-  
стых циклах for извлечение элементов последовательностей производится   
внутренними механизмами интерпретатора, они выполняются порой в два   
раза быстрее, чем эквивалентные циклы while. Избегайте искушения счи-  
тать что-либо в циклах на языке Python!  
 •  
Будьте внимательны, выполняя присваивание изменяемых объектов. Об   
этом уже говорилось в главе 11: следует быть особенно внимательным при

Типичные ошибки программирования   
455  
использовании изменяемых объектов в инструкциях множественного при-  
сваивания (a = b = []), а также в комбинированных инструкциях присваива-  
ния (a += [1, 2]). В обоих случаях непосредственные изменения могут затро-  
нуть другие переменные. Более подробно об этом рассказывается в главе 11.  
 •  
Не ожидайте получения результатов от функций, выполняющих непо-  
средственные изменения в объектах. Мы уже сталкивались с этим ранее:   
операции, выполняющие непосредственное изменение, такие как методы   
list.append и list.sort, представленные в главе 8, не имеют возвращаемых   
значений (кроме None), поэтому их следует вызывать без присваивания воз-  
вращаемого значения. Начинающие программисты часто допускают ошиб-  
ку, используя примерно такой программный код: mylist = mylist.append(X),   
пытаясь получить результат метода append, но в действительности в этом   
случае в переменную mylist записывается ссылка на объект None, а не на   
измененный список (фактически такая инструкция ведет к полной потере   
ссылки на список).  
Менее явный пример такой ошибки в Python 2.X – когда выполняется по-  
Python 2.X – когда выполняется по-  
 2.X – когда выполняется по-  
X – когда выполняется по-  
 – когда выполняется по-  
пытка обойти элементы словаря в порядке сортировки. Очень часто мож-  
но увидеть, например, такой программный код: for k in D.keys().sort():. Он   
почти работает – метод keys создает список ключей, а метод sort упорядо-  
чивает его, но так как метод sort возвращает объект None, цикл for терпит   
неудачу, потому что, в конечном счете, выполняется попытка обойти эле-  
менты объекта None (который не является последовательностью). Данный   
способ терпит неудачу даже в Python 3.0, потому что метод keys возвраща-  
ет объект представления, а не список! Этот алгоритм можно реализовать   
либо с помощью новой встроенной функции sorted, которая возвращает   
отсортированный список, либо необходимо разделить вызовы методов на   
инструкции: Ks = list(D.keys()), затем Ks.sort(), и наконец for k in Ks:. Это   
один из случаев, когда может потребоваться явный вызов метода keys для   
организации обхода элементов словаря в цикле вместо использования ите-  
раторов словарей, так как итераторы не выполняют сортировку.  
 •  
Всегда используйте круглые скобки при вызове функций. При вызове   
функций после их имен всегда следует добавлять круглые скобки незави-  
симо от наличия входных аргументов (например, вызов функции должен   
выглядеть как function(), а не function). В четвертой части книги вы узнае-  
те, что функции – это простые объекты, которые могут выполнять специ-  
альную операцию – вызов – при обращении к имени с круглыми скобками.  
Похоже, что эта проблема наиболее часто возникает в классах при рабо-  
те с файлами – нередко можно увидеть, как начинающие программисты   
пытаются оформить вызов метода как file.close, а не как file.close(). По-  
скольку обращение к имени метода без круглых скобок в языке Python счи-  
тается допустимым, такая попытка не приводит к появлению ошибки, но   
файл остается открытым!  
 •  
Не используйте расширения имен файлов в инструкциях import и reload.   
Не указывайте полные пути к файлам и расширения в инструкциях import   
(например, следует писать import mod, а не import mod.py). (Начальные сведе-  
ния о модулях приводились в главе 3, и мы будем еще обсуждать их в пятой   
части книги.) Так как помимо .py имена файлов модулей могут иметь дру-  
гие расширения (например, .pyc), указание расширения не только является   
нарушением синтаксиса, но и вообще не имеет смысла. Синтаксис опреде-

456   
Глава 15. Документация   
ления пути зависит от типа платформы и определяется настройками пара-  
метра пути поиска модулей, а не инструкцией import.  
В заключение  
В этой главе мы рассмотрели вопросы документирования программ, которые   
касаются как документации, которую мы пишем для наших собственных   
программ, так и документации к встроенным инструментам. Мы познакоми-  
лись со строками документирования, с ресурсами, содержащими справочные   
руководства по языку Python, и узнали, как с помощью функции help и веб-  
интерфейса PyDoc получить доступ к дополнительной документации. Так как   
это последняя глава в этой части книги, мы также рассмотрели наиболее часто   
встречающиеся ошибки, что должно помочь вам избежать их.  
В следующей части книги мы начнем применять полученные знания к более   
крупным программным конструкциям: к функциям. Однако прежде чем дви-  
нуться дальше, проработайте упражнения к этой части, которые приводятся   
в конце главы. Но перед этим ответьте на контрольные вопросы к главе.  
Закрепление пройденного  
Контрольные вопросы  
1. Когда вместо комментариев, начинающихся с символа решетки, следует ис-  
пользовать строки документирования?  
2. Назовите три способа извлечения строк документирования.  
3. Как получить перечень всех атрибутов объекта?  
4. Как можно получить перечень всех модулей, доступных на компьютере?  
5. Какие книги о Python, после этой, следует приобрести?  
Ответы  
1. Строки документирования считаются более удобными для создания функ-  
ционального описания, где поясняются принципы использования модулей,   
функций, классов и методов. Комментарии, начинающиеся с символа ре-  
шетки, лучше подходят для пояснений к выражениям и инструкциям. Та-  
кой порядок принят не только потому, что строки документирования проще   
отыскать в файле с исходными текстами, но и потому, что они могут извле-  
каться и просматриваться с помощью системы PyDoc.  
2. Получить содержимое строк документирования можно с помощью атрибу-  
та \_\_doc\_\_ объекта, передав его функции help, или выбирая модули в поис-  
ковой системе PyDoc с графическим интерфейсом, в режиме клиент/сервер.   
Дополнительно PyDoc обладает возможностью сохранять описание моду-  
лей в файлах HTML для последующего просмотра.  
3. Список всех атрибутов, имеющихся у любого объекта, можно получить   
с помощью функции dir(X).  
4. Запустите графический интерфейс PyDoc, оставьте пустым поле ввода име-  
ни модуля и щелкните на кнопке Оpen browser (Открыть броузер). В результате

Закрепление пройденного   
457  
будет открыта веб-страница, содержащая ссылки на описания всех моду-  
лей, доступных вашей программе.  
5. Мои, конечно. (А если серьезно, в предисловии имеется перечень некоторых   
книг, как справочных пособий, так и учебников, рекомендуемых для даль-  
нейшего прочтения.)  
Упражнения к третьей части  
Теперь, когда вы узнали, как описывается логика работы программы, в сле-  
дующих упражнениях вам будет предложено реализовать решение некото-  
рых простых задач с использованием инструкций. Самым большим является   
упражнение 4, где вам будет предложено рассмотреть альтернативные вариан-  
ты реализации. Одна и та же задача всегда может быть решена разными спо-  
собами, и отчасти изучение языка Python заключается в том, чтобы находить   
более оптимальные решения.  
Решения приводятся в приложении B, в разделе «Часть III».  
1. Основы циклов.  
a. Напишите цикл for, который выводит ASCII-коды всех символов в стро-  
ке с именем S. Для преобразования символов в целочисленные ASCII-  
коды используйте встроенную функцию ord(character). (Поэксперимен-  
тируйте с ней в интерактивной оболочке, чтобы понять, как она рабо-  
тает.)  
b. Затем измените цикл так, чтобы он вычислял сумму кодов ASCII всех   
символов в строке.  
c. Наконец, измените свой программный код так, чтобы он возвращал но-  
вый список, содержащий ASCII-коды всех символов в строке. Дает ли   
выражение map(ord, S) похожий результат? (Подсказка: прочитайте гла-  
ву 14.)  
2. Символы обратного слеша. Что произойдет, если в интерактивной оболочке   
ввести следующий программный код?  
for i in range(50):  
 print ‘hello %d\n\a’ % i  
Будьте осторожны при запуске этого примера не в среде IDLE, он может   
сгенерировать звуковой сигнал, что может не понравиться окружающим.   
Среда разработки IDLE вместо этого выводит малопонятные символы (сим-  
волы, экранированные обратным слешем, приводятся в табл. 7.2).  
3. Сортировка словарей. В главе 8 мы видели, что словари представляют со-  
бой неупорядоченные коллекции. Напишите цикл for, который выводит   
элементы словаря в порядке возрастания. (Подсказка: используйте метод   
keys словаря и метод списка sort или новую встроенную функцию sorted.)  
4. Программирование  альтернативной  логики. Изучите следующий фраг-  
мент, где для поиска числа 2 в пятой степени (32) в списке степеней числа   
2, используется цикл while и флаг found. Этот фрагмент хранится в файле   
power.py.  
L = [1, 2, 4, 8, 16, 32, 64]  
X = 5

458   
Глава 15. Документация   
found = False  
i = 0  
while not found and i < len(L):  
 if 2 \*\* X == L[i]:  
 found = 1  
 else:  
 i = i+1  
if found:  
 print(‘at index’, i)  
else:  
 print(X, ‘not found’)  
C:\book\tests> python power.py  
at index 5  
В этом примере не используются обычные приемы программирования, при-  
нятые в языке Python. Следуя указаниям ниже, попробуйте улучшить его   
(вы можете вносить изменения в интерактивной оболочке или сохранять   
в файле сценария и запускать его из командной строки системы – использо-  
вание файла существенно упростит это упражнение):  
a. Сначала добавьте в цикл while блок else, чтобы избавиться от флага found   
и последней инструкции if.  
b. Затем перепишите пример с циклом for и блоком else, чтобы избавиться   
от логики вычисления индексов в списке. (Подсказка: получить индекс   
элемента можно с помощью метода index (L.index(X)), возвращающего   
смещение первого элемента со значением X в списке.)  
c. Затем вообще избавьтесь от цикла, реализовав решение на основе опе-  
ратора in проверки вхождения. (Подробности вы найдете в главе 8 или   
попробуйте ввести такое выражение: 2 in [1,2,3].)  
d. Наконец, вместо литерала списка L используйте цикл for и метод append   
для заполнения списка степеней двойки.  
Более глубокие улучшения:  
e. Как вы думаете, повысится ли производительность, если выражение   
2 \*\* X вынести за пределы циклов? Как это можно сделать?  
f. Как мы видели в упражнении 1, Python включает в себя функцию   
map(function, list), которая может создать список степеней числа 2:   
map(lambda x: 2 \*\* x, range(7)). Попробуйте выполнить этот программный   
код в интерактивной оболочке� с инструкцией lambda мы познакомимся   
в главе 19.

Часть IV.  
Функции

Глава 16.  
   
Основы функций  
В третьей части книги мы рассмотрели основные процедурные инструкции   
языка Python. В этой части мы переходим к исследованию набора дополни-  
тельных инструкций, которые используются при создании функций.   
Если говорить просто, то функция – это средство, позволяющее группировать   
наборы инструкций так, что в программе они могут запускаться неоднократно.   
Функции могут вычислять некоторый результат и позволять указывать вход-  
ные параметры, отличающиеся по своим значениям от вызова к вызову. Воз-  
можность оформления операций в виде функций – это очень удобный инстру-  
мент, который мы можем использовать в самых разных ситуациях.   
С принципиальной точки зрения функции устраняют необходимость встав-  
лять в программу избыточные копии блоков одного и того же программного   
кода, так как они могут быть заменены единственной функцией. Благодаря   
функциям можно существенно уменьшить трудозатраты на программирова-  
ние: если операцию необходимо будет видоизменить, достаточно будет внести   
изменения всего в одном месте, а не во многих.  
Функции – это самые основные программные структуры в языке Python, обе-  
спечивающие многократное использование программного кода и уменьшаю-  
щие его избыточность. Как будет показано далее, функции – это еще и сред-  
ство проектирования, которое позволяет разбить сложную систему на доста-  
точно простые и легко управляемые части. В табл. 16.1 приводятся основные   
инструменты, имеющие отношение к функциям, которые мы будем изучать   
в этой части книги.  
Таблица 16.1. Инструкции и выражения, имеющие отношение к функциям  
Инструкция  
Примеры  
Вызов  
myfunc(‘spam’, ‘eggs’, meat=ham)  
def, return  
def adder(a, b=1, \*c):  
 return a+b+c[0]  
global  
def changer():  
 global x; x = ‘new’

462   
Глава 16. Основы функций   
Инструкция  
Примеры  
nonlocal  
def changer():  
 nonlocal x; x = ‘new’  
yield  
def squares(x):  
 for i in range(x): yield i \*\* 2  
lambda  
funcs = [lambda x: x\*\*2, lambda x: x\*3]  
Зачем нужны функции?  
Прежде чем перейти к обсуждению деталей, мы нарисуем себе четкую карти-  
ну, что из себя представляют функции. Функции – это практически универ-  
сальное средство структурирования программы. Возможно, раньше вам уже   
приходилось сталкиваться с ними в других языках программирования, где   
они могли называться подпрограммами или процедурами. В процессе разра-  
ботки функции играют две основные роли:  
Максимизировать  многократное  использование  программного  кода  и  мини-  
мизировать его избыточность   
Как и в большинстве других языков программирования, функции в языке   
Python представляют собой простейший способ упаковки логики выпол-  
нения, которая может использоваться в разных местах программы и более   
чем один раз. До сих пор весь программный код, который нам приходи-  
лось писать, выполнялся немедленно. Функции позволяют группировать   
и обобщать программный код, который может позднее использоваться про-  
извольное число раз. Так как функции позволяют поместить реализацию   
операции в одно место и использовать ее в разных местах, они являются   
самым основным инструментом структуризации: они дают возможность   
уменьшить избыточность программного кода и тем самым уменьшить тру-  
дозатраты на его сопровождение.  
Процедурная декомпозиция  
Функции также обеспечивают возможность разбить сложную систему на   
части, каждая из которых играет вполне определенную роль. Например,   
чтобы испечь пиццу, сначала нужно замесить тесто, раскатать его, добавить   
начинку, испечь и так далее. Если бы мы писали программу для машины   
по выпечке пиццы, мы могли бы общую задачу «испечь пиццу» разбить на   
мелкие части – по одной функции для каждого из этапов. Гораздо проще   
создать решение маленьких задач по отдельности, чем реализовать весь   
процесс целиком. Вообще функции описывают «как делать», а не «зачем де-  
лать». В шестой части книги мы увидим, почему это различие имеет такое   
большое значение.  
В этой части книги мы исследуем понятия языка Python, используемые при   
создании функций: основы функций, правила области видимости и передача   
аргументов, а также ряд сопутствующих концепций, таких как генераторы   
и функциональные инструменты. Мы также еще раз вернемся к понятию по-  
лиморфизма, введенному ранее в этой книге, поскольку на данном уровне его   
Таблица 16.1 (продолжение)

Создание функций   
463  
важность становится еще более очевидной. Как вы увидите, функции привно-  
сят не так много новых синтаксических конструкций, но они ведут нас к более   
существенным идеям программирования.  
Создание функций  
Несмотря на то что функции еще не были представлены формально, тем не ме-  
нее мы уже использовали некоторые из них в предыдущих главах. Например,   
для создания объекта файла мы вызывали функцию open� точно так же мы ис-  
пользовали встроенную функцию len, когда нам необходимо было узнать число   
элементов в объекте коллекции.  
В этой главе мы узнаем, как создаются новые функции в языке Python. Функ-  
ции, которые мы пишем сами, ведут себя точно так же, как и встроенные   
функции, которые нам уже приходилось встречать: они могут вызываться   
в выражениях, получать значения и возвращать результаты. Но для того что-  
бы создавать новые функции, необходимо ввести дополнительные понятия.   
Кроме того, в языке Python функции ведут себя иначе, чем в компилирующих   
языках программирования, таких как C. Ниже приводится краткое введение   
в основные концепции, составляющие основу функций в языке Python, каж-  
дую из которых мы будем изучать в этой части книги:  
 •  
def – это исполняемый программный код. Функции в языке Python созда-  
ются с помощью новой инструкции def. В отличие от функций в компили-  
рующих языках программирования, таких как C, def относится к классу   
исполняемых инструкций – функция не существует, пока интерпретатор   
не доберется до инструкции def и не выполнит ее. Фактически вполне допу-  
стимо (а иногда даже полезно) вкладывать инструкции def внутрь инструк-  
ций if, циклов while и даже в другие инструкции def. В случае наиболее   
типичного использования инструкции def вставляются в файлы модулей   
и генерируют функции при выполнении во время первой операции импор-  
тирования.  
 •  
def создает объект и присваивает ему имя. Когда интерпретатор Python   
встречает и выполняет инструкцию def, он создает новый объект-функцию   
и связывает его с именем функции. Как и в любой другой операции при-  
сваивания, имя становится ссылкой на объект-функцию. В имени функции   
нет ничего необычного – как будет показано далее, объект-функция может   
быть связан с несколькими именами, может сохраняться в списке и так да-  
лее. Кроме того, к функциям можно прикреплять различные атрибуты,   
определяемые пользователем, для сохранения каких-либо данных.  
 •  
Выражение lambda создает объект и возвращает его в виде результата.   
Функции могут также создаваться с помощью выражения lambda. Это по-  
Это по-  
зволяет создавать встроенные определения функций там, где синтаксис   
языка не позволяет использовать инструкцию def (это достаточно сложная   
концепция, рассмотрение которой мы отложим до главы 19).  
 •  
return передает объект результата вызывающей программе. Когда функция   
вызывается, вызывающая программа приостанавливает свою работу, пока   
функция не завершит работу и не вернет управление. Функции, вычисляю-  
щие какое-либо значение, возвращают его с помощью инструкции return –   
возвращаемое значение становится результатом обращения к функции.

464   
Глава 16. Основы функций   
 •  
yield передает объект результата вызывающей программе и запоминает,   
где был произведен возврат. Функции, известные как генераторы, для   
передачи возвращаемого значения могут также использовать инструкцию   
yield и сохранять свое состояние так, чтобы работа функции могла быть   
возобновлена позднее, – это еще одна из сложных тем, которые будут рас-  
сматриваться позже в этой части книги.  
 •  
Аргументы передаются посредством присваивания (в виде ссылок на объ-  
екты). В языке Python аргументы передаются функциям посредством вы-  
Python аргументы передаются функциям посредством вы-  
 аргументы передаются функциям посредством вы-  
полнения операции присваивания (что, как мы уже знаем, означает – в виде   
ссылок на объекты). Как будет показано далее, модель, принятая в языке   
Python, в действительности не эквивалентна правилам передачи аргумен-  
, в действительности не эквивалентна правилам передачи аргумен-  
тов по ссылке в языке C или C++ – и вызывающая программа, и функция   
совместно используют ссылку на объект, но здесь нет никакого совмещения   
имен. Изменение имени аргумента также не изменяет имени в вызывающей   
программе, но модификация изменяемых объектов внутри функции может   
приводить к изменению объектов в вызывающей программе.   
 •  
global объявляет переменные, глобальные для модуля, без присваивания   
им значений. По умолчанию все имена, присваивание которым произво-  
дится внутри функций, являются локальными для этих функций и суще-  
ствуют только во время выполнения функций. Чтобы присвоить значение   
имени в объемлющем модуле, функция должна объявить его с помощью   
инструкции global. Говоря в более широком смысле, поиск имен всегда про-  
изводится в некоторой области видимости – там, где хранятся перемен-  
ные, – а операция присваивания связывает имена с областями видимости.  
 •  
nonlocal объявляет переменные, находящиеся в области видимости объ-  
емлющей функции, без присваивания им значений. В Python 3 появилась   
новая инструкция nonlocal, позволяющая функциям присваивать значения   
переменным, находящимся в области видимости синтаксически объемлю-  
щей функции. Это позволяет использовать объемлющие функции, как ме-  
сто хранения информации о состоянии – информация восстанавливается   
в момент вызова функции, при этом отпадает необходимость использовать   
глобальные переменные.  
 •  
Аргументы получают свои значения (ссылки на объекты) в результате вы-  
полнения операции присваивания. При передаче аргументов функциям   
в языке Python выполняется операция присваивания значений (то есть, как   
мы уже знаем, ссылок на объекты). Как вы увидите далее, в языке Python   
передача объектов в функции производится по ссылкам, но это не означает,   
что создаются псевдонимы имен. Изменение имени аргумента внутри функ-  
ции не влечет за собой изменения соответствующего имени в вызывающей   
программе, но изменения в изменяемых объектах, переданных функции,   
отразятся на объектах, переданных вызывающей программой.  
 •  
Аргументы, возвращаемые значения и переменные не объявляются. Как   
и повсюду в языке Python, на функции также не накладывается никаких   
ограничений по типу. Фактически никакие элементы функций не требуют   
предварительного объявления: вы можете передавать функции аргументы   
любых типов, возвращать из функции объекты любого типа и так далее.   
Как следствие этого одна и та же функция может применяться к объектам   
различных типов – допустимыми считаются любые объекты, поддержи-

Создание функций   
465  
вающие совместимые интерфейсы (методы и выражения), независимо от   
конкретного типа.  
Если что-то из сказанного выше вам показалось непонятным, не волнуйтесь –   
в этой части книги мы исследуем все эти концепции на примерах программно-  
го кода. А теперь начнем изучение некоторых из этих идей и рассмотрим не-  
сколько примеров.  
Инструкция def  
Инструкция def создает объект функции и связывает его с именем. В общем   
виде инструкция имеет следующий формат:  
def <name>(arg1, arg2,... argN):  
 <statements>  
Как и все составные инструкции в языке Python, инструкция def состоит из   
строки заголовка и следующего за ней блока инструкций, обычно с отступа-  
ми (или простая инструкция вслед за двоеточием). Блок инструкций образует   
тело функции, то есть программный код, который выполняется интерпретато-  
ром всякий раз, когда производится вызов функции.  
В строке заголовка инструкции def определяются имя функции, с которым бу-  
дет связан объект функции, и список из нуля или более аргументов (иногда их   
называют параметрами) в круглых скобках. Имена аргументов в строке заго-  
ловка будут связаны с объектами, передаваемыми в функцию, в точке вызова.  
Тело функции часто содержит инструкцию return:  
def <name>(arg1, arg2,... argN):  
 ...  
 return <value>  
Инструкция return может располагаться в любом месте в теле функции – она   
завершает работу функции и передает результат вызывающей программе. Ин-  
струкция return содержит объектное выражение, которое дает результат функ-  
ции. Инструкция return является необязательной – если она отсутствует, рабо-  
та функции завершается, когда поток управления достигает конца тела функ-  
ции. С технической точки зрения, функция без инструкции return автоматиче-  
ски возвращает объект None, однако это значение обычно просто игнорируется.  
Функции могут также содержать инструкции yield, которые используются   
для воспроизведения серии значений с течением времени, однако обсуждение   
этой инструкции мы отложим до главы 20, где обсуждаются расширенные   
темы, касающиеся функций.  
Инструкции def исполняются во время выполнения  
Инструкция def в языке Python – это настоящая исполняемая инструкция:   
когда она исполняется, она создает новый объект функции и присваивает этот   
объект имени. (Не забывайте, все, что имеется в языке Python, относится ко   
времени  выполнения, здесь нет понятия времени компиляции.) Будучи ин-  
струкцией, def может появляться везде, где могут появляться инструкции, –   
даже внутри других инструкций. Например, даже при том, что инструкции   
def обычно исполняются, когда производится импорт вмещающего их модуля,

466   
Глава 16. Основы функций   
допускается вкладывать определения функций внутрь инструкций if, что по-  
зволяет производить выбор между альтернативами:  
if test:  
 def func(): # Определяет функцию таким способом  
 ...  
else:  
 def func(): # Или таким способом  
 ...  
...  
func() # Вызов выбранной версии  
Чтобы понять этот фрагмент программного кода, обратите внимание, что ин-  
струкция def напоминает инструкцию присваивания =: она просто выполняет   
присваивание во время выполнения. В отличие от компилирующих языков,   
таких как C, функции в языке Python не должны быть полностью определены   
к моменту запуска программы. Другими словами, инструкции def не интер-  
претируются, пока они не будут достигнуты и выполнены потоком выполне-  
ния, а программный код внутри инструкции def не выполняется, пока функ-  
ция не будет вызвана позднее.  
Так как определение функции происходит во время выполнения, в именах   
функций нет ничего особенного. Важен только объект, на который ссылается   
имя:  
othername = func # Связывание объекта функции с именем  
othername() # Вызов функции  
В этом фрагменте функция была связана с другим именем и вызвана уже с ис-  
пользованием нового имени. Как и все остальное в языке Python, функции –   
это обычные объекты� они явно записываются в память во время выполнения   
программы. Кроме поддержки возможности вызова, функции позволяют при-  
соединять любые атрибуты, в которых можно сохранять информацию для по-  
следующего использования:  
def func(): ... # Создает объект функции  
func() # Вызывает объект  
func.attr = value # Присоединяет атрибут к объекту  
Первый пример: определения и вызовы  
Кроме таких концепций времени выполнения (которые кажутся наиболее уни-  
кальными для программистов, имеющих опыт работы с традиционными ком-  
пилирующими языками программирования) в использовании функций нет   
ничего сложного. Давайте напишем первый пример, в котором продемонстри-  
руем основные моменты. Как видите, функции имеют две стороны: определе-  
ние (инструкция def, которая создает функцию) и вызов (выражение, которое   
предписывает интерпретатору выполнить тело функции).  
Определение  
Ниже приводится фрагмент сеанса работы в интерактивной оболочке, в кото-  
ром определяется функция с именем times. Эта функция возвращает результат   
обработки двух аргументов:

Первый пример: определения и вызовы   
467  
>>> def times(x, y): # Создать функцию и связать ее с именем  
... return x \* y # Тело, выполняемое при вызове функции  
...  
Когда интерпретатор достигнет инструкции def и выполнит ее, он создаст но-  
вый объект функции, в который упакует программный код функции и свяжет   
объект с именем times. Как правило, такие инструкции размещаются в файлах   
модулей и выполняются во время импортирования, однако такую небольшую   
функцию можно определить и в интерактивной оболочке.  
Вызов  
После выполнения инструкции def появляется возможность вызвать функ-  
цию в программе, добавив круглые скобки после ее имени. В круглых скобках   
можно указать один или более аргументов, значения которых будут присвоены   
именам, указанным в заголовке функции:  
>>> times(2, 4) # Аргументы в круглых скобках  
8  
Данное выражение передает функции times два аргумента. Как уже упомина-  
лось ранее, передача аргументов осуществляется за счет выполнения операции   
присваивания, таким образом, имени x в заголовке функции присваивается   
значение 2, а имени y – значение 4, после чего запускается тело функции. В дан-  
ном случае тело функции составляет единственная инструкция return, которая   
отправляет обратно результат выражения. В данном примере возвращаемый   
объект был выведен интерактивной оболочкой автоматически (как и в боль-  
шинстве языков, 2 \* 4 в языке Python равно 8), однако если бы результат потре-  
бовался позднее, мы могли бы присвоить его переменной. Например:  
>>> x = times(3.14, 4) # Сохранить объект результата  
>>> x  
12.56  
Теперь посмотрим, что произойдет, если функции передать объекты совершен-  
но разных типов:  
>>> times(‘Ni’, 4) # Функции не имеют типа  
‘NiNiNiNi’  
На этот раз функция выполнила нечто совершенно иное (здесь вполне уместна   
была бы ссылка на Монти Пайтона (Monty Python)). На этот раз вместо двух   
чисел в аргументах x и y функции были переданы строка и целое число. Вспом-  
ните, что оператор \* может работать как с числами, так и с последовательно-  
стями� поскольку в языке Python не требуется объявлять типы переменных,   
аргументов или возвращаемых значений, мы можем использовать функцию   
times для умножения чисел и повторения последовательностей.  
Другими словами, смысл функции times и тип возвращаемого значения опреде-  
ляется аргументами, которые ей передаются. Это основная идея языка Python   
(и, возможно, ключ к использованию языка), которую мы рассмотрим в сле-  
дующем разделе.

468   
Глава 16. Основы функций   
Полиморфизм в языке Python  
Как мы только что видели, смысл выражения x \* y в нашей простой функции   
times полностью зависит от типов объектов x и y – одна и та же функция может   
выполнять умножение в одном случае и повторение в другом. В языке Python   
именно объекты определяют синтаксический смысл операции. В действитель-  
ности оператор \* – это всего лишь указание для обрабатываемых объектов.  
Такого рода зависимость от типов известна как полиморфизм – термин, впер-  
вые встретившийся нам в главе 4, который означает, что смысл операции за-  
висит от типов обрабатываемых объектов. Поскольку Python – это язык с дина-  
Python – это язык с дина-  
 – это язык с дина-  
мической типизацией, полиморфизм в нем проявляется повсюду. Фактически   
все операции в языке Python являются полиморфическими: вывод, извлечение   
элемента, оператор \* и многие другие.  
Такое поведение заложено в язык изначально и объясняет в большой степени   
его краткость и гибкость. Например, единственная функция может автомати-  
чески применяться к целой категории типов объектов. Пока объекты поддер-  
живают ожидаемый интерфейс (или протокол), функция сможет обрабатывать   
их. То есть, если объект, передаваемый функции, поддерживает ожидаемые   
методы и операторы выражений, он будет совместим с логикой функции.  
Даже в случае с нашей простой функцией times это означает, что любые два   
объекта, поддерживающие оператор \*, смогут обрабатываться функцией, и не-  
важно, что они из себя представляют и когда были созданы. Эта функция будет   
работать с числами (выполняя операцию умножения) или со строкой и числом   
(выполняя операцию повторения) и с любыми другими комбинациями объек-  
тов, поддерживающими ожидаемый интерфейс, – даже с объектами, порож-  
денными на базе классов, которые мы еще пока не создали.  
Кроме того, если функции будут переданы объекты, которые не поддержива-  
ют ожидаемый интерфейс, интерпретатор обнаружит ошибку при выполнении   
выражения \* и автоматически возбудит исключение. Поэтому для нас совер-  
шенно бессмысленно предусматривать проверку на наличие ошибок в про-  
граммном коде. Фактически добавив такую проверку, мы ограничим область   
применения нашей функции, так как она сможет работать только с теми типа-  
ми объектов, которые мы предусмотрели.  
Это важнейшее отличие философии языка Python от языков программирова-  
ния со статической типизацией, таких как C++ и Java: программный код на   
языке Python не делает предположений о конкретных типах данных. В про-  
тивном случае он сможет работать только с теми типами данных, которые ожи-  
дались на момент его написания, и он не будет поддерживать объекты других   
совместимых типов, которые могут быть созданы в будущем. Проверку типа   
объекта можно выполнить с помощью таких средств, как встроенная функция   
type, но в этом случае программный код потеряет свою гибкость. Вообще гово-  
ря, при программировании на языке Python во внимание принимаются интер-  
фейсы объектов, а не типы данных.  
Конечно, такая модель полиморфизма предполагает необходимость тестиро-  
вания программного кода на наличие ошибок, так как из-за отсутствия объ-  
явлений типов нет возможности с помощью компилятора выявить некоторые   
виды ошибок на ранней стадии. Однако в обмен на незначительное увеличение   
объема отладки мы получаем существенное уменьшение объема программного

Второй пример: пересечение последовательностей   
469  
кода, который требуется написать, и существенное увеличение его гибкости.   
На практике это означает чистую победу.  
Второй пример:   
пересечение последовательностей  
Рассмотрим второй пример функции, которая делает немного больше, чем про-  
стое умножение аргументов, и продолжает иллюстрацию основ функций.  
В главе 13 мы написали цикл for, который выбирал элементы, общие для двух   
строк. Там было замечено, что полезность этого программного кода не так ве-  
лика, как могла бы быть, потому что он может работать только с определенны-  
ми переменными и не может быть использован повторно. Безусловно, можно   
было бы просто скопировать этот блок кода и вставлять его везде, где потребу-  
ется, но такое решение нельзя признать ни удачным, ни универсальным – нам   
по-прежнему придется редактировать каждую копию, изменяя имена последо-  
вательностей� изменение алгоритма также влечет за собой необходимость вно-  
сить изменения в каждую копию.  
Определение  
К настоящему моменту вы уже наверняка поняли, что решение этой дилеммы   
заключается в том, чтобы оформить этот цикл for в виде функции. Такой под-  
ход несет нам следующие преимущества:  
 •  
Оформив программный код в виде функции, появляется возможность ис-  
пользовать его столько раз, сколько потребуется.  
 •  
Так как вызывающая программа может передавать функции произволь-  
ные аргументы, функция сможет использоваться с любыми двумя после-  
довательностями (или итерируемыми объектами) для получения их пере-  
сечения.  
 •  
Когда логика работы оформлена в виде функции, достаточно изменить про-  
граммный код всего в одном месте, чтобы изменить способ получения пере-  
сечения.  
 •  
Поместив функцию в файл модуля, ее можно будет импортировать и ис-  
пользовать в любой программе на вашем компьютере.  
В результате программный код, обернутый в функцию, превращается в уни-  
версальную утилиту нахождения пересечения:  
def intersect(seq1, seq2):  
 res = [] # Изначально пустой результат  
 for x in seq1: # Обход последовательности seq1  
 if x in seq2: # Общий элемент?  
 res.append(x) # Добавить в конец  
 return res  
В том, чтобы преобразовать фрагмент кода из главы 13 в функцию, нет ничего   
сложного, – мы просто вложили оригинальную реализацию в инструкцию def   
и присвоили имена объектам, с которыми она работает. Поскольку эта функ-  
ция возвращает результат, мы также добавили инструкцию return, которая   
возвращает полученный объект результата вызывающей программе.

470   
Глава 16. Основы функций   
Вызов  
Прежде чем функцию можно будет вызвать, ее необходимо создать. Для этого   
нужно выполнить инструкцию def, либо введя ее в интерактивной оболочке,   
либо поместив ее в файл модуля и выполнив операцию импорта. Как только   
инструкция def будет выполнена, можно будет вызывать функцию и передать   
ей два объекта последовательностей в круглых скобках:  
>>> s1 = “SPAM”  
>>> s2 = “SCAM”  
>>> intersect(s1, s2) # Строки  
[‘S’, ‘A’, ‘M’]  
В данном примере мы передали функции две строки и получили список общих   
символов. Алгоритм работы функции можно выразить простой фразой: «Для   
всех элементов первого аргумента, если этот элемент присутствует и во втором   
аргументе, добавить его в конец результата». Этот алгоритм на языке Python   
описывается немного короче, чем на естественном языке, но работает он точно   
так же.  
Справедливости ради следует признать, что наша функция поиска пересече-  
ния работает слишком медленно (она выполняет вложенный цикл)� она нахо-  
дит пересечение, которое не является пересечением в математическом смысле   
(в результате могут иметься повторяющиеся значения)� и к тому же она вообще   
не нужна (как мы увидим дальше, операция пересечения в языке Python под-  
Python под-  
 под-  
держивается множествами). В действительности эту функцию можно заме-  
нить единственным выражением генератора списков, демонстрирующим клас-  
сический пример цикла выборки данных:  
>>> [x for x in s1 if x in s2]  
[‘S’, ‘A’, ‘M’]  
Однако она прекрасно подходит на роль простого примера, демонстрирующего   
основы применения функций, – один фрагмент программного кода может при-  
меняться к целому диапазону типов объектов, о чем подробнее рассказывается   
в следующем разделе.  
Еще о полиморфизме  
Как и любая другая функция в языке Python, функция intersect также явля-  
ется полиморфной. То есть она может обрабатывать объекты произвольных ти-  
пов, при условии, что они поддерживают ожидаемый интерфейс:  
>>> x = intersect([1, 2, 3], (1, 4)) # Смешивание типов  
>>> x # Объект с результатом  
[1]  
На этот раз функции были переданы объекты разных типов – список и кор-  
теж, – и это не помешало ей отыскать общие элементы. Благодаря отсутствию   
необходимости предварительного объявления типов аргументов функция in-  
tersect благополучно будет выполнять итерации по объектам последователь-  
ностей любых типов, если они будут поддерживать ожидаемые интерфейсы.  
Для функции intersect это означает, что первый объект должен обладать под-  
держкой циклов for, а второй – поддержкой оператора in, выполняющего про-

Второй пример: пересечение последовательностей   
471  
верку на вхождение. Любые два объекта, отвечающие этим условиям, будут   
обработаны независимо от их типов, включая как сами последовательности,   
такие как строки и списки, так и любые итерируемые объекты, с которыми мы   
встречались в главе 14, включая файлы, словари и даже объекты, созданные   
на основе классов и использующие перегрузку операторов (эту тему мы будем   
рассматривать в шестой части книги).1   
И снова, если функции передать объекты, которые не поддерживают эти ин-  
терфейсы (например, числа), интерпретатор автоматически обнаружит несоот-  
ветствие и возбудит исключение, то есть именно то, что нам требуется, и это   
лучше, чем добавление явной проверки типов аргументов. Отказываясь от   
реализации проверки типов и позволяя интерпретатору самому обнаруживать   
несоответствия, мы тем самым уменьшаем объем программного кода и повы-  
шаем его гибкость.  
Локальные переменные  
Пожалуй, самое интересное в этом примере заключено в переменных. Пере-  
менная res внутри функции intersect – это то, что в языке Python называется   
локальной  переменной, – имя, которое доступно только программному коду   
внутри инструкции def и существует только во время выполнения функции.   
Фактически любые имена, которым тем или иным способом были присвоены   
некоторые значения внутри функции, по умолчанию классифицируются как   
локальные переменные. Почти все имена в функции intersect являются ло-  
кальными переменными:  
 •  
Переменная res явно участвует в операции присваивания, поэтому она – ло-  
кальная переменная.  
 •  
Аргументы передаются через операцию присваивания, поэтому seq1 и seq2   
тоже локальные переменные.  
 •  
Цикл for присваивает элементы переменной, поэтому имя x также является   
локальным.  
Все эти локальные переменные появляются только в момент вызова функции   
и исчезают, когда функция возвращает управление – инструкция return, стоя-  
щая в конце функции intersect, возвращает объект результата, а имя res ис-  
чезает. Однако, чтобы полностью исследовать понятие локальности, нам необ-  
ходимо перейти к главе 17.  
1   
Эта функция будет работать, если содержимое файлов передается ей в виде результа-  
тов вызова метода file.readlines(). Она может не работать непосредственно с откры-  
тыми файлами, в зависимости от особенностей реализации поддержки оператора in   
в объектах файлов. Вообще, после достижения конца файла необходимо переустано-  
вить текущую позицию в начало этого файла (например, с помощью вызова метода   
file.seek(0)). Как мы увидим в главе 29, когда будем изучать возможность перегруз-  
ки операторов, классы реализуют поддержку оператора in либо с помощью метода   
\_\_contains\_\_, либо за счет реализации общей поддержки протокола итераций с помо-  
щью метода \_\_iter\_\_ или более старого метода \_\_getitem\_\_. С помощью этих методов   
мы можем определить, что означает понятие итерации для наших данных.

472   
Глава 16. Основы функций   
В заключение  
В этой главе были представлены основные идеи, на которых основано определе-  
ние функции, – синтаксис и принцип действия инструкций def и return, выра-  
жений вызова функций, а также суть и преимущества полиморфизма в функ-  
циях языка Python. Как было показано, инструкция def – это исполняемый   
программный код, который создает объект функции. Когда позднее произво-  
дится вызов функции, передача объектов производится за счет выполнения   
операции присваивания (вспомните, в главе 6 говорилось, что присваивание   
в языке Python означает передачу ссылок на объекты, которые фактически ре-  
ализованы в виде указателей), а вычисленные значения возвращаются обратно   
с помощью инструкции return. В этой главе мы также приступили к изучению   
таких понятий, как локальные переменные и области видимости, но более под-  
робно эти темы будут рассматриваться в главе 17. Сначала все-таки ответьте на   
контрольные вопросы.  
Закрепление пройденного  
Контрольные вопросы  
1. Какие преимущества несет использование функций?  
2. В какой момент времени интерпретатор Python создает функции?  
3. Что возвращает функция, если в ней отсутствует инструкция return?  
4. Когда выполняется программный код, вложенный в инструкцию определе-  
ния функции?  
5. Почему нежелательно выполнять проверку типов объектов, передаваемых   
функции?  
Ответы  
1. Функции представляют собой основной способ избежать избыточности   
программного кода – выделение программного кода в виде функции означа-  
ет, что в будущем нам потребуется изменить единственную копию действу-  
ющего кода. Кроме того, функции – это основные блоки программного кода   
многократного пользования – заключение программного кода в функции   
превращает его в инструмент многократного пользования, который может   
вызываться различными программами. Наконец, функции позволяют раз-  
бить сложную систему на небольшие и легко управляемые части, каждая   
из которых может разрабатываться отдельно.  
2. Функция создается, когда интерпретатор достигает инструкции def и вы-  
полняет ее� эта инструкция создает объект функции и связывает его с име-  
нем функции. Обычно это происходит, когда файл модуля, включающего   
функцию, импортируется другим модулем (вспомните, что во время им-  
порта программный код файла выполняется от начала до конца, включая   
и инструкции def), но это может происходить, когда инструкция def вводит-  
ся в интерактивной оболочке или во время выполнения вложенного блока   
другой инструкции, такой как if.  
3. Когда поток управления достигает конца тела функции, не встретив на   
своем пути инструкцию return, функция по умолчанию возвращает объект

Закрепление пройденного   
473  
None. Такие функции обычно вызываются как инструкции выражений, так   
как присваивание объекта None переменной в целом бессмысленно.  
4. Тело функции (программный код, вложенный в инструкцию определения   
функции) выполняется, когда позднее производится вызов функции. Тело   
функции выполняется всякий раз, когда происходит вызов.  
5. Проверка типов объектов, передаваемых функции, – это надежный способ   
ограничить их гибкость, потому что в этом случае функция сможет рабо-  
тать только с объектами определенных типов. Без такой проверки функция   
наверняка сможет обрабатывать целую категорию типов объектов – любые   
объекты, поддерживающие интерфейс, ожидаемый функцией, смогут быть   
обработаны. (Термин интерфейс означает набор методов и операций, кото-  
рые используются функцией.)

Глава 17.  
   
Области видимости  
В главе 16 были описаны основы определения и вызова функций. Как мы ви-  
дели, базовая модель функций в языке Python достаточно проста в исполь-  
зовании. Но даже примеры простых функций быстро вызвали у нас вопросы   
о переменных и их значении в программном коде. В этой главе будут представ-  
лены подробности, лежащие в основе областей видимости, – мест, где опреде-  
ляются переменные и где выполняется их поиск. Как мы увидим далее, место   
в программном коде, где выполняется присваивание переменной, чрезвычай-  
но важно для определения ее назначения. Мы также узнаем, что правильное   
использование областей видимости совершенно необходимо для обеспечения   
нормальной работы программ� злоупотребление глобальными переменными   
считается дурным тоном.  
Области видимости в языке Python  
Теперь, когда вы готовы приступить к созданию своих собственных функций,   
нам необходимо более формально определить, что означают имена в языке   
Python. Всякий раз, когда в программе используется некоторое имя, интер-  
претатор создает, изменяет или отыскивает это имя в пространстве имен –   
в области, где находятся имена. Когда мы говорим о поиске значения имени   
применительно к программному коду, под термином область видимости под-  
разумевается пространство имен: то есть место в программном коде, где имени   
было присвоено значение, определяет область видимости этого имени для про-  
граммного кода.  
Практически все, что имеет отношение к именам, включая классификацию об-  
ластей видимости, в языке Python связано с операциями присваивания. Как   
мы уже видели, имена появляются в тот момент, когда им впервые присваива-  
ются некоторые значения, и прежде чем имена смогут быть использованы, им   
необходимо присвоить значения. Поскольку имена не объявляются заранее,   
интерпретатор Python по местоположению операции присваивания связывает   
имя с конкретным пространством имен. Другими словами, место, где выпол-  
няется присваивание, определяет пространство имен, в котором будет нахо-  
диться имя, а следовательно, и область его видимости.

Области видимости в языке Python   
475  
Помимо упаковки программного кода функции привносят в программы еще   
один слой пространства имен – по умолчанию все имена, значения которым   
присваиваются внутри функции, ассоциируются с пространством имен этой   
функции и никак иначе. Это означает, что:  
 •  
Имена, определяемые внутри инструкции def, видны только программному   
коду внутри инструкции def. К этим именам нельзя обратиться за предела-  
ми функции.  
 •  
Имена, определяемые внутри инструкции def, не вступают в конфликт   
с именами, находящимися за пределами инструкции def, даже если и там   
и там присутствуют одинаковые имена. Имя X, которому присвоено значе-  
ние за пределами данной инструкции def (например, в другой инструкции   
def или на верхнем уровне модуля), полностью отлично от имени X, которо-  
му присвоено значение внутри инструкции def.  
В любом случае область видимости переменной (где она может использовать-  
ся) всегда определяется местом, где ей было присвоено значение, и никакого   
отношения не имеет к месту, откуда была вызвана функция. Как мы узнаем   
далее в этой главе, значения переменным могут быть присвоены в трех разных   
местах, соответствующих трем разным областям видимости:  
 •  
Если присваивание переменной выполняется внутри инструкции def, пере-  
менная является локальной для этой функции.  
 •  
Если присваивание производится в пределах объемлющей инструкции def,   
переменная является нелокальной для этой функции.  
 •  
Если присваивание производится за пределами всех инструкций def, она   
является глобальной для всего файла.   
Мы называем это лексической областью видимости, потому что области види-  
мости переменных целиком определяются местоположением этих переменных   
в исходных текстах программы, а не местом, откуда вызываются функции.  
Например, в следующем файле модуля инструкция присваивания X = 99 соз-  
дает глобальную переменную с именем X (она видима из любого места в файле),   
а инструкция X = 88 создает локальную переменную X (она видима только вну-  
три инструкции def):  
X = 99  
   
def func():  
 X = 88  
Даже при том, что обе переменные имеют имя X, области видимости делают   
их различными. Таким образом, области видимости функций позволяют избе-  
жать конфликтов имен в программах и превращают функции в самостоятель-  
ные элементы программ.  
Правила видимости имен   
До того как мы начали писать функции, весь программный код размещался   
на верхнем уровне модуля (он не был вложен в инструкции def), поэтому все   
имена, которые мы использовали, либо находились на верхнем уровне в мо-  
дуле, либо относились к предопределенным именам языка Python (например,   
open). Функции образуют вложенные пространства имен (области видимости),   
которые ограничивают доступ к используемым в них именам, благодаря чему

476   
Глава 17. Области видимости   
имена внутри функций не вступают в конфликт с именами за их пределами   
(внутри модуля или внутри других функций). Повторю еще раз, функции обра-  
зуют локальную область видимости, а модули – глобальную. Эти две области   
взаимосвязаны между собой следующим образом:  
 •  
Объемлющий модуль – это глобальная область видимости. Каждый мо-  
дуль – это глобальная область видимости, то есть пространство имен, в ко-  
тором создаются переменные на верхнем уровне в файле модуля. Глобаль-  
ные переменные для внешнего мира становятся атрибутами объекта моду-  
ля, но внутри модуля могут использоваться как простые переменные.  
 •  
Глобальная область видимости охватывает единственный файл. Не надо   
заблуждаться насчет слова «глобальный» – имена на верхнем уровне фай-  
ла являются глобальными только для программного кода в этом файле. На   
самом деле в языке Python не существует такого понятия, как всеобъемлю-  
Python не существует такого понятия, как всеобъемлю-  
 не существует такого понятия, как всеобъемлю-  
щая глобальная для всех файлов область видимости. Имена всегда относят-  
ся к какому-нибудь модулю и всегда необходимо явно импортировать мо-  
дуль, чтобы иметь возможность использовать имена, определяемые в нем.   
Когда вы слышите слово «глобальный», подразумевайте «модуль».  
 •  
Каждый вызов функции создает новую локальную область видимости.   
Всякий раз, когда вызывается функция, создается новая локальная область   
видимости – то есть пространство имен, в котором находятся имена, опре-  
деляемые внутри функции. Каждую инструкцию def (и выражение lambda)   
можно представить себе, как определение новой локальной области види-  
мости. Но так как язык Python позволяет функциям вызывать самих себя   
в цикле (этот прием известен как рекурсия), локальная область видимости   
с технической точки зрения соответствует вызову функции – другими сло-  
вами, каждый вызов создает новое локальное пространство имен. Рекур-  
сию удобно использовать, когда выполняется обработка данных, структура   
которых заранее не известна.  
 •  
Операция присваивания создает локальные имена, если они не были объ-  
явлены глобальными или нелокальными. По умолчанию все имена, ко-  
торым присваиваются значения внутри функции, помещаются в локаль-  
ную область видимости (пространство имен, ассоциированное с вызовом   
функции). Если необходимо присвоить значение имени верхнего уровня   
в модуле, который вмещает функцию, это имя необходимо объявить вну-  
три функции глобальным с помощью инструкции global. Если необходимо   
присвоить значение имени, которое находится в объемлющей инструкции   
def, в Python 3.0 это имя необходимо объявить внутри функции с помощью   
инструкции nonlocal.  
 •  
Все остальные имена являются локальными в области видимости объ-  
емлющей функции, глобальными или встроенными. Предполагается, что   
имена, которым не присваивались значения внутри определения функции,   
находятся в объемлющей локальной области видимости (внутри объемлю-  
щей инструкции def), глобальной (в пространстве имен модуля) или встро-  
енной (предопределенные имена в модуле builtins ).  
Здесь следует отметить несколько важных особенностей. Прежде всего, имейте   
в виду, что программный код, который вводится в интерактивной оболочке,   
подчиняется тем же самым правилам. Возможно, вы этого еще не знаете, но   
программный код, который вводится в интерактивной оболочке, в действи-  
тельности находится на уровне модуля \_\_main\_\_ – этот модуль действует точно

Области видимости в языке Python   
477  
так же, как любой другой модуль� единственное отличие состоит лишь в том,   
что результаты вычислений выводятся немедленно. Вследствие этого имена,   
создаваемые в интерактивной оболочке, также находятся внутри модуля и сле-  
дуют обычным правилам видимости: они являются глобальными для интерак-  
тивного сеанса. Подробнее о модулях будет рассказываться в следующей части   
книги.  
Кроме того, обратите внимание, что любые операции присваивания, выполняе-  
мые внутри функции, классифицируют имена как локальные: инструкция =,   
инструкция import, инструкция def, передача аргументов и так далее. Если   
какая-либо из разновидностей операции присваивания выполняется в преде-  
лах инструкции def, имя становится локальным по отношению к этой функции.  
Следует также заметить, что операции непосредственного  изменения  объек-  
тов не рассматривают имена как локальные – это свойственно только операци-  
ям присваивания. Например, если имени L присвоен список, определенный на   
верхнем уровне в модуле, то такая инструкция, как L.append(X), внутри функ-  
ции не будет классифицировать имя L как локальное, тогда как инструкция L   
= X – будет. В первом случае происходит изменение объекта списка, на который   
указывает L, а не самого имени L, – список L будет найден в глобальной области   
видимости, как обычно, и Python изменит этот список, без необходимости объ-  
Python изменит этот список, без необходимости объ-  
 изменит этот список, без необходимости объ-  
явления имени global (или nonlocal). Этот пример должен помочь явственнее   
ощутить различия между именами и объектами: операция, изменяющая объ-  
ект, совсем не то, что операция присваивания объекта имени.  
Разрешение имен: правило LEGB  
Если предыдущий раздел показался вам запутанным, спешу успокоить – в дей-  
ствительности все сводится к трем простым правилам. Для инструкции def:  
 •  
Поиск имен ведется самое большее в четырех областях видимости: локаль-  
ной, затем в объемлющей функции (если таковая имеется), затем в глобаль-  
ной и, наконец, во встроенной.  
 •  
По умолчанию операция присваивания создает локальные имена.  
 •  
Объявления global и nonlocal отображают имена на область видимости вме-  
щающего модуля и функции соответственно.  
Другими словами, все имена, которым присваиваются значения внутри ин-  
струкции def (или внутри выражения lambda, с которым мы познакомимся   
позже), по умолчанию являются локальными� функции могут использовать   
имена в лексически объемлющих функциях и в глобальной области видимо-  
сти, но чтобы иметь возможность изменять их, они должны быть объявлены   
нелокальными и глобальными.  
Схема разрешения имен в языке Python иногда называется правилом LEGB,   
название которого состоит из первых букв названий областей видимости:  
 •  
Когда внутри функции выполняется обращение к неизвестному имени, ин-  
терпретатор пытается отыскать его в четырех областях видимости – в ло-  
кальной (local, L), затем в локальной области любой объемлющей инструк-  
ции def (enclosing, E) или в выражении lambda, затем в глобальной (global,   
G) и, наконец, во встроенной (built-in, B). Поиск завершается, как только   
будет найдено первое подходящее имя. Если требуемое имя не будет най-  
дено, интерпретатор выведет сообщение об ошибке. Как уже говорилось

478   
Глава 17. Области видимости   
в главе 6, прежде чем имя можно будет использовать, ему должно быть при-  
своено значение.  
 •  
Когда внутри функции выполняется операция присваивания (а не обраще-  
ние к имени внутри выражения), интерпретатор всегда создает или изме-  
няет имя в локальной области видимости, если в этой функции оно не было   
объявлено глобальным или нелокальным.  
 •  
Когда выполняется присваивание имени за пределами функции (то есть на   
уровне модуля или в интерактивной оболочке), локальная область видимо-  
сти совпадает с глобальной – с пространством имен модуля.  
На рис. 17.1 показаны четыре области видимости в языке Python. Примеча-  
Python. Примеча-  
. Примеча-  
тельно, что второй области видимости E – в области видимости объемлющей   
инструкции def или выражения lambda, с технической точки зрения может   
находиться несколько вложенных друг в друга областей. Но они появляются,   
только когда имеются вложенные друг в друга функции, и именно к ним от-  
носится объявление nonlocal.1  
Имена, определяемые тем или иным способом внутри функции  
(инструкция def или lambda), которые не были объявлены как глобальные.  
Встроенная область видимости (Python)  
Глобальная область видимости (модуль)  
Предопределенные имена в модуле встроенных имен:  
open, range, SyntaxError…  
Имена, определяемые на верхнем уровне модуля  
или объявленные внутри инструкций def как глобальные.  
Локальные области видимости объемлющих функций  
Имена в локальной области видимости любой и всех объемлющих  
функций (инструкция def или lambda), изнутри наружу.  
Локальная область видимости (функция)  
Рис. 17.1. Правило LEGB поиска имен в областях видимости. Когда произ-  
LEGB поиска имен в областях видимости. Когда произ-  
 поиска имен в областях видимости. Когда произ-  
водится обращение к переменной, интерпретатор Python начинает искать   
ее в следующем порядке: в локальной области видимости, во всех локальных   
областях видимости объемлющих функций, в глобальной области видимо-  
сти и, наконец, во встроенной области видимости. Поиск завершается, как   
только будет найдено первое подходящее имя. Место, где в программном коде   
производится присваивание значения переменной, обычно определяет ее об-  
ласть видимости  
1   
В первом издании этой книги правило поиска в областях видимости было названо   
«правилом LGB». Уровень «E» объемлющей инструкции def был добавлен в язык   
Python позднее, чтобы ликвидировать необходимость явной передачи объемлющей   
области видимости. Но эта тема едва ли представляет интерес для начинающих, по-  
этому мы рассмотрим ее позднее в этой главе. Так как именно к этой области види-  
мости относится объявление nonlocal в Python 3.0, вероятно, правило поиска было   
бы лучше назвать «LNGB», но обратная совместимость в книгах тоже имеет важное   
значение!

Области видимости в языке Python   
479  
Кроме того, имейте в виду, что эти правила применяются только к простым   
именам переменных (таким как spam). В пятой и шестой частях книги мы уви-  
дим, что полные имена атрибутов (такие как object.spam) принадлежат опре-  
деленным объектам и к ним применяются иные правила поиска, отличные от   
правил поиска в областях видимости, которые мы только что рассмотрели.   
При обращении к атрибутам (имя, следующее за точкой) поиск производится   
в одном или более объектах, а не в областях видимости, что связано с механиз-  
мом, который называется «наследованием» (рассматривается в шестой части   
книги).  
Пример области видимости  
Рассмотрим более крупный пример, демонстрирующий суть областей видимо-  
сти. Предположим, что следующий фрагмент составляет содержимое файла   
модуля:  
# Глобальная область видимости  
X = 99 # X и func определены в модуле: глобальная область  
   
def func(Y): # Y и Z определены в функции: локальная область  
 # Локальная область видимости  
 Z = X + Y # X – глобальная переменная  
 return Z  
   
func(1) # func в модуле: вернет число 100  
В этом примере функция и модуль используют в своей работе несколько имен.   
Применяя правила области видимости языка Python, можно классифициро-  
вать эти имена следующим образом:  
Глобальные имена: X и func  
X – это глобальное имя, так как оно объявлено на верхнем уровне модуля.   
К этому имени можно обращаться внутри функции, не объявляя его гло-  
бальным. func – это глобальное имя по тем же причинам. Инструкция def   
связывает объект функции с именем func на верхнем уровне модуля.  
Локальные имена: Y и Z  
Имена Y и Z являются локальными (и существуют только во время выполне-  
ния функции), потому что присваивание значений обоим именам осущест-  
вляется внутри определения функции: присваивание переменной Z произ-  
водится с помощью инструкции =, а Y – потому что аргументы всегда пере-  
даются через операцию присваивания.  
Суть такого разделения имен заключается в том, что локальные переменные   
играют роль временных имен, которые необходимы только на время испол-  
нения функции. Например, в предыдущем примере аргумент Y и результат   
сложения Z существуют только внутри функции – эти имена не пересекаются   
с вмещающим пространством имен модуля (или с пространствами имен любых   
других функций).  
Разделение имен на глобальные и локальные также облегчает понимание   
функций, так как большинство имен, используемых в функции, появляются   
непосредственно в самой функции, а не в каком-то другом, произвольном месте   
внутри модуля. Кроме того, можно быть уверенным, что локальные имена не   
будут изменены любой другой удаленной функцией в программе, а это в свою   
очередь упрощает отладку программ.

480   
Глава 17. Области видимости   
Встроенная область видимости  
Встроенная область видимости, уже упоминавшаяся выше, немного проще,   
чем можно было бы подумать. В действительности, встроенная область види-  
мости – это всего лишь встроенный модуль с именем builtis, но для того, чтобы   
использовать имя builtis, необходимо импортировать модуль builtis, потому   
что это имя само по себе не является встроенным.  
Я вполне серьезен! Встроенная область видимости реализована как модуль   
стандартной библиотеки с именем builtis, но само имя не находится во встро-  
енной области видимости, поэтому, чтобы исследовать его, необходимо импор-  
тировать модуль. После этого можно будет воспользоваться функцией dir, что-  
бы получить список предопределенных имен:  
>>> import builtins  
>>> dir(builtins)  
[‘ArithmeticError’, ‘AssertionError’, ‘AttributeError’, ‘BaseException’,  
‘BufferError’, ‘BytesWarning’, ‘DeprecationWarning’, ‘EOFError’, ‘Ellipsis’,  
 ...множество других имен опущено...  
‘print’, ‘property’, ‘quit’, ‘range’, ‘repr’, ‘reversed’, ‘round’, ‘set’,  
‘setattr’, ‘slice’, ‘sorted’, ‘staticmethod’, ‘str’, ‘sum’, ‘super’, ‘tuple’,  
‘type’, ‘vars’, ‘zip’]   
Имена в этом списке составляют встроенную область видимости языка Python.   
Примерно первая половина списка – это встроенные исключения, а вторая –   
встроенные функции. Согласно правилу LEGB интерпретатор выполняет по-  
иск имен в этом модуле в последнюю очередь. Все имена из этого списка вы   
получаете в свое распоряжение по умолчанию, то есть чтобы их использовать,   
не требуется импортировать какие-либо модули. Благодаря этому существует   
два способа вызвать встроенную функцию – используя правило LEGB или им-  
портируя модуль builtins вручную:  
>>> zip # Обычный способ  
<class zip>  
>>> import builtins # Более сложный способ  
>>> builtins.zip  
<class zip>  
Второй способ иногда удобно использовать при выполнении сложных дей-  
ствий. Внимательный читатель может заметить, что согласно правилу LEGB   
поиск имени прекращается, когда будет найдено первое подходящее имя, от-  
куда следует, что имена в локальной области видимости могут переопределять   
переменные с теми же самыми именами, как в глобальной, так и во встроенной   
области видимости, а глобальные имена могут переопределять имена во встро-  
енной области видимости. Например, внутри функции можно создать перемен-  
ную с именем open:  
def hider():  
 open = ‘spam’ # Локальная переменная, переопределяет встроенное имя  
 ...  
 open(‘data.txt’) # В этой области видимости файл не будет открыт!  
Однако в результате этого встроенная функция с именем open, которая распола-  
гается во встроенной (внешней) области видимости, окажется скрытой. Обыч-  
но это считается ошибкой, и самое неприятное, что интерпретатор Python не   
выведет сообщения с предупреждением (иногда в программировании возника-

Области видимости в языке Python   
481  
ют ситуации, когда действительно бывает необходимо подменить встроенные   
имена, переопределив их в своем коде).  
Таким же способом функции могут переопределять имена глобальных пере-  
менных, определяя локальные переменные с теми же именами:  
X = 88 # Глобальная переменная X  
   
def func():  
 X = 99 # Локальная переменная X: переопределяет глобальную  
   
func()  
print(X) # Выведет 88: значение не изменилось  
В этом примере операция присваивания создает локальную переменную X,   
которая совершенно отлична от глобальной переменной X, определенной в мо-  
дуле, за пределами функции. Вследствие этого внутри функции нет никакой   
возможности изменить переменную, расположенную за пределами функции,   
если не добавить объявление global (или nonlocal) в инструкцию def (как описа-  
но в следующем разделе).  
Примечание,  касающееся  различий  между  версиями: Умолчать   
об этом нельзя. Модуль builtins, упоминающийся здесь, в версии   
Python 2.6 называется \_\_builtin\_\_. Замечу, что в большинстве   
глобальных областей видимости (включая и интерактивный се-  
анс) присутствует имя \_\_builtins\_\_ (с символом «s» на конце),   
ссылающееся на модуль builtins (он же \_\_builtin\_\_ в 2.6).  
Если в версии 3.0 импортировать модуль builtins, выражение   
\_\_builtins\_\_ is builtins вернет True, а в версии 2.6 значение True   
вернет выражение \_\_builtins\_\_ is \_\_builtin\_\_. Отсюда следует,   
что мы можем исследовать встроенную область видимости с по-  
мощью вызова функции dir(\_\_builtins\_\_) в обеих версиях, 3.0   
и 2.6, не импортируя модуль. Однако в действующих програм-  
мах для версии 3.0 рекомендуется все-таки использовать модуль   
builtins. Кто сказал, что описать эту особенность было легко?  
Как испортить себе жизнь в Python 2.6  
Вот вам еще один пример того, что допустимо в языке Python, но чего   
не следует делать: в версии 2.6 имена True и False – это всего лишь пере-  
менные во встроенной области видимости. Их вполне возможно пере-  
определить с помощью такой инструкции: True = False. При этом вы не   
нарушите общую логическую целостность! Эта инструкция всего лишь   
переопределит значение слова True в единственной области видимости.   
Во всех остальных областях видимости по-прежнему будет использо-  
ваться оригинал из встроенной области видимости.  
Что еще интереснее, можно выполнить даже такую инструкцию: \_\_buil-  
tin\_\_.True = False, и тогда истина станет ложью для всей программы! По-  
добная возможность в версии 3.0 была ликвидирована, там слова True   
и False считаются зарезервированными, как и слово None. Кстати, эта   
операция отправляет среду IDLE в странное состояние, когда пользова-  
тельский процесс сбрасывается.

482   
Глава 17. Области видимости   
Однако такой прием удобен для создателей инструментальных средств,   
которые вынуждены переопределять встроенные имена, такие как open,   
для нужд специализированных функций. Кроме того, следует отметить,   
что инструменты сторонних производителей, такие как PyChecker, вы-  
водят предупреждения о типичных ошибках программирования, вклю-  
чая случайное переопределение встроенных имен (эта возможность,   
встроенная в PyChecker, известна как «shadowing» (сокрытие)).  
Инструкция global  
Инструкция global и родственная ей инструкция nonlocal – единственные ин-  
струкции в языке Python, отдаленно напоминающие инструкции объявления.   
Однако они не объявляют тип или размер – они объявляют  пространства   
имен. Инструкция global сообщает интерпретатору, что функция будет изме-  
нять одно или более глобальных имен, то есть имен, которые находятся в об-  
ласти видимости (в пространстве имен) вмещающего модуля.   
Инструкция global уже упоминалась выше, а ниже приводится общая инфор-  
мация о ней:  
 •  
Глобальные имена – это имена, которые определены на верхнем уровне вме-  
щающего модуля.  
 •  
Глобальные имена должны объявляться, только если им будут присваи-  
ваться значения внутри функций.  
 •  
Обращаться к глобальным именам внутри функций можно и без объявле-  
ния их глобальными.  
Другими словами, инструкция global позволяет изменять переменные, находя-  
щиеся на верхнем уровне модуля, за пределами инструкции def. Как вы узнае-  
те ниже, инструкция nonlocal практически идентична, но она применяется не   
к именам на верхнем уровне модуля, а к именам, находящимся в локальных   
областях видимости объемлющих инструкций def.  
Инструкция global состоит из ключевого слова global и следующих за ним   
одного или более имен, разделенных запятыми, которые будут отображены на   
область видимости вмещающего модуля при обращении к ним или при выпол-  
нении операции присваивания внутри тела функции. Например:  
X = 88 # Глобальная переменная X  
   
def func():  
 global X  
 X = 99 # Глобальная переменная X: за пределами инструкции def  
   
func()  
print(X) # Выведет 99  
В этом примере было добавлено объявление global, поэтому имя X внутри ин-  
струкции def теперь ссылается на переменную X за ее пределами. На этот раз   
оба имени представляют одну и ту же переменную. Ниже приводится более   
сложный пример использования инструкции global:

Инструкция global   
483  
y, z = 1, 2 # Глобальные переменные в модуле  
def all\_global():  
 global x # Объявляется глобальной для присваивания  
 x = y + z # Объявлять y, z не требуется: применяется правило LEGB  
Здесь все три переменные x, y и z, используемые внутри функции all\_global,   
являются глобальными. Переменные y и z глобальными считаются потому, что   
внутри функции им не присваиваются значения. Переменная x считается гло-  
бальной потому, что она перечислена в инструкции global, которая явно ото-  
бражает ее на область видимости модуля. Без инструкции global переменная x   
считалась бы локальной, так как ей присваивается значение внутри функции.  
Обратите внимание: переменные y и z не были объявлены как глобальные, од-  
нако, следуя правилу LEGB, интерпретатор автоматически отыщет их в обла-  
сти видимости модуля. Кроме того, следует отметить, что переменная x может   
не существовать в модуле на момент вызова функции – в этом случае операция   
присваивания в функции создаст переменную x в области видимости модуля.  
Минимизируйте количество глобальных переменных  
По умолчанию имена, значения которым присваиваются внутри функций, яв-  
ляются локальными, поэтому, если необходимо изменять имена за пределами   
функций, следует использовать инструкцию global. Это сделано в соответствии   
с общей идеологией языка Python – чтобы сделать что-то «неправильное», не-  
обходимо писать дополнительный программный код. Иногда бывает удобно ис-  
пользовать глобальные переменные, однако по умолчанию, если переменной   
присваивается значение внутри инструкции def, она становится локальной, по-  
тому что это, как правило, наилучшее решение. Изменение глобальных пере-  
менных может привести к появлению проблем, хорошо известных в разработ-  
ке программного обеспечения: когда значения переменных зависят от поряд-  
ка, в каком вызываются функции, это может осложнить отладку программы.  
Рассмотрим следующий пример модуля:  
X = 99  
   
def func1():  
 global X  
 X = 88  
   
def func2():  
 global X  
 X = 77  
Теперь представим, что перед нами стоит задача модифицировать этот модуль   
или использовать его в другой программе. Каким будет значение переменной X?   
На самом деле этот вопрос не имеет смысла, если не указывать момент време-  
ни – значение переменной X зависит от выбранного момента времени, так как   
оно зависит от того, какая функция вызывалась последней (этого нельзя ска-  
зать только по одному файлу модуля).  
В результате, чтобы понять этот программный код, необходимо знать путь   
потока выполнения всей программы. И если возникнет необходимость изме-  
нить этот модуль или использовать его в другой программе, необходимо будет   
удерживать в своей памяти всю программу. В этой ситуации невозможно ис-

484   
Глава 17. Области видимости   
пользовать одну функцию, не принимая во внимание другую. От них зависит   
значение глобальной переменной. Это типичная проблема глобальных пере-  
менных – они вообще делают программный код более сложным для понима-  
ния и использования, в отличие от кода, состоящего только из независимых   
функций, логика выполнения которых построена на использовании локаль-  
ных имен.  
С другой стороны, за исключением случаев использования классов и принци-  
пов объектно-ориентированного программирования, глобальные переменные   
являются едва ли не самым удобным способом хранения информации о состоя-  
нии (информации, которую необходимо хранить между вызовами функции) –   
локальные переменные исчезают, когда функция возвращает управление,   
а глобальные – нет. Это можно реализовать с помощью других приемов, таких   
как использование изменяемых аргументов по умолчанию и области видимо-  
сти объемлющих функций, но они слишком сложны по сравнению с глобаль-  
ными переменными.  
Некоторые программы определяют отдельный глобальный модуль для хране-  
ния всех глобальных имен – если это предусмотрено заранее, это не так вред-  
но. Кроме того, программы на языке Python, использующие многопоточную   
модель выполнения для параллельной обработки данных, тесно связаны с гло-  
бальными переменными – они играют роль памяти, совместно используемой   
функциями, исполняющимися в параллельных потоках, и выступают в каче-  
стве средств связи.1  
А пока, особенно если вы не имеете достаточного опыта программирования, по   
мере возможности избегайте искушения использовать глобальные переменные   
(старайтесь организовать обмен данными через параметры и возвращаемые   
значения). Шесть месяцев спустя вы и ваши коллеги будете рады, что посту-  
пали таким образом.  
Минимизируйте количество изменений   
в соседних файлах  
В этом разделе описывается еще одна проблема, связанная с областями види-  
мости: несмотря на то, что существует возможность непосредственно изме-  
нять переменные в другом файле, этого следует избегать. Файлы модулей были   
представлены в главе 3 и более подробно будут рассматриваться в следующей   
части книги. Рассмотрим следующие два модуля:  
1   
Многопоточный режим позволяет запускать функции, которые выполняются па-  
раллельно с основной программой, и поддерживается модулями \_thread, threading   
и queue (thread, threading и Queue в Python 2.6), входящими в состав стандартной би-  
блиотеки. Так как все потоки управления выполняются в рамках одного и того же   
процесса, глобальная область видимости зачастую может служить аналогом области   
памяти, совместно используемой всеми потоками. Многопоточный режим обычно   
используется в программах с графическим интерфейсом пользователя, для реализа-  
ции неблокирующих операций и более оптимального использования вычислитель-  
ной мощности процессора. Однако описание многопоточной модели выполнения   
выходит далеко за рамки данной книги, поэтому за дополнительной информацией   
обращайтесь к книгам, упомянутым в предисловии (таким как «Программирование   
на Python»), и к руководству по стандартной библиотеке.

Инструкция global   
485  
# first.py  
X = 99 # Это программный код не знает о существовании second.py  
   
# second.py  
import first  
print(first.X) # Нет ничего плохого в том, чтобы обратиться к имени   
 # в другом файле  
first.X = 88 # Но изменение может привести к сложностям  
Первый модуль определяет переменную X, а второй – выводит ее и затем изме-  
няет значение в инструкции присваивания. Обратите внимание, что для это-  
го во втором модуле необходимо импортировать первый модуль. Как вы уже   
знаете, каждый модуль представляет собой отдельное пространство имен (где   
размещаются переменные), поэтому, чтобы увидеть содержимое одного модуля   
во втором, его необходимо импортировать. Это главная особенность модулей:   
разделение пространств имен файлов позволяет избежать конфликтов имен.  
В терминах этой главы глобальная область видимости модуля после импорти-  
рования превращается в пространство имен атрибутов объекта модуля – им-  
портирующий модуль автоматически получает доступ ко всем глобальным пе-  
ременным импортируемого модуля, поэтому при импортировании глобальная   
область видимости импортируемого модуля, по сути, трансформируется в про-  
странство имен атрибутов.  
После импортирования первого модуля второй модуль выводит значение его   
переменной и затем присваивает ей новое значение. Нет ничего плохого в том,   
чтобы в одном модуле сослаться на переменную в другом модуле и вывести ее, –   
как правило, именно таким способом обеспечивается связь между модулями   
в крупных программах. Проблема состоит в том, что эта операция выполняет-  
ся слишком неявно: для любого, кто занимается сопровождением или исполь-  
зует первый модуль, будет сложно догадаться, что какой-то другой модуль, да-  
леко отстоящий в цепочке импорта, может изменить значение переменной X.   
В конце концов, второй модуль может находиться вообще в другом каталоге,   
из-за чего его сложно будет найти.   
Хотя возможность изменения переменных в другом модуле всегда поддержи-  
валась в языке Python, следует помнить, что эти изменения могут повлечь за   
собой трудноуловимые ошибки. Эта возможность порождает слишком тесную   
зависимость между двумя модулями – так как оба они зависят от значения   
переменной X, будет трудно понять или повторно использовать один модуль без   
другого. Такая неочевидная зависимость между модулями в лучшем случае   
приводит к снижению гибкости программы, а в худшем случае – к ошибкам.  
Лучшая рекомендация в подобной ситуации – не использовать такую возмож-  
ность� лучше организовать взаимодействие между модулями через вызовы   
функций, передавая им аргументы и получая возвращаемые значения. В дан-  
ном конкретном случае было бы лучше добавить функцию доступа, которая   
будет выполнять изменения:  
# first.py  
X = 99  
   
def setX(new):  
 global X  
 X = new

486   
Глава 17. Области видимости   
# second.py  
import first  
first.setX(88)  
Для этого потребуется добавить дополнительный программный код, но он име-  
ет огромное значение в смысле обеспечения удобочитаемости и удобства в со-  
провождении – когда тот, кто впервые будет знакомиться с модулем, увидит   
функцию, он будет знать, что это – часть интерфейса модуля, и поймет, что   
переменная X может изменяться. Другими словами, эта функция устраняет   
элемент неожиданности, который вряд ли может считаться положительной   
характеристикой программного продукта. Мы не можем полностью избавить-  
ся от изменений в соседних файлах, однако здравый смысл диктует необходи-  
мость минимизировать их число, если это не является широко распространен-  
ным явлением в программе.  
Другие способы доступа к глобальным переменным  
Интересно, что благодаря трансформации глобальных переменных в атрибу-  
ты объекта загруженного модуля существует возможность имитировать ин-  
струкцию global, импортируя вмещающий модуль и выполняя присваивание   
его атрибутам, как показано в следующем примере модуля. Программный код   
в этом файле в одном случае импортирует вмещающий модуль по имени, а в   
другом использует таблицу загруженных модулей sys.modules (подробнее об   
этой таблице рассказывается в главе 21):  
# thismod.py  
   
var = 99 # Глобальная переменная == атрибут модуля  
   
def local():  
 var = 0 # Изменяется локальная переменная  
   
def glob1():  
 global var # Глобальное объявление (обычное)  
 var += 1 # Изменяется глобальная переменная  
   
def glob2():  
 var = 0 # Изменяется локальная переменная  
 import thismod # Импорт самого себя  
 glob.var += 1 # Изменяется глобальная переменная  
   
def glob3():  
 var = 0 # Изменяется локальная переменная  
 import sys # Импорт системной таблицы  
 glob = sys.modules[‘thismod’] # Получить объект модуля   
 # (или использовать \_\_name\_\_)  
 glob.var += 1 # Изменяется глобальная переменная  
   
def test():  
 print(var)  
 local(); glob1(); glob2(); glob3()  
 print(var)  
После запуска будут добавлены 3 глобальные переменные (только первая функ-  
ция ничего не добавляет):

Области видимости и вложенные функции   
487  
>>> import thismod  
>>> thismod.test()  
99  
102  
>>> thismod.var  
102  
Этот пример иллюстрирует эквивалентность глобальных имен и атрибутов мо-  
дуля, однако чтобы явно выразить свои намерения, нам потребовалось напи-  
сать немного больше, чем при использовании инструкции global.  
Как видите, инструкция global обеспечивает возможность изменения пере-  
менных в модуле из функций. Существует также родственная ей инструкция   
nonlocal, которая обеспечивает возможность изменения переменных в объем-  
лющих функциях, но чтобы понять, где может пригодиться эта инструкция,   
нам сначала нужно исследовать возможность вложения функций друг в друга.  
Области видимости и вложенные функции  
Мы до сих пор не рассмотрели еще одну часть правила области видимости   
в языке Python (просто потому, что с нею редко сталкиваются на практике).   
Однако пришло время более пристально посмотреть на E в правиле LEGB. Уро-  
вень E появился относительно недавно (он был добавлен в Python 2.2) – это   
локальные области видимости объемлющих инструкций def. Иногда объемлю-  
щие области видимости называют статически вложенными областями види-  
мости. В действительности вложение является лексическим – вложенные об-  
ласти видимости соответствуют физически вложенным блокам программного   
кода в исходных текстах программы.  
Вложенные области видимости  
С появлением областей видимости вложенных функций правила поиска пере-  
менных стали немного более сложными. Внутри функции:  
 •  
При обращении к переменной (X) поиск имени X сначала производится в ло-  
кальной области видимости (функции)� затем в локальных областях види-  
мости всех лексически объемлющих функций, изнутри наружу� затем в те-  
кущей глобальной области видимости (в модуле)� и, наконец, во встроенной   
области видимости (модуль builtins). Поиск имен, объявленных в инструк-  
ции global, начинается сразу с глобальной (в модуле) области видимости.  
 •  
Операция присваивания (X = value) по умолчанию создает или изменяет имя   
X в текущей локальной области видимости. Если имя X объявлено глобаль-  
ным внутри функции, операция присваивания создает или изменяет имя X   
в области видимости объемлющего модуля. Если имя X объявлено нелокаль-  
ным внутри функции, операция присваивания создает или изменяет имя X   
в ближайшей области видимости объемлющей функции.  
Обратите внимание, что инструкция global отображает имена в область ви-  
димости объемлющего модуля. Когда имеются вложенные функции, можно   
получить значения переменных в объемлющих функциях, но чтобы их изме-  
нить, переменные должны быть указаны в объявлении nonlocal.

488   
Глава 17. Области видимости   
Примеры вложенных областей видимости  
Чтобы пояснить положения, описанные в предыдущем разделе, рассмотрим их   
на примере программного кода. Ниже приводится пример вложенной области   
видимости:  
X = 99 # Имя в глобальной области видимости: не используется  
   
def f1():  
 X = 88 # Локальное имя в объемлющей функции  
 def f2():  
 print(X) # Обращение к переменной во вложенной функции  
 f2()  
   
f1() # Выведет 88: локальная переменная в объемлющей функции  
Прежде всего – это вполне допустимый программный код на языке Python:   
инструкция def – это обычная исполняемая инструкция, которая может появ-  
ляться в любом месте программы, где могут появляться другие инструкции,   
включая вложение в другую инструкцию def. В этом примере вложенная ин-  
струкция def исполняется в момент вызова функции f1 – она создает функцию   
и связывает ее с именем f2, которое является локальным и размещается в ло-  
кальной области видимости функции f1. В некотором смысле f2 – это времен-  
ная функция, которая существует только во время работы (и видима только   
для программного кода) объемлющей функции f1.  
Однако обратите внимание, что происходит внутри функции f2: когда произ-  
водится вывод переменной X, она ссылается на переменную X в локальной об-  
ласти видимости объемлющей функции f1. Функции имеют возможность об-  
ращаться к именам, которые физически располагаются в любых объемлющих   
инструкциях def, и имя X в функции f2 автоматически отображается на имя X   
в функции f1 в соответствии с правилом поиска LEGB.  
Это правило поиска в объемлющих областях видимости выполняется, даже   
если объемлющая функция фактически уже вернула управление. Например,   
следующий фрагмент определяет функцию, которая создает и возвращает дру-  
гую функцию:  
def f1():  
 X = 88  
 def f2():  
 print(X) # Сохраняет значение X в объемлющей области видимости  
 return f2 # Возвращает f2, но не вызывает ее  
   
action = f1() # Создает и возвращает функцию  
action() # Вызов этой функции: выведет 88  
В этом фрагменте при вызове action фактически запускается функция, создан-  
ная во время выполнения функции f1. Функция f2 помнит переменную X в об-  
ласти видимости объемлющей функции f1, которая уже неактивна.  
Фабричные функции  
В зависимости от того, кому задается вопрос о том, как называется такое пове-  
дение, можно услышать такие термины, как замыкание или фабричная функ-  
ция. Под этими терминами подразумевается объект функции, который сохра-  
няет значения в объемлющих областях видимости, даже когда эти области   
могут прекратить свое существование. Классы (описываются в шестой части

Области видимости и вложенные функции   
489  
книги) обычно лучше подходят для сохранения состояния, потому что они по-  
зволяют делать это явно, посредством присваивания значений атрибутам, тем   
не менее подобные функции обеспечивают другую альтернативу.  
Например, фабричные функции иногда используются в программах, когда не-  
обходимо создавать обработчики событий прямо в процессе выполнения, в со-  
ответствии со сложившимися условиями (например, когда желательно запре-  
тить пользователю вводить данные). Рассмотрим в качестве примера следую-  
щую функцию:  
>>> def maker(N):  
... def action(X): # Создать и вернуть функцию  
... return X \*\* N # Функция action запоминает значение N в объемлющей  
... return action # области видимости  
...  
Здесь определяется внешняя функция, которая просто создает и возвращает   
вложенную функцию, не вызывая ее. Если вызвать внешнюю функцию:  
>>> f = maker(2) # Запишет 2 в N  
>>> f  
<function action at 0x014720B0>  
она вернет ссылку на созданную ею вложенную функцию, созданную при вы-  
полнении вложенной инструкции def. Если теперь вызвать то, что было полу-  
чено от внешней функции:  
>>> f(3) # Запишет 3 в X, в N по-прежнему хранится число 2  
9  
>>> f(4) # 4 \*\* 2  
16  
будет вызвана вложенная функция, с именем action внутри функции maker. Са-  
мое необычное здесь то, что вложенная функция продолжает хранить число   
2, значение переменной N в функции maker даже при том, что к моменту вызова   
функции action функция maker уже завершила свою работу и вернула управле-  
ние. В действительности имя N из объемлющей локальной области видимости   
сохраняется как информация о состоянии, присоединенная к функции action,   
и мы получаем обратно значение аргумента, возведенное в квадрат.  
Теперь, если снова вызвать внешнюю функцию, мы получим новую вложенную   
функцию уже с другой информацией о состоянии, присоединенной к ней, –   
в результате вместо квадрата будет вычисляться куб аргумента, но ранее со-  
храненная функция по-прежнему будет возвращать квадрат аргумента:  
>>> g = maker(3) # Функция g хранит число 3, а f – число 2  
>>> g(3) # 3 \*\* 3  
27  
>>> f(3) # 3 \*\* 2  
9  
Такое возможно благодаря тому, что при каждом обращении к фабричной   
функции, как в данном примере, произведенные ею функции сохраняют свой   
собственный блок данных с информацией о состоянии. В нашем случае благо-  
даря тому, что каждая из функций получает свой собственный блок данных   
с информацией о состоянии, функция, которая присваивается имени g, запо-  
минает число 3 в переменной N функции maker, а функция f – число 2.

490   
Глава 17. Области видимости   
Это довольно сложный прием, который вам вряд ли часто придется часто   
встречать на практике, впрочем, он распространен среди программистов, об-  
ладающих опытом работы с функциональными языками программирования.   
С другой стороны, с объемлющими областями видимости часто можно встре-  
титься в выражениях lambda (рассматриваются ниже в этой главе), потому что   
они практически всегда используются внутри функций. Кроме того, прием   
вложения функций обычно используется при разработке декораторов (рас-  
сматриваются в главе 38)� в некоторых ситуациях это оказывается наиболее   
эффективным приемом.  
Вообще классы, которые будут обсуждаться позднее, лучше подходят на роль   
«памяти», как в данном случае, потому что они обеспечивают явное сохране-  
ние информации. Помимо классов, основными средствами хранения информа-  
ции о состоянии функций в языке Python являются глобальные переменные,   
объемлющие области видимости, как в данном случае, и аргументы по умол-  
чанию. Полное описание аргументов со значениями по умолчанию приводится   
в главе 18, но для того, чтобы начать их использовать, достаточный объем ввод-  
ной информации приводится уже в следующем разделе.  
Сохранение состояния объемлющей области видимости   
с помощью аргументов по умолчанию  
В первых версиях Python такой программный код, как в предыдущем разделе,   
терпел неудачу из-за отсутствия вложенных областей видимости в инструкци-  
ях def – при обращении к переменной внутри функции f2 поиск производился   
сначала в локальной области видимости (f2), затем в глобальной (программ-  
ный код за пределами f1) и затем во встроенной области видимости. Области   
видимости объемлющих функций не просматривались, что могло приводить   
к ошибке. Чтобы разрешить ситуацию, программисты обычно использовали   
аргументы со значениями по умолчанию для передачи (сохранения) объектов,   
расположенных в объемлющей области видимости:  
def f1():  
 x = 88  
 def f2(x=x): # Сохраняет значение переменной x в объемлющей области   
 print(x) # в виде аргумента  
 f2()  
   
f1() # Выведет 88  
Этот фрагмент будет работать во всех версиях Python, и такой подход по-  
прежнему можно встретить в существующих программах. В двух словах за-  
мечу, что конструкция arg = val в заголовке инструкции def означает, что аргу-  
мент arg по умолчанию будет иметь значение val, если функции не передается   
какого-либо другого значения.  
В измененной версии f2 запись x=x означает, что аргумент x по умолчанию бу-  
дет иметь значение переменной x объемлющей области видимости. Поскольку   
значение для второго имени x вычисляется еще до того, как интерпретатор   
Python войдет во вложенную инструкцию def, оно все еще ссылается на имя x   
в функции f1. В результате в значении по умолчанию запоминается значение   
переменной x в функции f1 (то есть объект 88).  
Все это довольно сложно и полностью зависит от того, когда вычисляется зна-  
чение по умолчанию. Фактически поиск во вложенных областях видимости

Области видимости и вложенные функции   
491  
был добавлен в Python, чтобы избавиться от такого способа использования   
значений по умолчанию, – сейчас Python автоматически сохраняет любые зна-  
чения в объемлющей области видимости для последующего использования во   
вложенных инструкциях def.  
Безусловно, наилучшей рекомендацией будет просто избегать вложения ин-  
струкций def в другие инструкции def, так как это существенно упростит про-  
граммы. Ниже приводится фрагмент, который является эквивалентом пред-  
шествующего примера, в котором просто отсутствует понятие вложенности.   
Обратите внимание, что вполне допустимо вызывать функцию, определение   
которой в тексте программы находится ниже функции, откуда производится   
вызов, как в данном случае, при условии, что вторая инструкция def будет ис-  
полнена до того, как первая функция попытается вызвать ее, – программный   
код внутри инструкции def не выполняется, пока не будет произведен факти-  
ческий вызов функции:  
>>> def f1():  
... x = 88 # Передача значения x вместо вложения функций   
... f2(x) # Опережающие ссылки считаются допустимыми  
...  
>>> def f2(x):  
... print(x)  
...  
>>> f1()  
88  
При использовании такого способа можно забыть о концепции вложенных об-  
ластей видимости в языке Python, если вам не потребуется создавать фабрич-  
ные функции, обсуждавшиеся выше, – по крайней мере, при использовании   
инструкций def. Выражения lambda, которые практически всегда вкладыва-  
ются в инструкции def, часто используют вложенные области видимости, как   
описывается в следующем разделе.  
Вложенные области видимости и lambda-выражения  
Несмотря на то что на практике вложенные инструкции def используются до-  
статочно редко, тем не менее вам наверняка придется столкнуться с областями   
видимости вложенных функций, когда вы начнете использовать выражения   
lambda. Мы не будем подробно рассматривать эти выражения до главы 19, но   
в двух словах замечу, что это выражение генерирует новую функцию, которая   
будет вызываться позднее, и оно очень похоже на инструкцию def. Поскольку   
lambda – это выражение, оно может использоваться там, где не допускается ис-  
пользование инструкции def, например в литералах списков и словарей.  
Подобно инструкции def, выражение lambda сопровождается появлением новой   
локальной области видимости. Благодаря наличию возможности поиска имен   
в объемлющей области видимости выражения lambda способны обращаться ко   
всем переменным, которые присутствуют в функциях, где находятся эти вы-  
ражения. Таким образом, следующий программный код будет работать исклю-  
чительно благодаря тому, что в настоящее время действуют правила поиска во   
вложенных областях видимости:  
def func():  
 x = 4  
 action = (lambda n: x \*\* n) # запоминается x из объемлющей инструкции def

492   
Глава 17. Области видимости   
 return action  
   
x = func()  
print(x(2)) # Выведет 16, 4 \*\* 2  
До того как появилось понятие областей видимости вложенных функций, для   
передачи значений из объемлющей области видимости в выражения lambda   
программисты использовали значения по умолчанию� точно так же, как и в   
случае с инструкциями def. Например, следующий фрагмент будет работать во   
всех версиях Python:  
def func():  
 x = 4  
 action = (lambda n, x=x: x \*\* n) # Передача x вручную  
 return action  
Поскольку lambda – это выражения, они естественно (и даже обычно) вклады-  
ваются внутрь инструкций def. Следовательно, именно они извлекли наиболь-  
шую выгоду от добавления областей видимости объемлющих функций в пра-  
вила поиска имен – в большинстве случаев отпадает необходимость передавать   
в выражения lambda аргументы со значениями по умолчанию.  
Области видимости и значения по умолчанию   
применительно к переменным цикла  
Существует одно известное исключение из правила, которое я только что дал:   
если lambda-выражение или инструкция def вложены в цикл внутри другой   
функции и вложенная функция ссылается на переменную из объемлющей об-  
ласти видимости, которая изменяется в цикле, все функции, созданные в этом   
цикле, будут иметь одно и то же значение – значение, которое имела перемен-  
ная на последней итерации.   
Например, ниже предпринята попытка создать список функций, каждая из   
которых запоминает текущее значение переменной i из объемлющей области   
видимости:  
>>> def makeActions():  
... acts = []  
... for i in range(5): # Сохранить каждое значение i  
... acts.append(lambda x: i \*\* x) # Все запомнят последнее значение i!  
... return acts  
...  
>>> acts = makeActions()  
>>> acts[0]  
<function <lambda> at 0x012B16B0>  
Такой подход не дает желаемого результата, потому что поиск переменной   
в объемлющей области видимости производится позднее, при вызове вложен-  
ных функций, в результате все они получат одно и то же значение (значение,   
которое имела переменная цикла на последней итерации). То есть каждая   
функция в списке будет возвращать 4 во второй степени, потому что во всех   
них переменная i имеет одно и то же значение:  
>>> acts[0](2) # Все возвращают 4 \*\* 2, последнее значение i  
16  
>>> acts[2](2) # Здесь должно быть 2 \*\* 2  
16

Области видимости и вложенные функции   
493  
>>> acts[4](2) # Здесь должно быть 4 \*\* 2  
16  
Это один из случаев, когда необходимо явно сохранять значение из объемлю-  
щей области видимости в виде аргумента со значением по умолчанию вместо   
использования ссылки на переменную из объемлющей области видимости.   
То есть, чтобы этот фрагмент заработал, необходимо передать текущее значе-  
ние переменной из объемлющей области видимости в виде значения по умол-  
чанию. Значения по умолчанию вычисляются в момент создания вложенной   
функции (а не когда она вызывается), поэтому каждая из них сохранит свое   
собственное значение i:  
>>> def makeActions():  
... acts = []  
... for i in range(5): # Использовать значения по умолчанию  
... acts.append(lambda x, i=i: i \*\* x) # Сохранить текущее значение i  
... return acts  
...  
>>> acts = makeActions()  
>>> acts[0](2) # 0 \*\* 2  
0  
>>> acts[2](2) # 2 \*\* 2  
4  
>>> acts[4](2) # 4 \*\* 2  
16  
Это достаточно замысловатый случай, но с ним можно столкнуться на практи-  
ке, особенно в программном коде, который генерирует функции-обработчики   
событий для элементов управления в графическом интерфейсе (например, об-  
работчики нажатия кнопок). Подробнее о значениях по умолчанию мы погово-  
рим в главе 18, а о lambda-выражениях – в главе 19, поэтому позднее вам может   
потребоваться вернуться к этому разделу.1  
Произвольное вложение областей видимости  
Прежде чем закончить это исследование, я должен заметить, что области види-  
мости могут вкладываться произвольно, но поиск будет производиться только   
в объемлющих функциях (не в классах, которые описываются в шестой части   
книги):  
>>> def f1():  
... x = 99  
... def f2():  
... def f3():  
... print(x) # Будет найдена в области видимости f1!  
1   
В разделе «Типичные ошибки при работе с функциями», в конце этой части книги,   
мы также увидим, что существуют определенные проблемы с использованием из-  
меняемых объектов, таких как списки и словари, при использовании их в качестве   
значений по умолчанию для аргументов (например, def f(a=[])). Так как значения по   
умолчанию реализованы в виде отдельных объектов, присоединяемых к функциям,   
изменяемые объекты по умолчанию сохраняют свое состояние от вызова к вызову,   
а не инициализируются заново при каждом вызове. В зависимости от точки зрения,   
эта особенность может рассматриваться как преимущество, позволяющее сохранять   
информацию о состоянии, или как недостаток. Подробнее об этом будет говориться   
в конце главы 20.

494   
Глава 17. Области видимости   
... f3()  
... f2()  
...  
>>> f1()  
99  
Интерпретатор будет искать переменную в локальных областях видимости   
всех объемлющих инструкций def, начиная от внутренних к внешним, выше   
локальной области видимости и ниже глобальной области видимости модуля.   
Однако такой программный код едва ли может получиться на практике. В язы-  
ке Python считается, что плоское лучше вложенного, – ваша жизнь и жизнь ва-  
ших коллег будет проще, если вы сведете к минимуму количество вложенных   
определений функций.  
Инструкция nonlocal  
В предыдущем разделе мы узнали, что вложенная функция может получать   
значения переменных в области видимости объемлющей функции даже после   
того, как эта функция вернет управление. Оказывается, в Python 3.0 также   
имеется возможность изменять значения переменных в области видимости   
объемлющей функции – при условии, что они объявлены с помощью инструк-  
ции nonlocal. Эта инструкция позволяет вложенным функциям не только чи-  
тать, но и изменять значения переменных в областях видимости объемлющих   
функций.  
Инструкция nonlocal – близкий родственник инструкции global, описанной   
выше. Подобно инструкции global, nonlocal объявляет имена, которые будут   
изменяться в теле функции и которые находятся в объемлющей области види-  
мости. Однако, в отличие от инструкции global, nonlocal применяется только   
к областям видимости объемлющих функций и не затрагивает глобальную об-  
ласть видимости модуля. Кроме того, в отличие от инструкции global, имена,   
перечисленные в инструкции nonlocal, должны фактически существовать в об-  
ласти видимости, вмещающей функцию, где встречается это объявление, – они   
могут существовать только в объемлющей области видимости и не могут быть   
созданы первой инструкцией присваивания во вложенной функции.  
Другими словами, инструкция nonlocal позволяет присваивать значения пере-  
менным в объемлющих областях видимости и ограничивает поиск таких имен   
областями видимости объемлющих функций. В результате мы получаем более   
очевидный и более надежный инструмент реализации изменения информации   
в областях видимости для программ, где нежелательно или невозможно ис-  
пользовать для этих же целей классы с атрибутами.  
Основы использования инструкции nonlocal  
В версии Python 3.0 появилась новая инструкция nonlocal, которая приобрета-  
ет смысл только внутри функций:  
def func():  
 nonlocal name1, name2, ...  
Эта инструкция позволяет вложенным функциям изменять переменные, кото-  
рые определены в областях видимости синтаксически объемлющих функций.   
В Python 2.X (включая 2.6), когда одна функция объявляется внутри другой,

Инструкция nonlocal   
495  
вложенная функция может читать значения любых имен, которые были опре-  
делены с помощью инструкции присваивания в области видимости объемлю-  
щей функции, но она не может изменять их. В 3.0 объявление имен, находя-  
щихся в объемлющей области видимости, в инструкции nonlocal дает вложен-  
ной функции возможность присваивать им новые значения.  
Благодаря этому для вложенных функций обеспечивается возможность под-  
держивать доступную для изменения информацию о состоянии, которая вос-  
станавливается при последующих вызовах вложенной функции. Способность   
изменять информацию о состоянии увеличивает практическую ценность вло-  
женных функций (например, представьте, что в объемлющей области видимо-  
сти хранится счетчик). В Python 2.X для достижения аналогичного эффекта   
обычно используются классы или другие инструменты. Применение вложен-  
ных функций стало практически стандартным приемом, когда требуется обе-  
спечить сохранение состояния, при этом инструкция nonlocal еще больше рас-  
ширяет область их применения.  
Кроме того, что инструкция nonlocal позволяет изменять значения перемен-  
ных в объемлющих функциях, она также ограничивает область поиска имен –   
подобно инструкции global, инструкция nonlocal вынуждает интерпретатор   
начинать поиск с областей видимости объемлющих функций, пропуская ло-  
кальную область видимости функции. То есть, кроме всего прочего, инструк-  
ция nonlocal означает: «пропустить локальную область видимости при поиске   
имен».  
На практике имена, перечисленные в инструкции nonlocal, должны быть опре-  
делены в объемлющих функциях к моменту, когда поток управления достиг-  
нет инструкции nonlocal� в противном случае будет возбуждено исключение.   
По своему действию инструкция nonlocal близко напоминает global: объявле-  
ние global означает, что имена находятся в глобальной области видимости вме-  
щающего модуля, а объявление nonlocal означает, что они находятся в области   
видимости вмещающих функций. Впрочем, инструкция nonlocal даже более   
строгая – она ограничивает область поиска только областями видимости объ-  
емлющих функций. То есть нелокальные имена могут присутствовать только   
в областях видимости объемлющих функций.  
Кроме того, инструкция nonlocal вообще не вносит никаких изменений в пра-  
вило поиска имен – поиск будет выполняться в полном соответствии с прави-  
лом «LEGB», описанным выше. Основное назначение инструкции nonlocal со-  
стоит в том, чтобы обеспечить возможность не только получения, но и измене-  
ния значений переменных в объемлющих областях видимости. Однако, если   
быть более точными, инструкции global и nonlocal несколько ограничивают   
правила поиска:  
 •  
global вынуждает интерпретатор начинать поиск имен с области объемлю-  
щего модуля и позволяет присваивать переменным новые значения. Об-  
ласть поиска простирается вплоть до встроенной области видимости, если   
искомое имя не будет найдено в модуле, при этом операция присваивания   
значений глобальным именам всегда будет создавать или изменять пере-  
менные в области видимости модуля.  
 •  
nonlocal ограничивает область поиска областями видимости объемлющих   
функций� она требует, чтобы перечисленные в инструкции имена уже су-  
ществовали, и позволяет присваивать им новые значения. В область поиска   
не входят глобальная и встроенная области видимости.

496   
Глава 17. Области видимости   
В Python 2.6 допускается ссылаться на имена в областях видимости объем-  
Python 2.6 допускается ссылаться на имена в областях видимости объем-  
 2.6 допускается ссылаться на имена в областях видимости объем-  
лющих функций, но присвоить им новые значения невозможно. При этом для   
сохранения информации о состоянии и достижения того же эффекта, кото-  
рый дает применение инструкции nonlocal, вы можете использовать классы   
с атрибутами (что в некоторых случаях является даже более удачным реше-  
нием). Иногда для аналогичных целей можно также использовать глобальные   
переменные и атрибуты функций. Подробнее об этом мы поговорим чуть ниже,   
а пока обратимся к программному коду, чтобы конкретизировать все вышеска-  
занное.  
Инструкция nonlocal в действии  
Все примеры, что приводятся ниже, выполнялись в Python 3.0. Обращение   
к переменным в области видимости объемлющих функций действует точно так   
же, как и в Python 2.6. Ниже приводится пример функции tester, которая соз-  
дает и возвращает вложенную функцию nested. Обращение к переменной state   
из вложенной функции отображается на локальную область видимости функ-  
ции tester, с применением привычных правил поиска:  
C:\\misc> c:\python30\python  
>>> def tester(start):  
... state = start # Обращение к нелокальным переменным   
... def nested(label): # действует как обычно  
... print(label, state) # Извлекает значение state из области  
... return nested # видимости объемлющей функции  
...  
>>> F = tester(0)  
>>> F(‘spam’)  
spam 0  
>>> F(‘ham’)  
ham 0  
По умолчанию изменение значения переменной в объемлющей области види-  
мости не допускается – это нормальная ситуация и в версии 2.6:  
>>> def tester(start):  
... state = start  
... def nested(label):  
... print(label, state)  
... state += 1 # По умолчанию не изменяется (как и в 2.6)  
... return nested  
...  
>>> F = tester(0)  
>>> F(‘spam’)  
UnboundLocalError: local variable ‘state’ referenced before assignment  
Использование инструкции nonlocal   
для изменения переменных  
Если теперь (при условии, что используется Python 3.0) переменную state, ло-  
кальную для функции tester, объявить в функции nested с помощью инструк-  
ции nonlocal, мы сможем изменять ее внутри функции nested. Этот прием дей-  
ствует, даже несмотря на то, что функция tester уже завершила работу к мо-  
менту, когда мы вызываем функцию nested через имя F:

Инструкция nonlocal   
497  
>>> def tester(start):  
... state = start # В каждом вызове сохраняется свое значение state  
... def nested(label):  
... nonlocal state # Объект state находится   
... print(label, state) # в объемлющей области видимости  
... state += 1 # Изменит значение переменной, объявленной как nonlocal  
... return nested  
...  
>>> F = tester(0)  
>>> F(‘spam’) # Будет увеличивать значение state при каждом вызове  
spam 0  
>>> F(‘ham’)  
ham 1  
>>> F(‘eggs’)  
eggs 2  
Как обычно, мы можем вызвать фабричную функцию tester множество раз,   
и каждый раз в памяти будет создаваться отдельная копия переменной state.   
Объект state, находящийся в объемлющей области видимости, фактически   
прикрепляется к возвращаемому объекту функции nested – каждый вызов   
функции tester создает новый, независимый объект state, благодаря чему изме-  
нение state в одной функции не будет оказывать влияния на другие. В следую-  
щем листинге приводится продолжение предыдущего интерактивного сеанса:  
>>> G = tester(42) # Создаст новую функцию, которая начнет счет с 42  
>>> G(‘spam’)  
spam 42  
>>> G(‘eggs’) # Обновит значение state до 43  
eggs 43  
>>> F(‘bacon’) # Но в функции F значение state останется прежним  
bacon 3 # Каждая новая функция получает свой экземпляр state  
Граничные случаи  
Важно не упускать из виду несколько моментов. Во-первых, в отличие от имен,   
перечисленных в инструкции global, имена в инструкции nonlocal к момен-  
ту объявления уже должны существовать в области видимости объемлющей   
функции, в противном случае интерпретатор возбудит исключение – нельзя   
создавать имена в объемлющей области видимости с помощью инструкции   
присваивания:  
>>> def tester(start):  
... def nested(label):  
... nonlocal state # Нелокальные переменные должны существовать!  
... state = 0  
... print(label, state)  
... return nested  
...  
SyntaxError: no binding for nonlocal ‘state’ found  
   
>>> def tester(start):  
... def nested(label):  
... global state # Глобальные переменные могут отсутствовать  
... state = 0 # Создаст переменную в области видимости модуля  
... print(label, state)  
... return nested  
...

498   
Глава 17. Области видимости   
>>> F = tester(0)  
>>> F(‘abc’)  
abc 0  
>>> state  
0  
Во-вторых, инструкция nonlocal ограничивает область поиска имен перемен-  
ных только областями видимости объемлющих функций – поиск нелокальных   
переменных не производится за пределами инструкций def ни в глобальной об-  
ласти видимости объемлющего модуля, ни во встроенной области видимости,   
даже если переменные с такими именами там существуют:  
>>> spam = 99  
>>> def tester():  
... def nested():  
... nonlocal spam # Переменная должна быть внутри def, а не в модуле!  
... print(‘Current=’, spam)  
... spam += 1  
... return nested  
...  
SyntaxError: no binding for nonlocal ‘spam’ found  
Эти ограничения станут понятны, как только вы поймете, что иначе интер-  
претатор не смог бы определить, в какой из объемлющих областей следует соз-  
давать совершенно новое имя. Например, где в предыдущем примере должна   
была бы быть создана переменная spam при выполнении инструкции присваи-  
вания – в функции tester или в объемлющем модуле? Из-за этой неоднозначно-  
сти интерпретатор вынужден определять местоположение нелокальных имен   
в момент создания функции, а не в момент ее вызова.   
Когда следует использовать инструкцию nonlocal?  
Учитывая сложность работы с вложенными функциями, у вас может появить-  
ся вопрос: «Когда следует использовать инструкцию nonlocal?» Хотя это трудно   
заметить по нашим маленьким примерам, тем не менее проблема сохранения   
информации о состоянии имеет важное значение во многих программах. В язы-  
ке Python существуют различные способы «сохранения» информации между   
вызовами функций и методов. Каждый из способов имеет свои преимущества,   
однако, когда речь заходит о переменных в областях видимости объемлющих   
функций, инструкция nonlocal обеспечивает более оптимальное решение – она   
позволяет создавать множество копий переменных, доступных для изменения,   
и способна удовлетворить наиболее простые потребности там, где использова-  
ние классов может быть нежелательным.  
Как мы видели в предыдущем разделе, следующий программный код позволя-  
ет сохранять и изменять переменные в объемлющей области видимости. Каж-  
дый вызов функции tester создает небольшой самостоятельный пакет инфор-  
мации, доступной для изменения, имена в котором не вступают в конфликт   
с именами в других частях программы:  
def tester(start):  
 state = start # Каждый вызов сохраняет отдельный экземпляр state  
 def nested(label):  
 nonlocal state # Объект state находится в объемлющей области видимости  
 print(label, state)  
 state += 1 # Изменит значение переменной, объявленной как nonlocal

Инструкция nonlocal   
499  
 return nested  
   
F = tester(0)  
F(‘spam’)  
К сожалению, этот программный код будет работать только в Python 3.0. Если   
вы используете Python 2.6, используйте другие способы, в зависимости от пре-  
Python 2.6, используйте другие способы, в зависимости от пре-  
 2.6, используйте другие способы, в зависимости от пре-  
следуемых целей. В следующих двух разделах приводятся некоторые альтер-  
нативные решения.  
Сохранение информации в глобальных переменных  
Обычно для достижения эффекта, который дает применение инструкции nonlo-  
cal, в Python 2.6 и в более ранних версиях достаточно просто переместить пере-  
Python 2.6 и в более ранних версиях достаточно просто переместить пере-  
 2.6 и в более ранних версиях достаточно просто переместить пере-  
менную state в глобальную область видимости (в область видимости модуля):  
>>> def tester(start):  
... global state # Переместить в область видимости модуля  
... state = start # global позволяет изменять переменные, находящиеся   
... def nested(label): # в области видимости модуля  
... global state  
... print(label, state)  
... state += 1  
... return nested  
...  
>>> F = tester(0)  
>>> F(‘spam’) # Каждый вызов будет изменять глобальную   
spam 0 # переменную state  
>>> F(‘eggs’)  
eggs 1  
Этот прием может использоваться в данном конкретном случае, но он требует   
наличия объявлений global в обеих функциях и может приводить к конфлик-  
там имен в глобальной области видимости (что если имя «state» уже исполь-  
state» уже исполь-  
» уже исполь-  
зуется?). Хуже всего, что этот способ позволяет создать в области видимости   
модуля лишь одну копию информации о состоянии – если вызвать функцию   
tester еще раз, значение переменной state будет сброшено в исходное состоя-  
ние, то есть состояние, измененное предыдущими вызовами, будет затерто:  
>>> G = tester(42) # Сбросит значение единственной копии state   
>>> G(‘toast’) # в глобальной области видимости  
toast 42  
>>> G(‘bacon’)  
bacon 43  
>>> F(‘ham’) # Ой – значение моего счетчика было затерто!  
ham 44  
Как было показано выше, когда вместо инструкции global используется ин-  
струкция nonlocal, каждый вызов функции tester сохраняет отдельную, уни-  
кальную копию объекта state.  
Сохранение информации с помощью классов   
(предварительное знакомство)   
Другой способ сохранения доступной для изменения информации о состоянии   
в Python 2.6 и в более ранних версиях заключается в использовании классов   
с атрибутами. Он обеспечивает более явный доступ к информации, по срав-

500   
Глава 17. Области видимости   
нению с неявной магией правил поиска имен в областях видимости. В качестве   
дополнительного преимущества каждый экземпляр класса получает свежую   
копию информации о состоянии, как естественный побочный эффект объект-  
ной модели в языке Python.  
Мы еще не изучали классы во всех подробностях, но в качестве предваритель-  
ного знакомства взглянем на комбинацию функций tester/nested, использовав-  
шихся ранее, как на класс, – информация о состоянии может быть записана   
в объекты явно, после их создания. Чтобы понять следующий пример, вам не-  
обходимо знать, что инструкция def внутри инструкции class действует точно   
так же, как и за ее пределами. За исключением того, что функция, определяе-  
мая внутри класса, автоматически получает аргумент self, ссылающийся на   
объект, относительно которого был произведен вызов (экземпляр класса, или   
объект, создается обращением к имени самого класса как к функции):  
>>> class tester: # Альтернативное решение на основе классов (Часть VI)  
... def \_\_init\_\_(self, start): # Конструктор объекта,  
... self.state = start # сохранение информации в новом объекте  
... def nested(self, label):  
... print(label, self.state) # Явное обращение к информации  
... self.state += 1 # Изменения всегда допустимы  
...  
>>> F = tester(0) # Создаст экземпляр класса, вызовет \_\_init\_\_  
>>> F.nested(‘spam’) # Ссылка на F будет передана в аргументе self  
spam 0  
>>> F.nested(‘ham’)  
ham 1  
>>> G = tester(42) # Каждый экземпляр получает свою копию информации  
>>> G.nested(‘toast’) # Изменения в одном объекте не сказываются на других  
toast 42  
>>> G.nested(‘bacon’)  
bacon 43  
>>> F.nested(‘eggs’) # В объекте F сохранилась прежняя информация  
eggs 2  
>>> F.state # Информация может быть получена за пределами класса  
3  
Добавив чуть-чуть волшебства, которое мы еще будем изучать далее в этой   
книге, мы могли бы заставить наш класс выглядеть, как обычная функция,   
достаточно лишь выполнить перегрузку оператора. Если обратиться к экзем-  
пляру класса, как к функции, то автоматически будет вызван метод \_\_call\_\_.   
Благодаря этому мы можем ликвидировать необходимость вызова именован-  
ного метода:  
>>> class tester:  
... def \_\_init\_\_(self, start):  
... self.state = start  
... def \_\_call\_\_(self, label): # Вызывается при вызове экземпляра  
... print(label, self.state) # Благодаря этому отпадает   
... self.state += 1 # необходимость в методе .nested()  
...  
>>> H = tester(99)  
>>> H(‘juice’) # Вызовет метод \_\_call\_\_  
juice 99  
>>> H(‘pancakes’)  
pancakes 100

Инструкция nonlocal   
501  
Не стремитесь пока разобраться во всех особенностях этого примера – мы будем   
изучать классы в шестой части книги, а с такими инструментами перегрузки   
операторов, как метод \_\_call\_\_, мы познакомимся в главе 29, поэтому сейчас   
вы можете отложить изучение этого примера на будущее. Главное сейчас – за-  
помнить, что классы обеспечивают более очевидный способ сохранения инфор-  
мации о состоянии, используя явное присваивание атрибутам вместо поиска   
переменных в областях видимости.  
Даже при том, что классы обеспечивают отличный способ сохранения ин-  
формации о состоянии, они могут оказаться слишком тяжеловесным ме-  
ханизмом в таких простых случаях, когда под информацией о состоянии   
подразумевается единственный счетчик. Подобные тривиальные ситуации   
встречаются в практике гораздо чаще, чем можно было бы подумать. В таких   
случаях вложенные инструкции def могут оказаться более легковесным спо-  
собом, чем классы, особенно если учесть, что вы пока не знакомы с объектно-  
ориентированным программированием. Кроме того, существуют ситуации,   
когда вложенные инструкции def обеспечивают более высокую производи-  
тельность по сравнению с классами (смотрите описание метода, основанного   
на применении декораторов, в главе 38, где приводятся примеры, выходящие   
далеко за рамки этой главы).  
Сохранение информации в атрибутах функций  
В качестве последнего примера рассмотрим, как добиться того же эффекта, что   
дает применение инструкции nonlocal, с помощью атрибутов функции – эти   
атрибуты, определяемые пользователем, присоединяются непосредственно   
к функции. Ниже приводится окончательная версия нашего примера, осно-  
ванного на применении этого приема, – в нем объявление nonlocal замещается   
атрибутом, присоединяемым к вложенной функции. Кому-то это может пока-  
заться недостаточно очевидным, но данный способ позволяет получать инфор-  
мацию о состоянии за пределами вложенной функции (при использовании ин-  
струкции nonlocal переменные, объявленные с ее помощью, доступны только   
внутри вложенной инструкции def):  
>>> def tester(start):  
... def nested(label):  
... print(label, nested.state) # nested – объемлющая область видимости  
... nested.state += 1 # Изменит атрибут, а не значение имени nested  
... nested.state = start # Инициализация после создания функции  
... return nested  
...  
>>> F = tester(0)  
>>> F(‘spam’) # F – это функция ‘nested’   
spam 0 # с присоединенным атрибутом state  
>>> F(‘ham’)  
ham 1  
>>> F.state # Атрибут state доступен за пределами функции  
2  
>>>  
>>> G = tester(42) # G имеет собственный атрибут state,   
>>> G(‘eggs’) # отличный от одноименного атрибута функции F  
eggs 42  
>>> F(‘ham’)  
ham 2

502   
Глава 17. Области видимости   
Этот программный код опирается на тот факт, что имя функции nested являет-  
ся локальной переменной в области видимости функции tester, включающей   
имя nested, – на это имя можно ссылаться и внутри функции nested. Кроме того,   
здесь используется то обстоятельство, что изменение самого объекта не являет-  
ся операцией присваивания, – операция увеличения значения nested.state из-  
меняет часть объекта, на который ссылается имя nested, а не саму переменную   
с именем nested. Поскольку во вложенной функции не выполняется операция   
присваивания, необходимость в инструкции nonlocal отпадает сама собой.  
Как видите, инструкции global и nonlocal, классы и атрибуты функций обеспе-  
чивают возможность сохранения информации о состоянии между вызовами.   
Глобальные переменные могут использоваться только для поддержки совмест-  
но используемых данных� для применения классов необходимо владеть прие-  
мами объектно-ориентированного программирования. И классы, и атрибуты   
функций позволяют получать информацию о состоянии за пределами вложен-  
ной функции. Как обычно, выбор наиболее оптимального инструмента зависит   
от целей, преследуемых в программе.  
В заключение  
В этой главе мы изучили такую ключевую концепцию, имеющую отношение   
к функциям, как области  видимости (как выполняется поиск переменных   
при обращениях к ним). Здесь мы узнали, что переменные считаются локаль-  
ными для определений функций, где выполняется присваивание значений   
этим переменным при условии, что они не объявлены с помощью инструкции   
global или nonlocal. Мы также познакомились с дополнительными особенно-  
стями областей видимости, включая области видимости вложенных функций   
и атрибуты функций. Наконец, мы рассмотрели некоторые рекомендации по   
проектированию приложений, такие как необходимость стремиться миними-  
зировать количество глобальных переменных и избегать межфайлового изме-  
нения переменных.  
В следующей главе мы продолжим знакомство с функциями исследованием   
второй ключевой концепции: передачи аргументов. Как вы узнаете в этой гла-  
ве, аргументы передаются в функции посредством операции присваивания, но   
в языке Python имеются инструменты, которые обеспечивают существенную   
гибкость при передаче аргументов функциям. Однако, прежде чем углубиться   
в эти темы, изучите контрольные вопросы к этой главе, чтобы закрепить зна-  
ния, полученные здесь.  
Закрепление пройденного  
Контрольные вопросы  
1. Что выведет следующий фрагмент и почему?  
>>> X = ‘Spam’  
>>> def func():  
... print(X)  
...  
>>> func()

Закрепление пройденного   
503  
2. Что выведет следующий фрагмент и почему?  
>>> X = ‘Spam’  
>>> def func():  
... X =’NI!’  
...  
>>> func()  
>>> print(X)  
3. Что выведет следующий фрагмент и почему?  
>>> X = ‘Spam’  
>>> def func():  
... X = ‘NI’  
... print(X)  
...  
>>> func()  
>>> print(X)  
4. Что выведет следующий фрагмент на этот раз и почему?  
>>> X = ‘Spam’  
>>> def func():  
... global X  
... X = ‘NI’  
...  
>>> func()  
>>> print(X)  
5. Что можно сказать об этом фрагменте – что он выведет и почему?  
>>> X = ‘Spam’  
>>> def func():  
... X = ‘NI’  
... def nested():  
... print(X)  
... nested()  
...  
>>> func()  
>>> X  
6. Что вы думаете об этом фрагменте: что он выведет в Python 3.0 и почему?  
>>> def func():  
... X = ‘NI’  
... def nested():  
... nonlocal X  
... X = ‘Spam’  
... nested()  
... print(X)  
...  
>>> func()  
7. Назовите три или более способов в языке Python сохранять информацию   
о состоянии в функциях.

504   
Глава 17. Области видимости   
Ответы  
1. В данном случае будет выведена строка ‘Spam’, потому что функция обраща-  
ется к глобальной переменной в объемлющем модуле (если внутри функции   
переменной не присваивается значение, она интерпретируется как глобаль-  
ная).  
2. В данном случае снова будет выведена строка ‘Spam’, потому что операция   
присваивания внутри функции создает локальную переменную и тем са-  
мым скрывает глобальную переменную с тем же именем. Инструкция print   
находит неизмененную переменную в глобальной области видимости.  
3. Будет выведена последовательность символов ‘Ni’ в одной строке и ‘Spam’ –   
в другой, потому что внутри функции инструкция print найдет локальную   
переменную, а за ее пределами – глобальную.  
4. На этот раз будет выведена строка ‘Ni’, потому что объявление global пред-  
писывает выполнять присваивание внутри функции переменной, находя-  
щейся в глобальной области видимости объемлющего модуля.  
5. В этом случае снова будет выведена последовательность символов ‘Ni’   
в одной строке и ‘Spam’ – в другой, потому что инструкция print во вложен-  
ной функции отыщет имя в локальной области видимости объемлющей   
функции, а инструкция print в конце фрагмента отыщет имя в глобальной   
области видимости.  
6. Этот фрагмент выведет строку ‘Spam’, так как инструкция nonlocal (доступ-  
ная в Python 3.0, но не в 2.6) означает, что операция присваивания внутри   
вложенной функции изменит переменную X в локальной области видимости   
объемлющей функции. Без этой инструкции операция присваивания клас-  
сифицировала бы переменную X как локальную для вложенной функции   
и создала бы совершенно другую переменную – в этом случае приведенный   
фрагмент вывел бы строку ‘NI’.  
7. Поскольку значения локальных переменных исчезают, когда функция воз-  
вращает управление, то информацию о состоянии в языке Python можно   
сохранять в глобальных переменных, для вложенных функций – в области   
видимости объемлющих функций, а также посредством аргументов со зна-  
чениями по умолчанию. Иногда можно использовать прием, основанный   
на сохранении информации в атрибутах, присоединяемых к функциям,   
вместо использования области видимости объемлющей функции. Альтер-  
нативный способ заключается в использовании классов и приемов ООП,   
который обеспечивает лучшую поддержку возможности сохранения ин-  
формации о состоянии, чем любой из предыдущих приемов, основанных на   
использовании областей видимости, потому что этот способ делает сохране-  
ние явным, позволяя выполнять присваивание значений атрибутам – эту   
возможность мы будем детально рассматривать в шестой части книги.

Глава 18.  
   
Аргументы  
В главе 17 исследовались особенности реализации областей видимости в языке   
Python – мест, где находятся переменные и где выполняется их поиск. Как мы   
узнали, место в программном коде, где выполняется присваивание перемен-  
ной, во многом определяет ее назначение. Эта глава продолжает тему функций,   
и здесь мы рассмотрим концепции языка Python, связанные с передачей аргу-  
ментов – способом передачи объектов в функции. Как вы увидите ниже, значе-  
ния аргументов (или параметров) присваиваются именам внутри функции, но   
они ближе к ссылкам на объекты, чем к областям видимости переменных. Мы   
также познакомимся с дополнительными инструментами языка Python, таки-  
Python, таки-  
, таки-  
ми как именованные аргументы, аргументы со значениями по умолчанию, и с   
коллекциями произвольных аргументов, которые обеспечивают высокую гиб-  
кость при передаче аргументов в функции.  
Передача аргументов  
Раньше уже говорилось, что передача аргументов производится посредством   
операции  присваивания. Здесь имеется несколько моментов, не всегда оче-  
видных для начинающих, о которых будет говориться в этом разделе. Ниже   
приводится несколько важных замечаний, касающихся передачи аргументов   
в функции:  
 •  
Аргументы передаются через автоматическое присваивание объектов ло-  
кальным переменным. Аргументы функции – ссылки на объекты, которые   
(возможно) используются совместно с вызывающей программой, – это все-  
го лишь результат еще одной из разновидностей операции присваивания.   
Ссылки в языке Python реализованы в виде указателей, поэтому все аргу-  
менты фактически передаются по указателям. Объекты, которые переда-  
ются в виде аргументов, никогда не копируются автоматически.  
 •  
Операция присваивания именам аргументов внутри функции не оказы-  
вает влияния на вызывающую программу. При вызове функции имена ар-  
гументов, указанные в ее заголовке, становятся новыми локальными име-  
нами в области видимости функции. Это не является совмещением имен   
между именами аргументов и именами в вызывающей программе.

506   
Глава 18. Аргументы   
 •  
Изменение внутри функции аргумента, который является изменяемым   
объектом, может оказывать влияние на вызывающую программу. C дру-  
C дру-  
 дру-  
гой стороны, так как аргументы – это всего лишь результат операции при-  
сваивания полученных объектов, функции могут воздействовать на полу-  
ченные изменяемые объекты и тем самым оказывать влияние на вызываю-  
щую программу. Изменяемые объекты могут рассматриваться функциями   
как средство ввода, так и вывода информации.  
За дополнительной информацией о ссылках обращайтесь к главе 6 – все, что   
там говорится, вполне применимо и к аргументам функций, несмотря на то,   
что присваивание именам аргументов выполняется автоматически и неявно.  
Схема передачи аргументов посредством присваивания, принятая в языке   
Python, это далеко не то же самое, что передача аргументов по ссылке в языке   
C++, но она очень близка к модели передачи аргументов в языке C:  
 •  
Неизменяемые объекты передаются «по значению». Такие объекты, как   
целые числа и строки, передаются в виде ссылок на объекты, а не в виде   
копий объектов, но так как неизменяемые объекты невозможно изменить   
непосредственно, передача таких объектов очень напоминает копирование.  
 •  
Изменяемые объекты передаются «по указателю». Такие объекты, как   
списки и словари, также передаются в виде ссылок на объекты, что очень   
похоже на то, как в языке C передаются указатели на массивы, – изменяе-  
мые объекты допускают возможность непосредственного изменения внутри   
функции так же, как и массивы в языке C.  
Конечно, если вы никогда ранее не использовали язык C, модель передачи ар-  
гументов в языке Python покажется вам еще проще – согласно этой модели вы-  
полняется присваивание объектов именам аргументов, и она одинаково рабо-  
тает с любыми объектами, как с изменяемыми, так и с неизменяемыми.  
Аргументы и разделяемые ссылки  
Для иллюстрации особенностей, свойственных механизму передачи аргумен-  
тов, рассмотрим следующий пример:  
>>> def f(a): # Имени a присваивается переданный объект   
... a = 99 # Изменяется только локальная переменная  
...  
>>> b = 88  
>>> f(b) # Первоначально имена a и b ссылаются на одно и то же число 88  
>>> print(b) # Переменная b не изменилась  
88  
В этом фрагменте в момент вызова функции f(b) переменной a присваивается   
объект 88, но переменная a существует только внутри вызванной функции. Из-  
менение переменной a внутри функции не оказывает влияния на окружение,   
откуда была вызвана функция, – просто в момент вызова создается совершенно   
новый объект a.  
Это именно то, что подразумевалось выше под словами «не является совмеще-  
нием имен», – операция присваивания значений аргументам внутри функции   
(такая как a=99) не изменяет, как по волшебству, переменные, подобные пере-  
менной b, находящиеся в области видимости программного кода, вызывающе-  
го функцию. Первоначально переменные и аргументы могут ссылаться на одни   
и те же объекты (фактически они являются указателями на эти объекты), но

Передача аргументов   
507  
лишь временно, в первые мгновения после вызова. Как только имени аргумен-  
та будет присвоено другое значение, эта связь исчезнет.  
По крайней мере, это относится к случаю, когда выполняется операция при-  
сваивания значений самим аргументам. Когда в аргументах передаются изме-  
няемые объекты, такие как списки и словари, мы должны понимать, что непо-  
средственные изменения в таких объектах никуда не исчезнут после заверше-  
ния функции и тем самым могут оказывать влияние на вызывающую програм-  
му. Ниже приводится пример, который демонстрирует эту особенность:  
>>> def changer(a, b): # В аргументах передаются ссылки на объекты  
... a = 2 # Изменяется только значение локального имени  
... b[0] = ‘spam’ # Изменяется непосредственно разделяемый объект  
...  
>>> X = 1  
>>> L = [1, 2] # Вызывающая программа  
>>> changer(X, L) # Передаются изменяемый и неизменяемый объекты  
>>> X, L # Переменная X – не изменилась, L - изменилась  
(1, [‘spam’, 2])  
В этом фрагменте функция changer присваивает значения аргументу a и компо-  
ненту объекта, на который ссылается аргумент b. Две операции присваивания   
в функции имеют незначительные синтаксические различия, но дают совер-  
шенно разные результаты:  
 •  
Так как a – это локальная переменная в области видимости функции, пер-  
вая операция присваивания не имеет эффекта для вызывающей програм-  
мы – она всего лишь изменяет локальную переменную a, записывая в нее   
ссылку на совершенно другой объект, и не изменяет связанное с ней имя X   
в вызывающей программе. Здесь все происходит точно так же, как в преды-  
дущем примере.  
 •  
b – также локальная переменная, но в ней передается изменяемый объект   
(список, на который ссылается переменная L в вызывающей программе).   
Поскольку вторая операция присваивания воздействует непосредственно   
на изменяемый объект, результат присваивания элементу b[0] в функции   
оказывает влияние на значение имени L после выхода из функции.  
В действительности вторая операция присваивания в функции changer не изме-  
няет объект b – она изменяет часть объекта, на который ссылается аргумент b.   
Это изменение оказывает влияние на вызывающую программу только потому,   
что измененный объект продолжает существовать после завершения функции.   
Значение переменной L никак не изменилось – она по-прежнему ссылается на   
тот же самый, пусть и измененный объект, но все выглядит так, как будто пе-  
ременная L изменилась после вызова функции. Это объясняется тем, что вну-  
три функции был изменен объект, на который она ссылается.   
Рисунок 18.1 иллюстрирует связи имя/объект, которые имеют место непосред-  
ственно сразу после вызова функции, но перед тем, как будет запущено ее тело.  
Если этот пример все еще кажется вам непонятным, попробуйте представить   
себе автоматическое присваивание переданным аргументам, как последова-  
тельность простых инструкций присваивания. Для первого аргумента опера-  
ции присваивания не оказывают влияния на вызывающую программу:  
>>> X = 1  
>>> a = X # Разделяют один и тот же объект  
>>> a = 2 # Изменяется только ‘a’, значение ‘X’ остается равным 1

508   
Глава 18. Аргументы   
>>> print(X)  
1  
Но для второго аргумента операция присваивания сказывается на значении   
переменной, участвующей в вызове, так как она производит непосредственное   
изменение объекта:  
>>> L = [1, 2]  
>>> b = L # Разделяют один и тот же объект  
>>> b[0] = ‘spam’ # Изменение в самом объекте: ‘L’ также изменяется  
>>> print(L)  
[‘spam’, 2]  
Вспомните обсуждение вопросов совместного использования изменяемых объ-  
ектов в главах 6 и 9, и вы узнаете это явление: непосредственное воздействие   
на изменяемый объект может сказываться на других ссылках на этот объект.   
В данном случае один из параметров функции играет одновременно роль сред-  
ства ввода и вывода информации.  
функция  
Имена  
Объекты  
вызывающая программа  
X  
L  
a  
b  
1  
[1,2]  
Рис. 18.1. Ссылки: аргументы. Так как аргументы передаются посредством   
операции присваивания, имена аргументов в функции  могут ссылаться на   
объекты, на которые ссылаются переменные, участвующие в вызове функ-  
ции. Следовательно, непосредственные воздействия на изменяемые аргумен-  
ты внутри функции могут оказывать влияние на вызывающую программу.   
В данном случае a и b внутри функции изначально ссылаются на те же объ-  
екты, на которые ссылаются переменные X и L при первом вызове функции.   
Изменение списка, выполненное через переменную b, можно наблюдать в пере-  
менной L после того, как функция вернет управление  
Как избежать воздействий на изменяемые аргументы  
Такое поведение изменяемых объектов не является ошибкой – оно лишь от-  
ражает способ, каким передаются аргументы. В языке Python аргументы по   
умолчанию передаются в функции по ссылкам (то есть по указателям), потому   
что в большинстве случаев именно это и требуется. Это означает, что мы можем   
передавать крупные объекты в любые точки программы без создания множе-

Передача аргументов   
509  
ства копий, и легко изменять эти объекты в процессе работы. В действитель-  
ности, как мы увидим в шестой части книги, модель классов в языке Python   
в значительной степени опирается на возможность изменения объектом соб-  
ственного состояния с помощью аргумента «self».  
Однако если нам требуется избежать влияния непосредственных изменений   
объектов, производимых в функциях, на вызывающую программу, мы мо-  
жем просто явно копировать изменяемые объекты, как описывалось в главе 6.   
В случае с аргументами функций мы всегда можем скопировать список в точке   
вызова:  
L = [1, 2]  
changer(X, L[:]) # Передается копия, поэтому переменная ‘L’ не изменится  
Можно также создать копию внутри функции, если для нас нежелательно,   
чтобы функция изменяла объект, независимо от способа его передачи:  
def changer(a, b):  
 b = b[:] # Входной список копируется, что исключает воздействие  
 # на вызывающую программу  
 a = 2  
 b[0] = ‘spam’ # Изменится только копия списка  
Оба варианта копирования не мешают функции изменять объект, они лишь   
препятствуют воздействию этих изменений на вызывающую программу. Что-  
бы действительно предотвратить изменения, мы всегда можем преобразовать   
изменяемые объекты в неизменяемые, что позволит выявить источники про-  
блем. Например, при попытке изменения кортежа будет возбуждено исключе-  
ние:  
L = [1, 2]  
changer(X, tuple(L)) # Передается кортеж, поэтому попытка изменения   
 # возбудит исключение  
Здесь используется встроенная функция tuple, которая создает новый кортеж   
из всех элементов последовательности (в действительности – любого итерируе-  
мого объекта). Это особенно важно, потому что вынуждает писать функцию   
так, чтобы она никогда не пыталась изменять передаваемые ей аргументы. Од-  
нако такое решение накладывает на функцию больше ограничений, чем следу-  
ет, поэтому его вообще следует избегать (нельзя знать заранее, когда изменение   
аргументов окажется необходимым). Кроме того, при таком подходе функция   
теряет возможность применять к аргументу методы списков, включая методы,   
которые не производят непосредственных изменений.  
Главное, что следует запомнить, – функции могут оказывать воздействие   
на передаваемые им изменяемые объекты, такие как списки и словари. Это   
не всегда является проблемой и часто может приносить пользу. Кроме того,   
функции, которые изменяют объекты, наверняка проектировались и пред-  
назначались именно для этого – внесение изменений, скорее всего, является   
составной частью четко определенного интерфейса, который не следует нару-  
шать, создавая копии.  
Однако вам действительно необходимо знать об этой особенности – если из-  
менения в объектах происходят неожиданно для вас, проверьте вызываемые   
функции и в случае необходимости передавайте им копии объектов.

510   
Глава 18. Аргументы   
Имитация выходных параметров  
Мы уже обсудили инструкцию return и использовали ее в нескольких приме-  
рах. Эта инструкция может возвращать объект любого типа, поэтому с ее по-  
мощью можно возвращать сразу несколько значений, упаковав их в кортеж   
или в коллекцию любого другого типа. В языке Python фактически отсутству-  
Python фактически отсутству-  
 фактически отсутству-  
ет такое понятие, которое в некоторых других языках называется «передача   
аргументов по ссылке», однако мы можем имитировать такое поведение, воз-  
вращая кортеж и выполняя присваивание результатов оригинальным именам   
аргументов в вызывающей программе:  
>>> def multiple(x, y):  
... x = 2 # Изменяется только локальное имя  
... y = [3, 4]  
... return x, y # Новые значения возвращаются в виде кортежа  
...  
>>> X = 1  
>>> L = [1, 2]  
>>> X, L = multiple(X, L) # Результаты присваиваются именам   
>>> X, L # в вызывающей программе  
(2, [3, 4])  
Выглядит так, как будто функция возвращает два значения, но на самом   
деле – это единственный кортеж, состоящий из двух элементов, а необязатель-  
ные окружающие скобки просто опущены. После возврата из функции мож-  
но использовать операцию присваивания кортежа, чтобы извлечь отдельные   
элементы. (Если вы забыли, как это работает, вернитесь к разделу «Кортежи»   
в главе 4, к главе 9 и к разделу «Инструкции присваивания» в главе 11.) Та-  
кой прием позволяет имитировать выходные параметры, имеющиеся в других   
языках программирования, за счет использования явной операции присваива-  
ния. Переменные X и L изменятся после вызова функции, но только потому, что   
мы явно это предусмотрели.  
Распаковывание аргументов в Python 2.X: В предыдущем при-  
мере выполняется операция распаковывания кортежа, возвра-  
щаемого функцией и полученного операцией присваивания кор-  
тежа. В Python 2.6 существует возможность автоматического   
распаковывания кортежей, передаваемых функциям в виде ар-  
гументов. В версии 2.6 функции, объявленной как:   
def f((a, (b, c))):  
можно передавать кортежи, соответствующие ожидаемой   
структуре: в результате вызова f((1, (2, 3))) переменным a, b и c   
будут присвоены значения 1, 2 и 3 соответственно. Естественно,   
передаваемый кортеж может быть объектом, созданным перед   
вызовом (f(T)). Этот синтаксис инструкции def больше не поддер-  
живается в версии Python 3.0. Теперь подобные функции долж-  
Python 3.0. Теперь подобные функции долж-  
 3.0. Теперь подобные функции долж-  
ны объявляться так:  
def f(T): (a, (b, c)) = T  
А распаковывание должно производиться явной инструкцией   
присваивания. Эта явная форма может применяться в обеих

Специальные режимы сопоставления аргументов   
511  
версиях, 3.0 и 2.6. Операция распаковывания аргументов не-  
очевидна и редко используется на практике в Python 2.X. Кроме   
того, в заголовке функции, в версии 2.6, поддерживается только   
форма передачи кортежей – при попытке использовать в вер-  
сии 2.6 более общую форму присваивания последовательностей   
(например, def f((a, [b, c])):) будет возбуждено исключение с со-  
общением о синтаксической ошибке и требованием использо-  
вать явную форму присваивания.  
Синтаксис распаковывания кортежей в аргументах в версии 3.0   
также был запрещен к использованию в списках аргументов   
lambda-выражений: смотрите врезку «Придется держать в уме:   
генераторы списков и map», где приводятся примеры. Несколь-  
ко нарушая это правило, в версии 3.0 до сих пор сохранилась   
поддержка операции автоматического распаковывания корте-  
жей в циклах for – смотрите примеры в главе 13.  
Специальные режимы   
сопоставления аргументов  
Как только что было показано, в языке Python аргументы всегда передаются   
через операцию  присваивания – передаваемые объекты присваиваются име-  
нам, указанным в заголовке инструкции def. Однако на основе этой модели   
язык Python обеспечивает дополнительные возможности влиять на способ,   
которым объекты аргументов сопоставляются с именами аргументов в заго-  
ловке функции. Эти возможности можно и не использовать, но они позволяют   
писать функции, поддерживающие более гибкие схемы вызова, и вы можете   
встретиться с некоторыми библиотеками, требующими использования этого   
способа.  
По умолчанию сопоставление аргументов производится в соответствии с их   
позициями, слева направо, и функции должно передаваться столько аргумен-  
тов, сколько имен указано в заголовке функции. Но кроме этого существует   
возможность явно указывать соответствия между аргументами и именами,   
определять значения по умолчанию и передавать дополнительные аргументы.  
Основы  
Это достаточно сложный раздел, и прежде чем окунуться в обсуждение син-  
таксиса, я хотел бы подчеркнуть, что эти специальные режимы не являются   
обязательными и имеют отношение только к сопоставлению объектов и имен –   
основным механизмом передачи аргументов по-прежнему остается операция   
присваивания. Фактически некоторые из этих режимов предназначены в пер-  
вую очередь для разработчиков библиотек, а не для разработчиков приложе-  
ний. Но так как вы можете столкнуться с этими режимами, даже не используя   
их в своих программах, я коротко опишу их:  
Сопоставление по позиции: значения и имена ставятся в соответствие по   
порядку, слева направо  
Обычный случай, который мы использовали до сих пор, значения и имена   
аргументов ставятся в соответствие в порядке их следования слева направо.

512   
Глава 18. Аргументы   
Сопоставление по именам: соответствие определяется по указанным име-  
нам аргументов  
Вызывающая программа имеет возможность указать соответствие между   
аргументами функции и их значениями в момент вызова, используя син-  
таксис name=value.  
Значения по умолчанию: указываются значения аргументов, которые могут   
не передаваться  
Функции могут определять значения аргументов по умолчанию на тот слу-  
чай, если вызывающая программа передаст недостаточное количество зна-  
чений. Здесь также используется синтаксис name=value.  
Переменное число аргументов: прием произвольного числа аргументов, пози-  
ционных или именованных  
Функции могут использовать специальный аргумент, имени которого пред-  
шествует один или два символа \*, для объединения произвольного числа до-  
полнительных аргументов в коллекцию (эта особенность часто называется   
varargs, как в языке C, где также поддерживаются списки аргументов пере-  
менной длины).  
Переменное число аргументов: передача произвольного числа аргументов, по-  
зиционных или именованных  
Вызывающая программа также может использовать синтаксис с симво-  
лом \* для распаковывания коллекции аргументов в отдельные аргументы.   
В данном случае символ \* имеет обратный смысл по отношению к символу   
\* в заголовке функции – в заголовке он означает коллекцию произвольного   
числа аргументов, тогда как в вызывающей программе – передачу коллек-  
ции в виде произвольного числа отдельных аргументов.  
Только именованные аргументы: аргументы, которые должны передаваться   
только по имени  
В Python 3.0 (но не в 2.6) функции могут определять аргументы, которые   
должны передаваться по именам, то есть в виде именованных, а не позици-  
онных аргументов. Такие аргументы обычно используются для определе-  
ния параметров настройки, дополняющих фактические аргументы.  
Синтаксис сопоставления  
В табл. 18.1 приводится синтаксис использования специальных режимов со-  
поставления.  
Таблица 18.1. Виды сопоставления аргументов функций  
Синтаксис  
Местоположение  
Интерпретация  
func(value)  
Вызывающая   
программа  
Обычный аргумент: сопоставле-  
ние производится по позиции  
func(name=value)  
Вызывающая   
программа  
Именованный аргумент: сопо-  
ставление производится по ука-  
занному имени

Специальные режимы сопоставления аргументов   
513  
Синтаксис  
Местоположение  
Интерпретация  
func(\*sequence)  
Вызывающая   
программа  
Все объекты в указанной по-  
й по-  
 по-  
следовательности передаются   
как отдельные позиционные   
аргументы  
func(\*\*dict)  
Вызывающая   
программа  
Все пары ключ/значение в ука-  
занном словаре передаются как   
отдельные именованные аргу-  
менты   
def func(name)  
Функция  
Обычный аргумент: сопостав-  
ление производится по позиции   
или по имени  
def func(name=value)  
Функция  
Значение аргумента по умолча-  
нию, на случай, если аргумент   
не передается функции  
def func(\*name)  
Функция  
Определяет и объединяет все   
дополнительные аргументы   
в кортеж  
def func(\*\*name)  
Функция  
Определяет и объединяет все   
дополнительные именованные   
аргументы в словарь  
def func(\*args, name)  
def func(\*, name=value)  
Функция  
Аргументы, которые должны   
передаваться функции только   
по именам (3.0)  
Эти специальные режимы сопоставления делятся на случаи вызова функции   
и определения функции:  
 •  
В инструкции вызова  функции (первые четыре строки таблицы) при ис-  
пользовании простых значений соответствие именам аргументов опреде-  
ляется по позиции, но при использовании формы name=value соответствие   
определяется по именам аргументов – это называется передачей именован-  
ных аргументов. Использование форм \*sequence и \*\*dict в вызовах функций   
позволяет передавать произвольное число объектов по позиции или по име-  
нам в виде последовательностей и словарей соответственно.  
 •  
В заголовке  функции (остальная часть таблицы) при использовании про-  
стых значений соответствие определяется по позиции или по имени, в зави-  
симости от того, как вызывающая программа передает значения, но при ис-  
пользовании формы name=value определяются значения по умолчанию. При   
использовании формы \*name все дополнительные позиционные аргументы   
объединяются в кортеж, а при использовании формы \*\*name все дополни-  
тельные именованные аргументы объединяются в словарь. В версии Py-  
Py-  
thon 3.0 и выше любые обычные аргументы или аргументы со значениями   
по умолчанию, следующие за формой \*name или за единственным символом   
\*, являются именованными аргументами, которые при вызове функции   
должны передаваться только по имени.

514   
Глава 18. Аргументы   
Наиболее часто в программном коде на языке Python используются форма пе-  
редачи именованных аргументов и аргументов со значениями по умолчанию.   
Мы уже встречались с обеими формами ранее в книге:  
 •  
Мы уже пользовались именованными аргументами для передачи необяза-  
тельных параметров функции print в Python 3.0, но они имеют более уни-  
Python 3.0, но они имеют более уни-  
 3.0, но они имеют более уни-  
версальный характер – именованные аргументы позволяют указывать   
значения аргументов вместе с их именами, чтобы придать вызову функции   
больше смысла.   
 •  
Со значениями по умолчанию мы также уже встречались, когда рассматри-  
вали способы передачи значений из объемлющей области видимости, но   
на самом деле эта форма имеет более широкую область применения – она   
позволяет определять необязательные аргументы и указывать значения по   
умолчанию в определении функции.  
Как мы увидим далее, комбинирование аргументов со значениями по умолча-  
нию в заголовках функций с именованными аргументами в вызовах функций   
дает нам возможность выбирать и переопределять значения, используемые по   
умолчанию.  
Специальные режимы сопоставления позволяют обеспечить свободу выбора   
числа аргументов, которые должны передаваться функции в обязательном по-  
рядке. Если функция определяет значения по умолчанию, они будут исполь-  
зоваться, когда функции передается недостаточное число аргументов. Если   
функция использует форму \* определения списка аргументов переменной дли-  
ны, она сможет принимать большее число аргументов – дополнительные аргу-  
менты будут собраны в структуру данных под именем с символом \*.  
Тонкости сопоставления  
Ниже приводится несколько правил в языке Python, которым вам необходимо   
следовать, если у вас появится потребность использовать специальные режи-  
мы сопоставления аргументов:  
 •  
В вызове функции аргументы должны указываться в следующем порядке:   
любые позиционные аргументы (значения), за которыми могут следовать   
любые именованные аргументы (name=value) и аргументы в форме \*sequence,   
за которыми могут следовать аргументы в форме \*\*dict.  
 •  
В заголовке функции аргументы должны указываться в следующем поряд-  
ке: любые обычные аргументы (name), за которыми могут следовать аргумен-  
ты со значениями по умолчанию (name=value), за которыми следуют аргумен-  
ты в форме \*name (или \* в 3.0), если имеются, за которыми могут следовать   
любые имена или пары name=value аргументов, которые передаются только   
по имени (в 3.0), за которыми могут следовать аргументы в форме \*\*name.  
В обоих случаях, и в вызове, и в заголовке функции, форма \*\*arg должна следо-  
вать последней в списке. Если попытаться расставить аргументы в любом дру-  
гом порядке, вы получите сообщение о синтаксической ошибке, потому что все   
остальные комбинации могут порождать неоднозначность. Действия, которые   
выполняет интерпретатор при сопоставлении аргументов перед присваивани-  
ем, грубо можно описать так:  
1. Сопоставление неименованных аргументов по позициям.  
2. Сопоставление именованных аргументов по именам.

Специальные режимы сопоставления аргументов   
515  
3. Сопоставление дополнительных неименованных аргументов с кортежем   
\*name.  
4. Сопоставление дополнительных именованных аргументов со словарем   
\*\*name.  
5. Сопоставление значений по умолчанию с отсутствующими именованными   
аргументами.  
После этого интерпретатор убеждается, что каждому аргументу соответству-  
ет только одно значение, – в противном случае возбуждается исключение. По   
окончании сопоставления всех аргументов интерпретатор связывает имена ар-  
гументов с полученными объектами.  
В действительности интерпретатор использует более сложный алгоритм сопо-  
ставления (например, он также должен учитывать сопоставление аргументов,   
которые могут передаваться только по имени, – появившихся в версии 3.0),   
поэтому мы оставим более точное описание за руководством по языку програм-  
мирования Python. Хотя знание этого алгоритма и не является обязательным,   
тем не менее его понимание может помочь разобраться в некоторых сложных   
ситуациях, особенно когда в деле замешаны режимы сопоставления.  
В Python 3.0 имена аргументов в заголовке функции могут так-  
же снабжаться аннотациями в форме name:value (или name:value=-  
default, если имеется значение по умолчанию). Это просто воз-  
можность вставлять дополнительное описание аргументов, ко-  
торая никак не влияет и не изменяет правила, определяющие   
порядок следования аргументов. Сама функция также может   
снабжаться аннотацией в форме def f()->value. Подробнее об ан-  
нотациях функций рассказывается в главе 19.  
Примеры использования именованных аргументов   
и значений по умолчанию  
Предыдущие пояснения в программном коде выглядят гораздо проще. Если   
не используются какие-то специальные формы сопоставления, по умолчанию   
сопоставление значений и имен аргументов производится по позиции, сле-  
ва направо, как и в большинстве других языков. Например, если определена   
функция, которая требует передачи трех аргументов, она должна вызываться   
с тремя аргументами:  
>>> def f(a, b, c): print(a, b, c)  
...  
Здесь значения передаются по позиции – имени a соответствует значение 1,   
имени b соответствует значение 2 и так далее (этот пример будет работать в обе-  
их версиях Python, 3.0 и 2.6, только в Python 2.6 в выводе появятся лишние   
круглые скобки, так как в этой версии вызов функции print интерпретируется   
как вывод кортежа):  
>>> f(1, 2, 3)  
1 2 3

516   
Глава 18. Аргументы   
Именованные аргументы  
В языке Python существует возможность явно определить соответствия между   
значениями и именами аргументов при вызове функции. Именованные аргу-  
менты позволяют определять соответствие по именам, а не по позициям:  
>>> f(c=3, b=2, a=1)  
1 2 3  
В этом вызове c=3, например, означает, что значение 3 передается функции   
в аргументе с именем c. Говоря более формальным языком, интерпретатор со-  
поставляет имя с в вызове функции с именем аргумента с в заголовке опреде-  
ления функции и затем передает значение 3 в этот аргумент. Результат этого   
вызова ничем не будет отличаться от предыдущего. Обратите внимание, что   
при использовании именованных аргументов порядок их следования не имеет   
никакого значения, потому что сопоставление производится по именам, а не   
по позициям. Существует даже возможность объединять передачу аргументов   
по позициям и по именам в одном вызове. В этом случае сначала будут сопо-  
ставлены все позиционные аргументы, слева направо, а потом будет выполнено   
сопоставление именованных аргументов:  
>>> f(1, c=3, b=2)  
1 2 3  
Большинство из тех, кто впервые сталкивается с такой возможностью, зада-  
ются вопросом: где такая возможность может пригодиться? Именованные ар-  
гументы в языке Python обычно играют две роли. Во-первых, они делают вы-  
зовы функций более описательными (представьте, что вы используете имена   
аргументов более осмысленные, чем простые a, b и c). Например, такой вызов:  
func(name=’Bob’, age=40, job=’dev’)  
несет больше смысла, чем вызов с тремя простыми значениями, разделенными   
запятыми, – имена аргументов играют роль меток для данных, участвующих   
в вызове. Во-вторых, именованные аргументы используются вместе со значе-  
ниями по умолчанию, о которых мы поговорим далее.  
Значения по умолчанию  
О значениях по умолчанию мы немного говорили ранее, когда обсуждали об-  
ласти видимости вложенных функций. Если коротко, значения по умолчанию   
позволяют сделать отдельные аргументы функции необязательными – если   
значение не передается при вызове, аргумент получит значение по умолчанию,   
перед тем как будет запущено тело функции. Например, ниже приводится   
функция, в которой один аргумент является обязательным, а два имеют зна-  
чения по умолчанию:  
>>> def f(a, b=2, c=3): print a, b, c  
...  
При вызове такой функции мы обязаны передать значение для аргумента a, по   
позиции или по имени, а значения для аргументов b и с можно опустить. Если   
значения аргументов b и с будут опущены, они примут значения по умолчанию   
2 и 3 соответственно:  
>>> f(1)  
1 2 3

Специальные режимы сопоставления аргументов   
517  
>>> f(a=1)  
1 2 3  
Если функции передать только два значения, аргумент с примет значение по   
умолчанию, а если три – ни одно из значений по умолчанию не будет исполь-  
зовано:  
>>> f(1, 4)  
1 4 3  
>>> f(1, 4, 5)  
1 4 5  
Наконец, ниже приводится пример взаимодействия режимов передачи значе-  
ний по именам и по умолчанию. Поскольку именованные аргументы могут на-  
рушать обычный порядок следования аргументов слева направо, они, по сути,   
позволяют «перепрыгивать» через аргументы со значениями по умолчанию:  
>>> f(1, c=6)  
1 2 6  
Здесь значение 1 будет сопоставлено с аргументом по позиции, аргумент c по-  
лучит значение 6, а аргумент b, в середине, – значение по умолчанию 2.  
Не путайте синтаксические конструкции name=value в заголовке функции и в   
вызове функции – в вызове она означает использование режима сопоставления   
значения с именованным аргументом, а в заголовке определяет значение по   
умолчанию для необязательного аргумента. В обоих случаях это не инструк-  
ция присваивания – это особая синтаксическая конструкция для этих двух   
случаев, которая модифицирует механику сопоставления значений с именами   
аргументов, используемую по умолчанию.  
Комбинирование именованных аргументов   
и значений по умолчанию  
Ниже приводится немного больший по объему пример, демонстрирующий ис-  
пользование именованных аргументов и аргументов со значениями по умолча-  
нию. В этом примере вызывающая программа всегда должна передавать функ-  
ции как минимум два аргумента (spam и eggs), два других аргумента являются   
необязательными. В случае их отсутствия интерпретатор присвоит именам   
toast и ham значения по умолчанию, указанные в заголовке:  
def func(spam, eggs, toast=0, ham=0): # Первые 2 являются обязательными  
 print(spam, eggs, toast, ham)  
   
func(1, 2) # Выведет: (1, 2, 0, 0)  
func(1, ham=1, eggs=0) # Выведет: (1, 0, 0, 1)  
func(spam=1, eggs=0) # Выведет: (1, 0, 0, 0)  
func(toast=1, eggs=2, spam=3) # Выведет: (3, 2, 1, 0)  
func(1, 2, 3, 4) # Выведет: (1, 2, 3, 4)  
Обратите внимание еще раз: когда в вызовах используются именованные аргу-  
менты, порядок их следования не имеет значения, потому что сопоставление   
выполняется по именам, а не по позициям. Вызывающая программа обязана   
передать значения для аргументов spam и eggs, а сопоставление может выпол-  
няться как по позиции, так и по именам. Обратите также внимание на то, что   
форма name=value имеет разный смысл в вызове функции и в инструкции def   
(именованный аргумент – в вызове и значение по умолчанию – в заголовке).

518   
Глава 18. Аргументы   
Примеры произвольного числа аргументов  
Последние два расширения \* и \*\* механизма сопоставления аргументов и их   
значений предназначены для поддержки возможности передачи произвольно-  
го числа аргументов функциям. Оба варианта могут появляться как в опреде-  
лениях функций, так и в их вызовах, и в обоих случаях они имеют сходные   
назначения.  
Сбор аргументов в коллекцию  
В первом случае, в определении функции, выполняется сборка лишних пози-  
ционных аргументов в кортеж:  
>>> def f(\*args): print(args)  
...  
При вызове этой функции интерпретатор Python соберет все позиционные ар-  
гументы в новый кортеж и присвоит этот кортеж переменной args. Это будет   
обычный объект кортежа, поэтому из него можно извлекать элементы по ин-  
дексам, выполнять обход в цикле for и так далее:  
>>> f()  
()  
>>> f(1)  
(1,)  
>>> f(1,2,3,4)  
(1, 2, 3, 4)  
Комбинация \*\* дает похожий результат, но применяется при передаче имено-  
ванных аргументов – в этом случае аргументы будут собраны в новый словарь,   
который можно обрабатывать обычными инструментами, предназначенными   
для работы со словарями. В определенном смысле форма \*\* позволяет преоб-  
разовать аргументы, передаваемые по именам, в словари, которые можно будет   
обойти с помощью метода keys, итераторов словарей и так далее:  
>>> def f(\*\*args): print(args)  
...  
>>> f()  
{}  
>>> f(a=1, b=2)  
{‘a’: 1, ‘b’: 2}  
Наконец, в заголовках функций можно комбинировать обычные аргументы,   
\* и \*\* для реализации чрезвычайно гибких сигнатур вызова. Например, в сле-  
дующем фрагменте число 1 передается как позиционный аргумент, 2 и 3 объ-  
единяются в кортеж pargs с позиционными аргументами, а x и y помещаются   
в словарь kargs с именованными аргументами:  
>>> def f(a, \*pargs, \*\*kargs): print(a, pargs, kargs)  
...  
>>> f(1, 2, 3, x=1, y=2)  
1 (2, 3) {‘y’: 2, ‘x’: 1}  
Фактически все эти формы передачи аргументов можно объединять в еще бо-  
лее сложные комбинации, которые на первый взгляд могут показаться неодно-  
значными, – к этой идее мы еще вернемся ниже, в этой главе. А сейчас посмо-  
трим, как используются формы \* и \*\* в вызовах функций.

Специальные режимы сопоставления аргументов   
519  
Извлечение аргументов из коллекции  
В последних версиях Python форму \* можно также использовать в вызовах   
функций. В этом случае данная форма передачи аргументов имеет противо-  
положный смысл по сравнению с применением этой формы в определениях   
функций – она распаковывает, а не создает коллекцию аргументов. Например,   
можно передать в функцию четыре аргумента в виде кортежа и позволить ин-  
терпретатору распаковать их в отдельные аргументы:  
>>> def func(a, b, c, d): print(a, b, c, d)  
...  
>>> args = (1, 2)  
>>> args += (3, 4)  
>>> func(\*args)  
1 2 3 4  
Точно так же форма \*\* в вызовах функций распаковывает словари пар ключ/  
значение в отдельные аргументы, которые передаются по ключу:  
>>> args = {‘a’: 1, ‘b’: 2, ‘c’: 3}  
>>> args[‘d’] = 4  
>>> func(\*\*args)  
1 2 3 4  
Здесь также можно очень гибко комбинировать в одном вызове обычные пози-  
ционные и именованные аргументы:  
>>> func(\*(1, 2), \*\*{‘d’: 4, ‘c’: 4})  
1 2 4 4  
   
>>> func(1, \*(2, 3), \*\*{‘d’: 4})  
1 2 3 4  
   
>>> func(1, c=3, \*(2,), \*\*{‘d’: 4})  
1 2 3 4  
   
>>> func(1, \*(2, 3), d=4)  
1 2 3 4  
   
>>> f(1, \*(2,), c=3, \*\*{‘d’:4})  
1 2 3 4  
Такого рода программный код удобно использовать, когда заранее невозмож-  
но предсказать число аргументов, которые могут быть переданы функции –   
вы можете собирать коллекцию аргументов во время исполнения и вызывать   
функцию таким способом. Не путайте синтаксические конструкции \*/\*\* в заго-  
ловках функций и в их вызовах – в заголовке они означают сбор всех лишних   
аргументов в коллекцию, а в вызове выполняют распаковку коллекций в от-  
дельные аргументы.  
Как мы видели в главе 14, форма \*pargs в вызовах функций яв-  
ляется одной из разновидностей итерационного контекста, по-  
этому с технической точки зрения в таком виде допускается пе-  
редавать не только кортежи или последовательности других ти-  
пов, но и любые итерируемые объекты, как показано в этих при-  
мерах. Например, после символа \* можно передать объект   
файла, и он будет распакован в отдельные аргументы (например,   
func(\*open(‘fname’))).

520   
Глава 18. Аргументы   
Это обобщение поддерживается в обеих версиях Python, 3.0   
и 2.6, но она действует исключительно в вызовах функций –   
в форме \*pargs в вызовах функций можно передавать любые   
итерируемые объекты – та же самая форма в заголовке ин-  
струкции def всегда объединяет лишние позиционные аргумен-  
ты в кортеж. Такое поведение формы со звездочкой напомина-  
ет синтаксис расширенной операции \* распаковывания после-  
довательностей в Python 3, с которой мы встречались в главе 11   
(например, x, \*y = z), только эта конструкция всегда создает   
списки, а не кортежи.  
Обобщенные способы вызова функций  
Примеры в предыдущем разделе могут показаться чересчур простыми, но они   
демонстрируют способы использования функций, которые на практике при-  
меняются гораздо чаще, чем можно было бы подумать. В программе может   
потребоваться вызывать произвольные функции единообразным способом, не   
имея представления об их именах и аргументах. В действительности, истинное   
преимущество специального синтаксиса передачи переменного числа аргумен-  
тов («varargs») состоит в том, что он не требует от вас заранее знать количество   
обязательных аргументов в функции. Например, в программе можно исполь-  
зовать условную инструкцию if для выбора из множества функций и списков   
аргументов и вызывать любую из них единообразным способом:  
if <test>:  
 action, args = func1, (1,) # Вызвать func1 с 1 аргументом  
else:  
 action, args = func2, (1, 2, 3) # Вызвать func2 с 3 аргументами  
...  
action(\*args) # Фактический вызов универсальным способом  
В более общем случае синтаксис передачи произвольного числа аргументов   
удобно использовать всегда, когда перечень аргументов не известен заранее.   
Например, если пользователь выбирает функцию с помощью пользовательско-  
го интерфейса, для вас может оказаться невозможным записать явный вызов   
функции в программном коде. Чтобы решить эту проблему, можно просто по-  
строить список аргументов с применением операций над последовательностя-  
ми и вызвать требуемую функцию, воспользовавшись синтаксисом передачи   
произвольного числа аргументов:  
>>> args = (2,3)  
>>> args += (4,)  
>>> args  
(2, 3, 4)  
>>> func(\*args)  
Так как список аргументов передается функции в виде кортежа, програм-  
ма может создать его во время выполнения. Этот прием удобно использовать   
в функциях, которые тестируют или измеряют производительность других   
функций. Например, в следующем фрагменте мы реализовали поддержку вы-  
зова произвольных функций с любым количеством любых аргументов, пере-  
давая все аргументы, которые были получены:  
def tracer(func, \*pargs, \*\*kargs): # Принимает произвольные аргументы  
 print(‘calling:’, func.\_\_name\_\_)

Специальные режимы сопоставления аргументов   
521  
 return func(\*pargs, \*\*kargs) # Передает все полученные аргументы  
   
def func(a, b, c, d):  
 return a + b + c + d  
   
print(tracer(func, 1, 2, c=3, d=4))  
При вызове функции tracer все аргументы будут собраны в коллекции и пере-  
даны требуемой функции с использованием синтаксиса передачи произвольно-  
го количества аргументов:   
calling: func  
10  
Более крупные примеры, демонстрирующие этот прием, мы увидим ниже   
в этой книге – смотрите, в частности, пример в главе 20, демонстрирующий   
операции над последовательностями, и примеры различных декораторов в гла-  
ве 38.  
Исчезнувшая встроенная функция apply (Python 2.6)  
До появления версии Python 3.0 того же эффекта, который дает использование   
синтаксиса \*args и \*\*args в вызовах функций, можно было добиться с помощью   
встроенной функции apply. Эта оригинальная возможность была ликвидирова-  
на в версии 3.0, как избыточная (в версии 3.0 было убрано множество подобных   
устаревших инструментов, которые появились за долгие годы существования   
языка). Однако она все еще присутствует в Python 2.6, и вы вполне можете   
столкнуться с ней в старых программах, написанных для Python 2.X.  
Проще говоря, следующие две инструкции являются эквивалентными в верси-  
ях Python ниже версии 3.0:  
func(\*pargs, \*\*kargs) # Новейший синтаксис вызова: func(\*sequence, \*\*dict)  
   
apply(func, pargs, kargs) # Устаревшая функция: apply(func, sequence, dict)  
В качестве примера рассмотрим следующую функцию, которая принимает   
произвольное число позиционных и именованных аргументов:  
>>> def echo(\*args, \*\*kwargs): print(args, kwargs)  
...  
>>> echo(1, 2, a=3, b=4)  
(1, 2) {‘a’: 3, ‘b’: 4}  
В Python 2.6 мы могли бы использовать более обобщенную форму вызова с по-  
Python 2.6 мы могли бы использовать более обобщенную форму вызова с по-  
 2.6 мы могли бы использовать более обобщенную форму вызова с по-  
мощью функции apply или с использованием синтаксиса, который теперь явля-  
ется обязательным в 3.0:  
>>> pargs = (1, 2)  
>>> kargs = {‘a’:3, ‘b’:4}  
   
>>> apply(echo, pargs, kargs)  
(1, 2) {‘a’: 3, ‘b’: 4}  
   
>>> echo(\*pargs, \*\*kargs)  
(1, 2) {‘a’: 3, ‘b’: 4}  
Синтаксис распаковывания аргументов является более новым, чем функция   
apply. Он считается более предпочтительным вообще и обязательным в вер-  
сии 3.0. Новый синтаксис не только обеспечивает симметричность форм \*pargs

522   
Глава 18. Аргументы   
и \*\*kargs в заголовках инструкций def и фактически является более кратким,   
он также позволяет передавать дополнительные аргументы без необходимости   
вручную распаковывать последовательности и словари с аргументами:  
>>> echo(0, c=5, \*pargs, \*\*kargs) # Обычный, именованный,   
(0, 1, 2) {‘a’: 3, ‘c’: 5, ‘b’: 4} # \*sequence, \*\*dictionary  
Этот синтаксис вызова функций более универсален. А поскольку он является   
еще и обязательным в версии 3.0, вы должны полностью забыть о существова-  
нии функции apply (если только она не встретится вам в старых программах,   
написанных для Python 2.X, которые вам приходится использовать и сопрово-  
Python 2.X, которые вам приходится использовать и сопрово-  
 2.X, которые вам приходится использовать и сопрово-  
X, которые вам приходится использовать и сопрово-  
, которые вам приходится использовать и сопрово-  
ждать...).  
Python 3.0: аргументы, которые могут передаваться   
только по именам   
В версии Python 3.0 были обобщены правила, определяющие порядок следо-  
Python 3.0 были обобщены правила, определяющие порядок следо-  
 3.0 были обобщены правила, определяющие порядок следо-  
вания разных видов аргументов в заголовках функций, что позволяет нам   
определять аргументы, которые могут передаваться только в форме именован-  
ных  аргументов и никогда не будут заполняться значениями позиционных   
аргументов. Эту особенность удобно использовать, когда необходимо, чтобы   
функция могла принимать произвольное количество аргументов, а также ряд   
дополнительных параметров настройки.  
Синтаксически аргументы, которые могут передаваться только в виде имено-  
ванных аргументов, оформляются в виде обычных именованных аргументов,   
следующих за формой \*args в списке аргументов. Все такие аргументы могут   
передаваться в вызовы функций только по именам. Например, в следующем   
фрагменте аргумент a может передаваться как именованный или как позици-  
онный аргумент, в b собираются все дополнительные позиционные аргументы   
и аргумент c может передаваться только как именованный аргумент:  
>>> def kwonly(a, \*b, c):  
... print(a, b, c)  
...  
>>> kwonly(1, 2, c=3)  
1 (2,) 3  
>>> kwonly(a=1, c=3)  
1 () 3  
>>> kwonly(1, 2, 3)  
TypeError: kwonly() needs keyword-only argument c  
Чтобы показать, что функция не принимает списки аргументов произвольной   
длины, можно использовать одиночный символ \*, при этом она ожидает, что   
все следующие за звездочкой аргументы будут передаваться по именам. Сле-  
дующей функции аргумент a можно передать как позиционный или как име-  
нованный, но аргументы b и c могут передаваться только как именованные   
аргументы� при этом функция не может принимать списки аргументов произ-  
вольной длины:  
>>> def kwonly(a, \*, b, c):  
... print(a, b, c)  
...  
>>> kwonly(1, c=3, b=2)  
1 2 3  
>>> kwonly(c=3, b=2, a=1)

Специальные режимы сопоставления аргументов   
523  
1 2 3  
>>> kwonly(1, 2, 3)  
TypeError: kwonly() takes exactly 1 positional argument (3 given)  
>>> kwonly(1)  
TypeError: kwonly() needs keyword-only argument b  
Вы по-прежнему можете использовать значения по умолчанию для аргумен-  
тов, которые могут передаваться только в виде именованных, несмотря на   
то, что в заголовке функции они располагаются после символа \*. Следующей   
функции аргумент a можно передать как именованный или как позиционный,   
а аргументы b и c являются необязательными, но передаваться должны только   
в виде именованных аргументов:   
>>> def kwonly(a, \*, b=’spam’, c=’ham’):  
... print(a, b, c)  
...  
>>> kwonly(1)  
1 spam ham  
>>> kwonly(1, c=3)  
1 spam 3  
>>> kwonly(a=1)  
1 spam ham  
>>> kwonly(c=3, b=2, a=1)  
1 2 3  
>>> kwonly(1, 2)  
TypeError: kwonly() takes exactly 1 positional argument (2 given)  
Аргументы со значениями по умолчанию, которые могут передаваться только   
по именам, в действительности являются необязательными, а те же самые ар-  
гументы без значений по умолчанию превращаются в обязательные именован-  
ные аргументы:  
>>> def kwonly(a, \*, b, c=’spam’):  
... print(a, b, c)  
...  
>>> kwonly(1, b=’eggs’)  
1 eggs spam  
>>> kwonly(1, c=’eggs’)  
TypeError: kwonly() needs keyword-only argument b  
>>> kwonly(1, 2)  
TypeError: kwonly() takes exactly 1 positional argument (2 given)  
   
>>> def kwonly(a, \*, b=1, c, d=2):  
... print(a, b, c, d)  
...  
>>> kwonly(3, c=4)  
3 1 4 2  
>>> kwonly(3, c=4, b=5)  
3 5 4 2  
>>> kwonly(3)  
TypeError: kwonly() needs keyword-only argument c  
>>> kwonly(1, 2, 3)  
TypeError: kwonly() takes exactly 1 positional argument (3 given)  
Правила, определяющие порядок следования  
Наконец, важно отметить, что аргументы, которые могут передаваться только   
по именам, должны указываться после одиночного символа звездочки, но не

524   
Глава 18. Аргументы   
двойного� эти аргументы не могут располагаться после формы \*\*args представ-  
ления списка именованных аргументов произвольной длины, и пара символов   
\*\* без следующего за ними имени аргумента также не может появляться в спи-  
ске аргументов. В обоих случаях будет выведено сообщение о синтаксической   
ошибке:  
>>> def kwonly(a, \*\*pargs, b, c):  
SyntaxError: invalid syntax  
>>> def kwonly(a, \*\*, b, c):  
SyntaxError: invalid syntax  
Это означает, что аргументы, которые могут передаваться только по именам,   
в заголовке функции должны предшествовать форме \*\*args представления   
списка именованных аргументов произвольной длины и следовать за формой   
\*args представления списка позиционных аргументов произвольной длины,   
когда присутствуют обе формы. Всякий раз, когда именованный аргумент по-  
является перед формой \*args, он интерпретируется как позиционный аргумент   
со значением по умолчанию, но не как аргумент, который может передаваться   
только по имени:  
>>> def f(a, \*b, \*\*d, c=6): print(a, b, c, d) # Только именованные аргументы   
SyntaxError: invalid syntax # должны предшествовать \*\*!  
   
>>> def f(a, \*b, c=6, \*\*d): print(a, b, c, d) # Коллекции аргументов   
... # в заголовке  
>>> f(1, 2, 3, x=4, y=5) # Используется значение по умолчанию  
1 (2, 3) 6 {‘y’: 5, ‘x’: 4}  
   
>>> f(1, 2, 3, x=4, y=5, c=7) # Переопределение значения по умолчанию  
1 (2, 3) 7 {‘y’: 5, ‘x’: 4}  
   
>>> f(1, 2, 3, c=7, x=4, y=5) # Среди именованных аргументов  
1 (2, 3) 7 {‘y’: 5, ‘x’: 4}  
   
>>> def f(a, c=6, \*b, \*\*d): print(a, b, c, d) # c не является только  
... # именованным аргументом!  
>>> f(1, 2, 3, x=4)  
1 (3,) 2 {‘x’: 4}  
В вызовах функций используются похожие правила, определяющие порядок   
следования аргументов: когда функции передаются аргументы, которые могут   
быть только именованными, они должны располагаться перед формой \*\*args.   
При этом аргументы, которые могут передаваться только по именам, могут   
располагаться как перед формой \*args, так и после нее, а также могут вклю-  
чаться в словарь \*\*args:  
>>> def f(a, \*b, c=6, \*\*d): print(a, b, c, d) # Только именованные аргументы  
... # между \* и \*\*  
>>> f(1, \*(2, 3), \*\*dict(x=4, y=5)) # Распаковывание аргументов   
1 (2, 3) 6 {‘y’: 5, ‘x’: 4} # при вызове  
   
>>> f(1, \*(2, 3), \*\*dict(x=4, y=5), c=7) # Именованные аргументы   
SyntaxError: invalid syntax # после \*\*args!  
   
>>> f(1, \*(2, 3), c=7, \*\*dict(x=4, y=5)) # Переопределение значений   
1 (2, 3) 7 {‘y’: 5, ‘x’: 4} # по умолчанию  
   
>>> f(1, c=7, \*(2, 3), \*\*dict(x=4, y=5)) # Перед \* или после нее

Функция поиска минимума   
525  
1 (2, 3) 7 {‘y’: 5, ‘x’: 4}  
   
>>> f(1, \*(2, 3), \*\*dict(x=4, y=5, c=7)) # Только именованные аргументы   
1 (2, 3) 7 {‘y’: 5, ‘x’: 4} # внутри \*\*  
Изучите эти случаи самостоятельно и попробуйте сопоставить их с описанны-  
ми выше правилами, определяющими порядок следования аргументов. Может   
показаться, что эти, достаточно искусственные, примеры представляют наи-  
худшие случаи, но они вполне могут возникнуть на практике, особенно у тех,   
кто разрабатывает библиотеки и инструменты на языке Python для других   
программистов.  
Когда используются аргументы,   
которые могут передаваться только по именам?   
Итак, когда же бывает желательно использовать аргументы, которые могут   
передаваться только по именам? Если говорить кратко, они упрощают созда-  
ние функций, которые принимают произвольное количество позиционных   
аргументов и параметры настройки, передаваемые в виде именованных ар-  
гументов. Аргументы, которые передаются только по именам, можно и не ис-  
пользовать, но без их использования может потребоваться выполнить лишнюю   
работу, чтобы определить значения по умолчанию для таких параметров и про-  
верять, что не было передано лишних именованных аргументов.  
Представьте функцию, которая обрабатывает множество передаваемых ей объ-  
ектов и дополнительно принимает флаг трассировки:  
process(X, Y, Z) # Используется значение флага по умолчанию  
process(X, Y, notify=True) # значение флага определяется явно  
Без использования аргумента, который может передаваться только по имени,   
нам пришлось бы использовать обе формы, \*args и \*\*args, и вручную проверять   
именованные аргументы, а благодаря аргументу, который может передаваться   
только по имени, программный код получится компактнее. Следующее опре-  
деление функции гарантирует, что ни один из позиционных аргументов не бу-  
дет по ошибке сопоставлен с аргументом notify, и требует, чтобы этот параметр   
передавался по имени:  
def process(\*args, notify=False): ...  
Далее в этой главе, в разделе «Имитация функции print в Python 3.0», приво-  
print в Python 3.0», приво-  
 в Python 3.0», приво-  
Python 3.0», приво-  
 3.0», приво-  
дится более реалистичный пример, где мы продолжим обсуждение этой темы.   
Дополнительные примеры использования аргументов, которые могут пере-  
даваться только по именам, вы найдете в главе 20. А дополнительные расши-  
рения, которые можно использовать в определениях функций, появившиеся   
в Python 3.0, рассматриваются в обсуждении синтаксиса аннотаций, в главе 19.  
Функция поиска минимума  
Пришло время заняться чем-нибудь более практичным. Чтобы придать обсуж-  
дению больше конкретики, выполним упражнение, которое демонстрирует   
практическое применение механизмов сопоставления аргументов.   
Предположим, что вам необходимо написать функцию, которая способна нахо-  
дить минимальное значение из произвольного множества аргументов с произ-

526   
Глава 18. Аргументы   
вольными типами данных. То есть функция должна принимать ноль или более   
аргументов – столько, сколько вы пожелаете передать. Более того, функция   
должна работать со всеми типами объектов, имеющимися в языке Python: чис-  
лами, строками, списками, списками словарей, файлами и даже None.  
Первое требование представляет собой обычный пример того, как можно най-  
ти применение форме \*, – мы можем собирать аргументы в кортеж и выполнять   
их обход с помощью простого цикла for. Второе требование тоже не представля-  
ет никакой сложности: все типы объектов поддерживают операцию сравнения,   
поэтому нам не требуется учитывать типы объектов в функции (полиморфизм   
в действии) – мы можем просто слепо сравнивать объекты и позволить интер-  
претатору самостоятельно выбрать корректную операцию сравнения.  
Основное задание  
Ниже представлены три способа реализации этой функции, из которых по   
крайней мере один был предложен студентом:   
 •  
Первая версия функции извлекает первый аргумент (args – это кортеж) и об-  
ходит остальную часть коллекции, отсекая первый элемент (нет никакого   
смысла сравнивать объект сам с собой, особенно если это довольно крупная   
структура данных).  
 •  
Вторая версия позволяет интерпретатору самому выбрать первый аргумент   
и остаток, благодаря чему отпадает необходимость извлекать первый аргу-  
мент и получать срез.  
 •  
Третья версия преобразует кортеж в список с помощью встроенной функ-  
ции list и использует метод списка sort.  
Метод sort написан на языке C, поэтому иногда он может обеспечивать более   
высокую производительность по сравнению с другими версиями функции, но   
линейный характер сканирования в первых двух версиях в большинстве слу-  
чаев обеспечивает им более высокую скорость.1 Файл mins.py содержит реали-  
зацию всех трех решений:  
def min1(\*args):  
 res = args[0]  
 for arg in args[1:]:  
 if arg < res:  
 res = arg  
1   
В действительности все совсем не просто. Функция sort в языке Python написана на   
языке C и реализует высокоэффективный алгоритм, который пытается использовать   
существующий порядок следования сортируемых элементов. Этот алгоритм называ-  
ется «timsort» в честь его создателя Тима Петерса (Tim Peters), а в его документации   
говорится, что время от времени он показывает «сверхъестественную производи-  
тельность» (очень неплохую для сортировки!). Однако сортировка по своей природе   
является экспоненциальной операцией (в процессе сортировки необходимо делить   
последовательность на составляющие и снова объединять их много раз), тогда как   
другие версии используют линейные алгоритмы сканирования слева направо. В ре-  
зультате сортировка выполняется быстрее, когда элементы последовательности ча-  
стично упорядочены, и медленнее в других случаях. Даже при всем при этом произ-  
водительность самого интерпретатора может изменяться по времени, и тот факт, что   
сортировка реализована на языке C, может существенно помочь. Для более точного   
анализа производительности вы можете использовать модули time и timeit, с которы-  
ми мы познакомимся в главе 20.

Функция поиска минимума   
527  
 return res  
   
def min2(first, \*rest):  
 for arg in rest:  
 if arg < first:  
 first = arg  
 return first  
   
def min3(\*args):  
 tmp = list(args) # Или, в Python 2.4+: return sorted(args)[0]  
 tmp.sort()  
 return tmp[0]  
   
print(min1(3,4,1,2))  
print(min2(“bb”, “aa”))  
print(min3([2,2], [1,1], [3,3]))  
Все три решения дают одинаковые результаты. Попробуйте несколько раз   
вызвать функции в интерактивной оболочке, чтобы поэкспериментировать   
с ними самостоятельно:  
% python mins.py  
1  
aa  
[1, 1]  
Обратите внимание, что ни один из этих трех вариантов не выполняет проверку   
ситуации, когда функции не передается ни одного аргумента. Такую проверку   
можно было бы предусмотреть, но в этом нет никакой необходимости – во всех   
трех решениях интерпретатор автоматически возбудит исключение, если та   
или иная функция не получит ни одного аргумента. В первом случае исключе-  
ние будет возбуждено при попытке получить нулевой элемент� во втором – когда   
обнаружится несоответствие списка аргументов� и в третьем – когда функция   
попытается вернуть нулевой элемент.  
Это именно то, что нам нужно, потому что эти функции поддерживают поиск   
среди данных любого типа и не существует какого-то особого значения, кото-  
рое можно было бы вернуть в качестве признака ошибки. Из этого правила есть   
свои исключения (например, когда приходится выполнить дорогостоящие дей-  
ствия, прежде чем появится ошибка), но вообще лучше исходить из предполо-  
жения, что аргументы не будут вызывать ошибок в работе программного кода   
функции, и позволить интерпретатору возбуждать исключения, когда этого не   
происходит.  
Дополнительные баллы  
Студенты и читатели могут получить дополнительные баллы, если изменят   
эти функции так, что они будут отыскивать не минимальное, а максимальное   
значение. Сделать это достаточно просто: в первых двух версиях достаточно за-  
менить < на >, а третья версия должна возвращать не элемент tmp[0], а элемент   
tmp[-1]. Дополнительные баллы будут начислены тем, кто догадается изменить   
имя функции на «max» (хотя это совершенно необязательно).  
Кроме того, вполне возможно обобщить функцию так, что она будет отыски-  
вать либо минимальное, либо максимальное значение, определяя отношения   
элементов за счет интерпретации строки выражения с помощью таких средств,   
как встроенная функция eval (подробности в руководстве к библиотеке), или

528   
Глава 18. Аргументы   
передавая произвольную функцию сравнения. В файле minmax.py содержится   
реализация последнего варианта:  
def minmax(test, \*args):  
 res = args[0]  
 for arg in args[1:]:  
 if test(arg, res):  
 res = arg  
 return res  
   
def lessthan(x, y): return x < y # См. также: lambda  
def grtrthan(x, y): return x > y  
   
print(minmax(lessthan, 4, 2, 1, 5, 6, 3)) # Тестирование  
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))  
   
% python minmax.py  
1  
6  
Функции – это одна из разновидностей объектов, которые могут передаваться   
в функции, как в этом случае. Например, чтобы заставить функцию отыски-  
вать максимальное (или любое другое) значение, мы могли бы просто передать   
ей нужную функцию test. На первый взгляд может показаться, что мы дела-  
ем лишнюю работу, однако главное преимущество такого обобщения функций   
(вместо того, чтобы содержать две версии, отличающиеся единственным сим-  
волом) заключается в том, что в будущем нам может потребоваться изменить   
одну функцию, а не две.   
Заключение  
Конечно, это было всего лишь упражнение. В действительности нет никаких   
причин создавать функции min и max, потому что обе они уже имеются в языке   
Python! Мы встречались с ними в главе 5, когда рассматривали инструменты   
для работы с числами, и затем еще раз в главе 14, когда исследовали итерации.   
Встроенные версии функций работают практически так же, как и наши, но   
они написаны на языке C для получения более высокой скорости работы и при-  
C для получения более высокой скорости работы и при-  
 для получения более высокой скорости работы и при-  
нимают либо единственный аргумент с итерируемым объектом, либо произ-  
вольное множество аргументов. Однако, хотя собственно наш пример можно   
считать избыточным, но прием создания универсальных функций вполне мо-  
жет быть распространен на другие случаи.  
Универсальные функции   
для работы с множествами  
Теперь рассмотрим более полезный пример использования специальных режи-  
мов сопоставления аргументов. В конце главы 16 мы написали функцию, кото-  
рая возвращает пересечение двух последовательностей (она отбирает элемен-  
ты, общие для обеих последовательностей). Ниже приводится версия функции,   
которая возвращает пересечение произвольного числа последовательностей   
(одной или более), где используется механизм передачи произвольного числа   
аргументов в форме \*args для сбора всех передаваемых аргументов в виде кол-

Универсальные функции для работы с множествами   
529  
лекции. Все аргументы передаются в тело функции в составе кортежа, поэтому   
для их обработки можно использовать простой цикл for. Ради интереса мы на-  
пишем функцию union, возвращающую объединение, которая также принима-  
ет произвольное число аргументов и собирает вместе все элементы, имеющиеся   
в любом из операндов:  
def intersect(\*args):  
 res = []  
 for x in args[0]: # Сканировать первую последовательность  
 for other in args[1:]: # Во всех остальных аргументах  
 if x not in other: break # Общий элемент?  
 else: # Нет: прервать цикл  
 res.append(x) # Да: добавить элемент в конец  
 return res  
   
def union(\*args):  
 res = []  
 for seq in args: # Для всех аргументов  
 for x in seq: # Для всех элементов  
 if not x in res:  
 res.append(x) # Добавить новый элемент в результат  
 return res  
Поскольку эти функции могут использоваться многократно (и они слишком   
большие, чтобы вводить их в интерактивной оболочке), мы сохраним их в мо-  
дуле с именем inter2.py (если вы забыли, как выполняется импортирование мо-  
дулей, прочитайте введение в главе 3 или подождите до пятой части книги).   
В обе функции аргументы передаются в виде кортежа args. Как и оригиналь-  
ная версия intersect, обе они работают с любыми типами последовательностей.   
Ниже приводится пример обработки строк, последовательностей разных типов   
и случай обработки более чем двух последовательностей:  
% python  
>>> from inter2 import intersect, union  
>>> s1, s2, s3 = “SPAM”, “SCAM”, “SLAM”  
   
>>> intersect(s1, s2), union(s1, s2) # Два операнда  
([‘S’, ‘A’, ‘M’], [‘S’, ‘P’, ‘A’, ‘M’, ‘C’])  
   
>>> intersect([1,2,3], (1,4)) # Смешивание типов  
[1]  
   
>>> intersect(s1, s2, s3) # Три операнда  
[‘S’, ‘A’, ‘M’]  
   
>>> union(s1, s2, s3)  
[‘S’, ‘P’, ‘A’, ‘M’, ‘C’, ‘L’]  
Следует заметить, что в языке Python появился новый тип дан-  
ных – множества (описывается в главе 5), поэтому, строго гово-  
ря, ни одна из этих функций больше не требуется, – они включе-  
ны в книгу только для демонстрации подходов к программиро-  
ванию функций. Так как Python постоянно улучшается, наблю-  
дается странная тенденция – мои книжные примеры имеют   
свойство устаревать с течением времени!

530   
Глава 18. Аргументы   
Имитация функции print в Python 3.0  
В заключение рассмотрим еще один, последний, пример использования спе-  
циальных режимов сопоставления аргументов. Программный код, который   
приводится здесь, предназначен для использования с интерпретатором версии   
Python 2.6 и ниже (он будет работать и в версии 3.0, но в этом нет никакого   
смысла): в нем используются обе формы представления аргументов (\*args –   
кортеж позиционных аргументов и \*\*args – словарь именованных аргументов)   
для имитации интерфейса функции print в Python 3.  
Как мы узнали в главе 11, в действительности в этом нет никакой необходимо-  
сти, потому что программисты, использующие версию 2.6, всегда могут вклю-  
чить поддержку функции print, появившейся в версии 3.0, выполнив инструк-  
цию импортирования:  
from \_\_future\_\_ import print\_function  
Однако для демонстрации работы механизма сопоставления аргументов я по-  
местил реализацию функции print30 в следующий ниже файл print30.py. Эта   
функция выполняет ту же самую работу, имеет небольшой объем и может ис-  
пользоваться многократно:  
“””  
Имитация большинства особенностей функции print в версии 3.0 для использования   
с интерпретатором версии 2.X  
Сигнатура вызова: print30(\*args, sep=’ ‘, end=’\n’, file=None)  
“””  
import sys  
   
def print30(\*args, \*\*kargs):  
 sep = kargs.get(‘sep’, ‘ ‘) # Именованные аргументы   
 end = kargs.get(‘end’, ‘\n’) # со значениями по умолчанию  
 file = kargs.get(‘file’, sys.stdout)  
 output = ‘’  
 first = True  
 for arg in args:  
 output += (‘’ if first else sep) + str(arg)  
 first = False  
 file.write(output + end)  
Чтобы опробовать функцию, импортируйте ее в другом модуле или в интерак-  
тивном сеансе и попытайтесь использовать ее как функцию print в версии 3.0.   
Ниже приводится испытательный сценарий testprint30.py (обратите внимание,   
что функции присвоено имя «print30», потому что «print» является зарезерви-  
print30», потому что «print» является зарезерви-  
30», потому что «print» является зарезерви-  
print» является зарезерви-  
» является зарезерви-  
рованным словом в версии 2.6):  
from print30 import print30  
print30(1, 2, 3)  
print30(1, 2, 3, sep=’’) # Подавить вывод разделителя  
print30(1, 2, 3, sep=’...’)  
print30(1, [2], (3,), sep=’...’) # Вывод объектов различных типов  
   
print30(4, 5, 6, sep=’’, end=’’) # Подавить вывод символа новой строки  
print30(7, 8, 9)  
print30() # Добавить новую строку  
   
import sys  
print30(1, 2, 3, sep=’??’, end=’.\n’, file=sys.stderr) # Перенаправить в файл

Имитация функции print в Python 3.0   
531  
Если запустить этот сценарий под управлением Python 2.6, мы получим те же   
результаты, что и при использовании функции print в версии 3.0:  
C:\misc> c:\python26\python testprint30.py  
1 2 3  
123  
1...2...3  
1...[2]...(3,)  
4567 8 9  
1??2??3.  
Хотя эта функция и не имеет практической ценности при работе с Python 3.0,   
тем не менее в этой версии она даст те же самые результаты. Как обычно, уни-  
версальная архитектура языка Python позволяет нам создавать модели и ис-  
Python позволяет нам создавать модели и ис-  
 позволяет нам создавать модели и ис-  
следовать концепции самого языка Python. В данном случае реализация ме-  
Python. В данном случае реализация ме-  
. В данном случае реализация ме-  
ханизма сопоставления аргументов на языке Python оказалась настолько же   
гибкой, как и внутренняя реализация.  
Использование аргументов, которые могут   
передаваться только по имени  
Интересно отметить, что этот пример можно было бы реализовать в версии   
Python 3.0 с применением аргументов, которые могут передаваться только по   
имени, о которых рассказывалось выше в этой главе, – для автоматической   
проверки дополнительных параметров настройки:  
# Использование аргументов, которые могут передаваться только по имени  
   
def print30(\*args, sep=’ ‘, end=’\n’, file=sys.stdout):  
 output = ‘’  
 first = True  
 for arg in args:  
 output += (‘’ if first else sep) + str(arg)  
 first = False  
 file.write(output + end)  
Эта версия действует точно так же, как и предыдущая, и может служить от-  
личным примером того, насколько удобными могут быть аргументы, которые   
могут передаваться только по имени. В оригинальной версии функции пред-  
полагается, что все позиционные аргументы являются объектами, которые   
требуется вывести, а все именованные аргументы являются дополнительными   
параметрами настройки. Это почти то, что нужно, но прежняя версия будет   
молча игнорировать все лишние именованные аргументы, тогда как новая вер-  
сия, например, вызовет исключение:  
>>> print30(99, name=’bob’)  
TypeError: print30() got an unexpected keyword argument ‘name’  
Чтобы выявить лишние именованные аргументы вручную, мы могли бы из-  
влекать допустимые параметры с помощью метода dict.pop() и по окончании   
проверять размер словаря. Ниже приводится измененная версия функции для   
Python 2.X, аналогичная по своей функциональности функции, где использу-  
ются аргументы, которые могут передаваться только по имени:  
# Удаляет допустимые именованные аргументы со значениями по умолчанию  
   
def print30(\*args, \*\*kargs):

532   
Глава 18. Аргументы   
 sep = kargs.pop(‘sep’, ‘ ‘)  
 end = kargs.pop(‘end’, ‘\n’)  
 file = kargs.pop(‘file’, sys.stdout)  
 if kargs: raise TypeError(‘extra keywords: %s’ % kargs)  
 output = ‘’  
 first = True  
 for arg in args:  
 output += (‘’ if first else sep) + str(arg)  
 first = False  
 file.write(output + end)  
Эта версия действует точно так же, но на этот раз она обнаруживает недопусти-  
мые именованные аргументы:  
>>> print30(99, name=’bob’)  
TypeError: extra keywords: {‘name’: ‘bob’}  
Эта версия функции работает под управлением Python 2.6, но она на четыре   
строки длиннее, чем версия, где применяются аргументы, которые могут пе-  
редаваться только по именам. К сожалению, от лишнего программного кода   
избавиться не удастся – версия с аргументами, которые могут передаваться   
только по именам, может работать только под управлением Python 3.0, что от-  
меняет причины, побудившие меня написать этот пример (имитация функции   
из версии 3.0, которая способна работать только в версии 3.0, не имеет никакой   
практической ценности!). Однако в программах, предназначенных для работы   
под управлением интерпретатора версии 3.0, аргументы, которые могут пере-  
даваться только по именам, могут упростить реализацию функций, прини-  
мающих аргументы и дополнительные параметры настройки. Еще один при-  
мер использования аргументов, которые могут передаваться только по именам   
в версии 3.0, вы найдете в главе 20.  
Придется держать в уме: именованные аргументы   
Как вы уже наверняка поняли, специальные режимы сопоставления   
аргументов могут быть весьма сложными. Однако они не являются обя-  
зательными – вы можете использовать самый простой режим сопостав-  
ления позиционных аргументов, и это будет, пожалуй, самое лучшее   
решение, пока вы только учитесь. Однако эти режимы используются   
в некоторых инструментах Python, поэтому так важно иметь некоторое   
представление об этих режимах.  
Именованные аргументы играют важную роль в модуле tkinter, кото-  
рый фактически стал стандартным средством для разработки графиче-  
ского интерфейса в языке Python (в Python 2.6 этот модуль называется   
Tkinter). Мы познакомимся с tkinter далее в этой книге, но в качестве   
предварительного знакомства замечу, что при использовании этой би-  
блиотеки для установки значений параметров компонентов графическо-  
го интерфейса используются именованные аргументы. Например, сле-  
дующий вызов:  
from tkinter import \*  
widget = Button(text=”Press me”, command=someFunction)

В заключение   
533  
создает новую кнопку и определяет текст на кнопке и функцию обрат-  
ного вызова с помощью ключевых аргументов text и command. Так как   
графические компоненты могут иметь большое число параметров, име-  
нованные аргументы позволяют указывать только необходимые пара-  
метры. В противном случае пришлось бы перечислять все возможные   
параметры в соответствии с их позициями или надеяться, что аргумен-  
ты со значениями по умолчанию будут правильно интерпретироваться   
во всех возможных ситуациях.  
Многие встроенные функции в языке Python ожидают, что мы будем ис-  
пользовать именованные аргументы для определения режимов работы,   
которые могут иметь, а могут и не иметь значения по умолчанию. Как   
мы узнали в главе 8, например, встроенная функция sorted:   
sorted(iterable, key=None, reverse=False)  
ожидает, что ей будет передан итерируемый объект для сортировки, но   
при этом позволяет передавать ей дополнительные именованные аргу-  
менты, определяющие ключ словаря, по которому будет выполнять-  
ся сортировка, и признак сортировки в обратном порядке, которые по   
умолчанию имеют значения None и False соответственно. Обычно мы не   
используем эти параметры, поэтому они могут быть опущены, чтобы за-  
действовать значения по умолчанию.  
В заключение  
В этой главе мы рассмотрели вторую из двух ключевых концепций, имеющих   
отношение к функциям: аргументы (как объекты передаются в функции). Мы   
узнали, что аргументы передаются функции через операцию присваивания, то   
есть в виде ссылок на объекты, которые в действительности являются указа-  
телями.  
Мы также познакомились с дополнительными особенностями, включая имено-  
ванные аргументы и аргументы со значениями по умолчанию, возможность ис-  
пользовать произвольное количество аргументов и аргументы, которые могут   
передаваться исключительно по именам в версии 3.0. Наконец, мы увидели,   
что изменяемые объекты в аргументах проявляют то же самое поведение, как   
и другие разделяемые ссылки на объекты, – если функции явно не передается   
копия объекта, воздействие непосредственно на изменяемый объект может от-  
разиться на вызывающей программе.  
В следующей главе мы продолжим изучение функций и исследуем более слож-  
ные понятия, связанные с функциями: аннотации функций, lambda-выражения   
и инструменты функционального программирования, такие как функции   
map и filter. Многие из этих концепций исходят из того, что функции в языке   
Python являются обычными объектами и потому поддерживают дополнитель-  
ные, очень гибкие режимы работы. Однако прежде чем углубиться в эти темы,   
изучите контрольные вопросы к этой главе, чтобы закрепить знания об аргу-  
ментах, полученные здесь.

534   
Глава 18. Аргументы   
Закрепление пройденного  
Контрольные вопросы  
1. Что выведет следующий фрагмент и почему?  
>>> def func(a, b=4, c=5):  
... print(a, b, c)  
...  
>>> func(1, 2)  
2. Что выведет следующий фрагмент и почему?  
>>> def func(a, b, c=5):  
... print(a, b, c)  
...  
>>> func(1, c=3, b=2)  
3. Что можно сказать об этом фрагменте – что он выведет и почему?  
>>> def func(a, \*pargs):  
... print(a, pargs)  
...  
>>> func(1, 2, 3)  
4. Что выведет следующий фрагмент и почему?  
>>> def func(a, \*\*kargs):  
... print(a, kargs)  
...  
>>> func(a=1, c=3, b=2)  
5. И наконец, что выведет следующий фрагмент и почему?  
>>> def func(a, b, c=3, d=4): print(a, b, c, d)  
...  
>>> func(1, \*(5,6))  
6. Назовите три способа, которые могут использоваться для передачи резуль-  
татов из функции в вызывающую программу.   
Ответы  
1. В данном случае будет выведена последовательность чисел ‘1 2 5’, потому   
что 1 и 2 передаются в виде позиционных аргументов a и b, а аргумент c опу-  
щен и при вызове функции он получает значение по умолчанию 5.  
2. На этот раз будет выведена последовательность чисел ‘1 2 3’: 1 передается   
в позиционном аргументе a, а значения 2 и 3 передаются в именованных ар-  
гументах b и c (при передаче именованных аргументов порядок их следова-  
ния не имеет значения).  
3. Этот фрагмент выведет последовательность ‘1 (2, 3)’, потому что число 1   
передается в аргументе a, а конструкция \*pargs соберет все остальные пози-  
ционные аргументы в объект кортежа. Мы могли бы выполнить обход кор-  
тежа с дополнительными позиционными аргументами с помощью любого   
инструмента итераций (например, for arg in pargs: ...).  
4. На этот раз будет выведено ‘1, {‘c’: 3, ‘b’: 2}’, потому что число 1 передается   
в именованном аргументе a, а конструкция \*\*kargs соберет все остальные

Закрепление пройденного   
535  
именованные аргументы в словарь. Мы могли бы выполнить обход ключей   
словаря с дополнительными именованными аргументами с помощью любо-  
го инструмента итераций (например, for key in kargs: ...).  
5. Здесь будет выведено “1 5 6 4”: 1 соответствует аргументу в первой позиции, 5   
и 6 соответствуют аргументам b и c в соответствии с формой \*name (значение   
6 переопределяет значение по умолчанию аргумента c) и d получит значение   
по умолчанию 4, потому что четвертый аргумент в вызове функции отсут-  
ствует.  
6. Функции могут возвращать результаты с помощью инструкции return,   
воздействуя на изменяемые объекты, передаваемые в аргументах, а так-  
же изменением глобальных переменных. Вообще глобальные переменные   
использовать для этих целей не рекомендуется (за исключением редких   
случаев, таких как многопоточные программы), потому что это осложняет   
понимание и использование программного кода. Наилучшим способом яв-  
ляется инструкция return, хотя воздействие на изменяемые объекты – тоже   
неплохой вариант при условии, что он предусмотрен заранее. Кроме того,   
функции могут выполнять обмен информацией через такие системные ме-  
ханизмы, как файлы и сокеты, но это уже выходит за рамки данной главы.

Глава 19.  
   
Расширенные возможности функций  
В этой главе будут представлены дополнительные расширенные возможности,   
имеющие отношение к функциям: рекурсивные функции, атрибуты и аннота-  
ции функций, lambda-выражения и средства функционального программиро-  
вания, такие как функции map и filter. Все они относятся к разряду сложных   
механизмов, с которыми, в зависимости от решаемых вами задач, вы можете   
и не встретиться. Тем не менее из-за особой роли этих возможностей в некото-  
рых областях понимание основных принципов их использования будет совсем   
не лишним – lambda-выражения, например, часто используются в реализации   
графического интерфейса.  
Отчасти искусство использования функций лежит в области интерфейсов   
между ними, поэтому здесь мы также исследуем некоторые общие принципы   
проектирования функций. Следующая глава продолжит тему расширенных   
возможностей функций обсуждением функций-генераторов и выражений-  
генераторов и вновь вернется к теме генераторов списков, но уже в контексте   
инструментов функционального программирования, рассматриваемых в дан-  
ной главе.  
Концепции проектирования функций  
Теперь, когда мы уже имеем некоторое представление об основах использо-  
вания функций в языке Python, продолжим изучение темы с нескольких ре-  
Python, продолжим изучение темы с нескольких ре-  
, продолжим изучение темы с нескольких ре-  
комендаций. Когда начинают использоваться функции, возникает проблема   
выбора, как лучше связать элементы между собой, например как разложить   
задачу на функции (связность), как должны взаимодействовать функции (вза-  
имодействие) и так далее. Вы должны учитывать такие особенности, как раз-  
мер функций, потому что от них напрямую зависит удобство сопровождения   
программного кода. Некоторые из них относятся к категории структурного   
анализа и проектирования, но они в равной степени могут применяться и к   
программному коду.   
Некоторые понятия, имеющие отношение к взаимодействию функций и моду-  
лей, были представлены в главе 17, а здесь мы коротко рассмотрим некоторые   
основные правила для тех, кто начинает осваивать язык Python:

Концепции проектирования функций   
537  
 •  
Взаимодействие: для передачи значений функции используйте аргумен-  
ты, для возврата результатов – инструкцию return. Всегда следует стре-  
миться сделать функцию максимально независимой от того, что проис-  
ходит за ее пределами. Аргументы и инструкция return часто являются   
лучшими способами ограничить внешнее воздействие небольшим числом   
известных мест в программном коде.  
 •  
Взаимодействие: используйте глобальные переменные, только если это   
действительно необходимо. Глобальные переменные (то есть имена в объ-  
емлющем модуле) обычно далеко не самый лучший способ организации   
взаимодействий с функциями. Они могут порождать зависимости и пробле-  
мы согласованности, которые существенно осложняют отладку программ.  
 •  
Взаимодействие: не воздействуйте на изменяемые аргументы, если вызы-  
вающая программа не предполагает этого. Функции могут оказывать воз-  
действие на части изменяемых объектов, получаемых в виде аргументов,   
но, как и в случае с глобальными переменными, это предполагает слишком   
тесную связь между вызывающей программой и вызываемой функцией,   
что может сделать функцию слишком специфичной и неустойчивой.  
 •  
Связность: каждая функция должна иметь единственное назначение. Хо-  
рошо спроектированная функция должна решать одну задачу, которую   
можно выразить в одном повествовательном предложении. Если это пред-  
ложение допускает слишком широкое толкование (например: «эта функ-  
ция реализует всю программу целиком») или содержит союзы (например:   
«эта функция дает возможность клиентам составлять и отправлять заказ   
на доставку пиццы»), то стоит подумать над тем, чтобы разбить ее на от-  
дельные и более простые функции. В противном случае окажется невоз-  
можным повторно использовать программный код функции, в котором сме-  
шаны различные действия.  
 •  
Размер: каждая функция должна иметь относительно небольшой размер.   
Это условие естественным образом следует из предыдущего, однако если   
функция начинает занимать несколько экранов – это явный признак, что   
пора подумать о том, чтобы разбить ее. Особенно, если учесть краткость,   
присущую языку Python. Длинная функция с большой глубиной вложен-  
ности часто свидетельствует о промахах в проектировании. Сохраняйте   
функции короткими и простыми.  
 •  
Взаимодействие: избегайте непосредственного изменения переменных   
в другом модуле. Мы рассматривали эту концепцию в главе 17 и еще вер-  
немся к ней в следующей части книги, когда сконцентрируем свое внима-  
ние на модулях. Однако для справки напомню, что непосредственное из-  
менение переменных в других модулях устанавливает тесную зависимость   
между модулями, так же как тесную зависимость устанавливает изменение   
глобальных переменных из функций – модули становятся сложными в по-  
нимании и малопригодными для многократного использования. Всегда,   
когда это возможно, для изменения переменных модуля вместо прямых ин-  
струкций присваивания используйте функции доступа.  
На рис. 19.1 приводится схема организации взаимодействий функций с внеш-  
ним миром – входные данные поступают в функции из элементов слева, а ре-  
зультаты могут возвращаться в любой из форм справа. Опытные программи-  
сты предпочитают использовать для ввода только аргументы, и для вывода –   
только инструкцию return.

538   
Глава 19. Расширенные возможности функций   
Другие функции  
Выходные данные  
Входные данные  
Аргументы  
Глобальные переменные  
Файлы/потоки  
Функция  
Локальные  
переменные  
Инструкция return  
Изменяемые аргументы  
Глобальные переменные  
Файлы/потоки  
Рис. 19.1. Окружение функции времени выполнения. Функция может полу-  
чать входные данные и возвращать результаты различными способами,   
однако функции проще в понимании и сопровождении, когда входные данные   
передаются в виде аргументов, а возврат результатов производится с по-  
мощью инструкции return или посредством воздействия на изменяемые аргу-  
менты, при условии, что последнее предполагается вызывающей программой.   
В Python 3 результаты могут также возвращаться с помощью нелокальных   
переменных, существующих в области видимости объемлющей функции  
Конечно, из приведенных выше правил проектирования есть свои исключе-  
ния, включая те, что связаны с поддержкой ООП в языке Python. Как вы уви-  
дите в шестой части книги, классы в языке Python зависят от изменения пере-  
даваемого изменяемого объекта – функции воздействуют на атрибуты аргу-  
мента self, получаемого автоматически, изменяя информацию о его состоянии   
(например, self.name = ‘bob’). Кроме того, когда нет возможности использовать   
классы, часто наилучший способ сохранения информации о состоянии между   
вызовами функций представляют глобальные переменные в модуле. Побочные   
эффекты опасны, только когда их не ожидают.  
Однако в общем случае необходимо стремиться минимизировать внешние за-  
висимости функций и других компонентов программ. Чем более автономной   
является функция, тем проще будет ее понять, использовать и изменять.  
Рекурсивные функции  
При обсуждении областей видимости, в начале главы 17, мы коротко отмети-  
ли, что язык Python поддерживает рекурсивные функции – функции, которые   
могут вызывать сами себя, прямо или косвенно, образуя цикл. Рекурсия – это   
достаточно сложная тема, и она относительно редко встречается в программах   
на языке Python. Тем не менее это достаточно полезный прием, чтобы знать   
о нем, позволяющий реализовывать обход структур данных с произвольной   
и неизвестной заранее организацией. Рекурсия даже является альтернативой   
простым циклам и итерациям, хотя и не обязательно более простой или более   
производительной.

Рекурсивные функции   
539  
Вычисление суммы с применением рекурсии  
Рассмотрим несколько примеров. Чтобы вычислить сумму чисел в списке   
(или в другой последовательности), можно либо воспользоваться встроенной   
функцией sum, либо написать свою, более специализированную, версию. Ниже   
приводится пример такой специализированной функции вычисления суммы,   
в которой используется прием рекурсии:  
>>> def mysum(L):  
... if not L:  
... return 0  
... else:  
... return L[0] + mysum(L[1:]) # Вызывает себя саму  
   
>>> mysum([1, 2, 3, 4, 5])  
15  
На каждом уровне рекурсии эта функция вызывает саму себя, чтобы получить   
сумму остатка списка, которая складывается с первым элементом. Рекурсив-  
ный цикл заканчивается и возвращается ноль, когда функция получит пустой   
список. Когда рекурсия используется, как показано здесь, на каждом уровне   
рекурсии для функции создается собственная копия локальной области види-  
мости на стеке вызовов – в данном случае это означает, что на каждом уровне   
создается собственная переменная L.  
Если вам трудно понять это (что вполне характерно для начинающих програм-  
мистов), попробуйте добавить в функцию вывод L и запустите пример еще раз,   
чтобы увидеть содержимое списка на каждом уровне рекурсии:  
>>> def mysum(L):  
... print(L) # Поможет отслеживать уровни рекурсии  
... if not L: # На каждом уровне список L будет получаться короче  
... return 0  
... else:  
... return L[0] + mysum(L[1:])  
...  
   
>>> mysum([1, 2, 3, 4, 5])  
[1, 2, 3, 4, 5]  
[2, 3, 4, 5]  
[3, 4, 5]  
[4, 5]  
[5]  
[]  
15  
Как видите, на каждом уровне рекурсии список становится все меньше и мень-  
ше, пока не опустеет, что вызовет конец рекурсивного цикла. Сумма вычисля-  
ется уже в процессе обратного раскручивания рекурсии.  
Альтернативные решения  
Интересно отметить, что мы также можем использовать здесь трехместный   
оператор if/else (описанный в главе 12), чтобы сделать программный код ком-  
пактнее. Кроме того, мы можем обобщить операцию вычисления суммы для   
любых типов данных, поддерживающих операцию сложения (что легко сде-

540   
Глава 19. Расширенные возможности функций   
лать, если предположить, что входная последовательность содержит хотя бы   
один элемент, как это было сделано в главе 18, в примере функции поиска ми-  
нимального значения), и использовать расширенную операцию присваивания   
последовательностей, имеющуюся в Python 3.0, чтобы упростить деление по-  
Python 3.0, чтобы упростить деление по-  
 3.0, чтобы упростить деление по-  
следовательности на части первый/остаток (как описано в главе 11):  
def mysum(L):  
 return 0 if not L else L[0] + mysum(L[1:]) # Трехместный оператор  
   
def mysum(L): # Суммирует любые типы  
 return L[0] if len(L) == 1 else L[0] + mysum(L[1:]) # предполагает наличие   
 # хотя бы одного значения  
   
def mysum(L): # Использует расширенную  
 first, \*rest = L # операцию присваивания  
 return first if not rest else first + mysum(rest) # последовательностей   
 # в Python 3.0   
Последние две функции будут завершаться с ошибкой при получении пустого   
списка, но они позволяют находить сумму последовательностей не только чи-  
сел, но и объектов любых типов, поддерживающих операцию +:  
>>> mysum([1]) # mysum([]) будет завершаться ошибкой в 2 последних функциях  
1  
>>> mysum([1, 2, 3, 4, 5])  
15  
>>> mysum((‘s’, ‘p’, ‘a’, ‘m’)) # Но они могут суммировать данные любых типов  
‘spam’  
>>> mysum([‘spam’, ‘ham’, ‘eggs’])  
‘spamhameggs’  
Если вы внимательно изучите эти три версии, вы обнаружите, что послед-  
ние две могут также работать с однострочным аргументом (например, mysum   
(‘spam’)), потому что строка – это последовательность односимвольных строк.   
Третья версия может работать с любыми итерируемыми объектами, включая   
открытые для чтения файлы, однако другие версии такой возможности не име-  
ют, так как опираются на операцию индексирования. Если в третьей версии   
заголовок функции изменить на def mysum(first, \* rest), она вообще работать не   
будет, потому что в этом случае она будет ожидать два отдельных аргумента,   
а не единственный итерируемый объект.  
Имейте в виду, что рекурсия может быть прямой, как в примерах выше, или   
косвенной, как в следующем примере (когда функция вызывает другую функ-  
цию, которая в свою очередь вызывает функцию, вызвавшую ее). Конечный ре-  
зультат будет тем же самым, только в этом случае на каждом уровне рекурсии   
будет выполняться два вызова функций вместо одного:  
>>> def mysum(L):  
... if not L: return 0  
... return nonempty(L) # Вызов функции, которая вызовет эту функцию  
...  
>>> def nonempty(L):  
... return L[0] + mysum(L[1:]) # Косвенная рекурсия  
...  
>>> mysum([1.1, 2.2, 3.3, 4.4])  
11.0

Рекурсивные функции   
541  
Инструкции циклов вместо рекурсии  
Хотя прием рекурсии вполне может использоваться для вычисления сумм эле-  
ментов последовательностей, как было показано в предыдущем разделе, тем не   
менее для данного случая рекурсия – это слишком тяжеловесный механизм.   
В действительности в языке Python рекурсия используется не так часто, как   
в более необычных языках программирования, таких как Prolog или Lisp,   
потому что в языке Python особое значение придается простым процедурным   
инструкциям, таким как циклы, которые более естественно подходят для ре-  
шения подобных задач. Цикл while, например, часто привносит чуть больше   
конкретики и не требует, чтобы функция была реализована как рекурсивная:  
>>> L = [1, 2, 3, 4, 5]  
>>> sum = 0  
>>> while L:  
... sum += L[0]  
... L = L[1:]  
...  
>>> sum  
15  
Инструкция for дает нам еще больше возможностей – выполняя итерации ав-  
томатически, она в большинстве случаев позволяет избавиться от рекурсии   
(которая обычно менее эффективна с точки зрения использования памяти   
и скорости выполнения):  
>>> L = [1, 2, 3, 4, 5]  
>>> sum = 0  
>>> for x in L: sum += x  
...  
>>> sum  
15  
При использовании инструкций циклов отпадает необходимость создавать для   
каждой итерации копии локальной области видимости в стеке вызовов и лик-  
видируются потери времени, необходимые на вызов функции. (В главе 20 при-  
водится пример измерения скорости выполнения различных вариантов, подоб-  
ных этим.)  
Обработка произвольных структур данных  
С другой стороны, рекурсия (или эквивалентные ей алгоритмы, основанные на   
использовании стека, которые мы рассмотрим здесь) может оказаться востре-  
бованной для реализации обхода структур данных с произвольной организа-  
цией. В качестве простого примера, который поможет оценить роль рекурсии   
в данном контексте, рассмотрим задачу вычисления суммы всех чисел в струк-  
туре, состоящей из вложенных списков, как показано ниже:  
[1, [2, [3, 4], 5], 6, [7, 8]] # Произвольно вложенные списки  
Простые инструкции циклов в этом случае не годятся, потому что выполнить   
обход такой структуры с помощью линейных итераций не удастся. Вложенные   
инструкции циклов также не могут использоваться, потому что списки могут   
иметь произвольные уровни вложенности и иметь произвольную организа-

542   
Глава 19. Расширенные возможности функций   
цию. Вместо этого следующий пример выполняет обход вложенных списков   
с помощью рекурсии:  
def sumtree(L):  
 tot = 0  
 for x in L: # Обход элементов одного уровня  
 if not isinstance(x, list):  
 tot += x # Числа суммируются непосредственно  
 else:  
 tot += sumtree(x) # Списки обрабатываются рекурсивными вызовами  
 return tot  
   
L = [1, [2, [3, 4], 5], 6, [7, 8]] # Произвольная глубина вложения  
print(sumtree(L)) # Выведет 36  
   
# Патологические случаи  
   
print(sumtree([1, [2, [3, [4, [5]]]]])) # Выведет 15 (центр тяжести справа)  
print(sumtree([[[[[1], 2], 3], 4], 5])) # Выведет 15 (центр тяжести слева)  
Проанализируйте два последних вызова, чтобы понять, как эта функция вы-  
полняет обход вложенных списков. Хотя данный пример достаточно искус-  
ственный, тем не менее он является представительным для широкого круга   
программ – деревья наследования и цепочки импортирования модулей, напри-  
мер, могут иметь похожую структуру. Далее в книге мы еще вернемся к этой   
роли рекурсии в более практичных примерах:  
 •  
В главе 24 приводится сценарий reloadall.py, выполняющий обход цепочек   
импортирования  
 •  
В главе 28 приводится сценарий classtree.py, выполняющий обход деревьев   
наследования классов  
 •  
В главе 30 приводится сценарий lister.py, также выполняющий обход дере-  
вьев наследования классов  
Обычно в случае линейных итераций предпочтение должно отдаваться ин-  
струкциям циклов по причине их простоты и эффективности, однако иногда   
рекурсия может оказаться незаменимой, как будет показано в перечисленных   
примерах.  
Кроме того, вы должны знать о возможности возникновения в программах не-  
преднамеренной рекурсии. Как будет показано ниже в этой книге, некоторые   
методы перегрузки операторов в классах, такие как \_\_setattr\_\_ и \_\_getattrib-  
getattrib-  
ute\_\_, при неправильном использовании могут приводить к рекурсии. Рекур-  
сия – это мощный инструмент, но ее лучше использовать только тогда, когда   
без нее невозможно обойтись!  
Функции – это объекты: атрибуты и аннотации  
Функции в языке Python гораздо более гибкие, чем можно было бы предста-  
Python гораздо более гибкие, чем можно было бы предста-  
 гораздо более гибкие, чем можно было бы предста-  
вить. Как мы уже видели выше в этой части книги, функции в языке Python –   
это намного больше, чем блок инструкций для компилятора. Функции в языке   
Python являются полноценными объектами, хранящими в памяти все, чем   
они владеют. Кроме того, они свободно могут передаваться между частями про-

Функции – это объекты: атрибуты и аннотации   
543  
граммы и вызываться косвенно. У них также есть некоторые особенности, ко-  
торые имеют мало общего с вызовами, – атрибуты и аннотации.  
Косвенный вызов функций  
Так как функции в языке Python являются объектами, можно написать такую   
программу, которая будет работать с ними, как с обычными объектами. Объек-  
ты функций могут присваиваться, передаться другим функциям, сохраняться   
в структурах данных и так далее, как если бы они были простыми числами   
или строками. Кроме того, объекты функций поддерживают специальные опе-  
рации: они могут вызываться перечислением аргументов в круглых скобках,   
следующих сразу же за выражением функции. И тем не менее, функции при-  
надлежат к категории обычных объектов.  
Мы встречали уже такие способы использования функций в более ранних при-  
мерах, тем не менее краткий обзор поможет нам лучше понять модель объек-  
тов. Например, в имени, которое используется в инструкции def, нет ничего   
уникального: это всего лишь переменная, которая создается в текущей области   
видимости, как если бы оно стояло слева от знака =. После того как инструкция   
def будет выполнена, имя функции представляет собой всего лишь ссылку на   
объект – ее можно присвоить другим именам и вызывать функцию по любому   
из них (не только по первоначальному имени):  
>>> def echo(message): # Имени echo присваивается объект функции  
... print(message)  
...  
>>> echo(‘Direct call’) # Вызов объекта по оригинальному имени  
Direct call  
   
>>> x = echo # Теперь на эту функцию ссылается еще и имя x  
>>> x(‘Indirect call!’) # Вызов объекта по другому имени добавлением ()  
Indirect call  
Поскольку аргументы передаются путем присваивания объектов, функции   
легко можно передавать другим функциям в виде аргументов. В результате   
вызываемая функция может вызвать переданную ей функцию простым добав-  
лением списка аргументов в круглых скобках:  
>>> def indirect(func, arg):  
... func(arg) # Вызов объекта добавлением ()  
...  
>>> indirect(echo, ‘Argument call!’) # Передача функции в функцию  
Argument call!  
Существует даже возможность наполнять структуры данных функциями, как   
если бы они были простыми числами или строками. В этом нет ничего необыч-  
ного, так как составные типы объектов могут содержать объекты любых типов:  
>>> schedule = [ (echo, ‘Spam!’), (echo, ‘Ham!’) ]  
>>> for (func, arg) in schedule:  
... func(arg) # Вызов функции, сохраненной в контейнере  
...  
Spam!  
Ham!

544   
Глава 19. Расширенные возможности функций   
В этом фрагменте просто выполняется обход списка schedule и производится   
вызов функции echo с одним аргументом (обратите внимание на операцию при-  
сваивания кортежа в заголовке инструкции цикла for, которая была представ-  
лена в главе 13). Как мы уже видели в главе 17, функции могут также созда-  
ваться и возвращаться другими функциями:  
>>> def make(label): # Создает функцию, но не вызывает ее  
... def echo(message):  
... print(label + ‘:’ + message)  
... return echo  
...  
>>> F = make(‘Spam’) # Метка сохраняется во вложенной области видимости  
>>> F(‘Ham!’) # Вызов функции, созданной функцией make  
Spam:Ham!  
>>> F(‘Eggs!’)  
Spam:Eggs!  
Универсальность модели объектов в языке Python и отсутствие необходимости   
объявлять типы переменных обеспечивают функциям невероятную гибкость.  
Интроспекция функций  
Функции являются обычными объектами, поэтому мы можем оперировать   
функциями с помощью привычных инструментов. Функции обладают боль-  
шей гибкостью, чем можно было бы представить. Например, если мы создали   
функцию, мы можем вызвать ее:  
>>> def func(a):  
... b = ‘spam’  
... return b \* a  
...  
>>> func(8)  
‘spamspamspamspamspamspamspamspam’  
Но выражение вызова – это лишь одна из операций, которые могут приме-  
няться к объектам функций. Кроме того, мы можем получить базовый доступ   
к атрибутам функции (следующие результаты были получены в Python 3.0�   
в версии 2.6 результаты будут похожими):  
>>> func.\_\_name\_\_  
‘func’  
>>> dir(func)  
[‘\_\_annotations\_\_’, ‘\_\_call\_\_’, ‘\_\_class\_\_’, ‘\_\_closure\_\_’, ‘\_\_code\_\_’,  
...остальные имена опущены...  
‘\_\_repr\_\_’, ‘\_\_setattr\_\_’, ‘\_\_sizeof\_\_’, ‘\_\_str\_\_’, ‘\_\_subclasshook\_\_’]  
Механизмы интроспекции позволяют нам также исследовать детали реализа-  
ции – каждая функция, например, имеет присоединенный к ней объект с про-  
граммным кодом, из которого можно получить такие сведения, как список ло-  
кальных переменных и аргументов:  
>>> func.\_\_code\_\_  
<code object func at 0x0257C9B0, file “<stdin>”, line 1>  
   
>>> dir(func.\_\_code\_\_)  
[‘\_\_class\_\_’, ‘\_\_delattr\_\_’, ‘\_\_doc\_\_’, ‘\_\_eq\_\_’, ‘\_\_format\_\_’, ‘\_\_ge\_\_’,

Функции – это объекты: атрибуты и аннотации   
545  
...остальные имена опущены...  
‘co\_argcount’, ‘co\_cellvars’, ‘co\_code’, ‘co\_consts’, ‘co\_filename’,  
‘co\_firstlineno’, ‘co\_flags’, ‘co\_freevars’, ‘co\_kwonlyargcount’, ‘co\_lnotab’,  
‘co\_name’, ‘co\_names’, ‘co\_nlocals’, ‘co\_stacksize’, ‘co\_varnames’]  
   
>>> func.\_\_code\_\_.co\_varnames  
(‘a’, ‘b’)  
>>> func.\_\_code\_\_.co\_argcount  
1  
Разработчики специализированных инструментов могут использовать эту ин-  
формацию для организации управления функциями (в действительности, мы   
тоже будем пользоваться этой информацией в главе 38, когда будем осущест-  
влять проверку аргументов функции в декораторах).  
Атрибуты функций   
Перечень атрибутов, которые могут иметь объекты функций, не ограничива-  
ется предопределенными атрибутами, которые были перечислены в преды-  
дущем разделе. Как мы узнали в главе 17, к функциям можно присоединять   
и свои атрибуты:  
>>> func  
<function func at 0x0257C738>  
>>> func.count = 0  
>>> func.count += 1  
>>> func.count  
1  
>>> func.handles = ‘Button-Press’  
>>> func.handles  
‘Button-Press’  
>>> dir(func)  
[‘\_\_annotations\_\_’, ‘\_\_call\_\_’, ‘\_\_class\_\_’, ‘\_\_closure\_\_’, ‘\_\_code\_\_’,  
...остальные имена опущены...  
\_\_str\_\_’, ‘\_\_subclasshook\_\_’, ‘count’, ‘handles’]  
Как было показано в указанной главе, такие атрибуты можно использовать   
для хранения информации о состоянии непосредственно в объекте функции   
и отказаться от использования других приемов, таких как применение гло-  
бальных или нелокальных переменных и классов. В отличие от нелокальных   
переменных, атрибуты функций доступны в любом месте программы, где до-  
ступна сама функция. В некотором смысле атрибуты можно рассматривать,   
как имитацию «статических локальных переменных», имеющихся в других   
языках программирования, – переменных, которые являются локальными   
для функции, но сохраняют свои значения после выхода из функции. Атрибу-  
ты связаны с объектами, а не с областями видимости, но конечный эффект от   
их использования получается тот же.  
Аннотации функций в версии 3.0  
В Python 3.0 (но не в 2.6) имеется также возможность присоединять к объектам   
функций краткое описание (аннотацию) – произвольные данные об аргумен-  
тах функции и о возвращаемом значении. Для создания аннотаций в языке

546   
Глава 19. Расширенные возможности функций   
Python используется специальный синтаксис, но интерпретатор не выполняет   
никаких операций с ними – аннотации совершенно необязательны� если они   
присутствуют, они просто сохраняются в атрибутах \_\_annotations\_\_ объектов   
функций и могут использоваться другими инструментами.  
В предыдущей главе мы познакомились с новой особенностью версии Py-  
Py-  
thon 3.0 – с аргументами, которые могут передаваться только по именам. Ан-  
 3.0 – с аргументами, которые могут передаваться только по именам. Ан-  
нотации еще больше обобщают синтаксис заголовка функции. Взгляните на   
следующую, неаннотированную, функцию, которая принимает три аргумента   
и возвращает результат:  
>>> def func(a, b, c):  
... return a + b + c  
...  
>>> func(1, 2, 3)  
6  
Синтаксически аннотации функций находятся в заголовках инструкций def,   
в виде произвольных выражений, ассоциированных с аргументами и возвра-  
щаемыми значениями. Аннотации для аргументов указываются через двоето-  
чие, сразу после имени аргумента. Для возвращаемого значения – после сим-  
волов ->, вслед за списком аргументов. В следующем примере были добавлены   
аннотации ко всем трем аргументам и возвращаемому значению предыдущей   
функции:  
>>> def func(a: ‘spam’, b: (1, 10), c: float) -> int:  
... return a + b + c  
...  
>>> func(1, 2, 3)  
6  
Вызов аннотированной функции ничем не отличается от вызова обычной функ-  
ции, но если в объявлении функции присутствуют аннотации, интерпретатор   
соберет их в словарь и присоединит его к объекту функции. Имена аргументов   
станут ключами, аннотация возвращаемого значения будет сохранена в ключе   
«return», а значениям ключей этого словаря будут присвоены результаты вы-  
return», а значениям ключей этого словаря будут присвоены результаты вы-  
», а значениям ключей этого словаря будут присвоены результаты вы-  
ражений в аннотациях:  
>>> func.\_\_annotations\_\_  
{‘a’: ‘spam’, ‘c’: <class ‘float’>, ‘b’: (1, 10), ‘return’: <class ‘int’>}  
Поскольку аннотация – это всего лишь объект, присоединенный к другому объ-  
екту, ее легко можно обрабатывать. В следующем примере были аннотированы   
два аргумента из трех, и затем выполнен обход присоединенных аннотаций:  
>>> def func(a: ‘spam’, b, c: 99):  
... return a + b + c  
...  
>>> func(1, 2, 3)  
6  
>>> func.\_\_annotations\_\_  
{‘a’: ‘spam’, ‘c’: 99}  
   
>>> for arg in func.\_\_annotations\_\_:  
... print(arg, ‘=>’, func.\_\_annotations\_\_[arg])

Функции – это объекты: атрибуты и аннотации   
547  
...  
a => spam  
c => 99  
Обратите внимание на два интересных момента. Во-первых, в аннотированных   
аргументах все еще можно указывать значения по умолчанию – аннотация (и   
символ :) находится перед значением по умолчанию (и перед символом =). В сле-  
В сле-  
дующем примере фрагмент a: ‘spam’ = 4 означает, что аргумент a по умолчанию   
получает значение 4 и аннотирован строкой ‘spam’:  
>>> def func(a: ‘spam’ = 4, b: (1, 10) = 5, c: float = 6) -> int:  
... return a + b + c  
...  
>>> func(1, 2, 3)  
6  
>>> func() # 4 + 5 + 6 (все аргументы получают значения по умолчанию)  
15  
>>> func(1, c=10) # 1 + 5 + 10 (именованные аргументы действуют как обычно)  
16  
>>> func.\_\_annotations\_\_  
{‘a’: ‘spam’, ‘c’: <class ‘float’>, ‘b’: (1, 10), ‘return’: <class ‘int’>}  
Во-вторых, обратите внимание, что все пробелы в предыдущем примере явля-  
ются необязательными – вы можете использовать или не использовать пробе-  
лы между компонентами в заголовках функций, однако отказ от использова-  
ния пробелов может ухудшить удобочитаемость программного кода:  
>>> def func(a:’spam’=4, b:(1,10)=5, c:float=6)->int:  
... return a + b + c  
...  
>>> func(1, 2) # 1 + 2 + 6  
9  
>>> func.\_\_annotations\_\_  
{‘a’: ‘spam’, ‘c’: <class ‘float’>, ‘b’: (1, 10), ‘return’: <class ‘int’>}  
Аннотации – это новая особенность, появившаяся в версии 3.0, и исследова-  
ны не все аспекты их практического применения. Однако легко представить   
себе, как аннотации могут использоваться для наложения ограничений на   
типы или значения аргументов, и в крупных системах эта особенность могла   
бы использоваться для регистрации информации об интерфейсе. В главе 38 мы   
познакомимся с некоторыми применениями аннотаций, когда будем рассма-  
тривать их как альтернативу аргументам декораторов функций (более общая   
концепция, когда информация располагается за пределами заголовка функ-  
ции и потому не ограничивается единственной ролью). Как и сам язык Python,   
аннотации – это инструмент, область применения которого ограничивается   
лишь вашим воображением.  
Наконец, обратите внимание, что аннотации могут указываться только в ин-  
струкциях def, – они не могут использоваться в lambda-выражениях, потому что   
синтаксис lambda-выражений уже ограничивает область использования функ-  
ций, определяемых таким способом. На удивление, lambda-выражения – это   
тема нашего следующего раздела.

548   
Глава 19. Расширенные возможности функций   
Анонимные функции: lambda  
Помимо инструкции def в языке Python имеется возможность создавать объ-  
екты функций в форме выражений. Из-за сходства с аналогичной возможно-  
стью в языке LISP она получила название lambda1. Подобно инструкции def   
это выражение создает функцию, которая будет вызываться позднее, но в от-  
личие от инструкции def, выражение возвращает функцию, а не связывает ее   
с именем. Именно поэтому lambda-выражения иногда называют анонимными   
(то есть безымянными) функциями. На практике они часто используются, как   
способ получить встроенную функцию или отложить выполнение фрагмента   
программного кода.  
Основы lambda-выражений  
В общем виде lambda-выражение состоит из ключевого слова lambda, за кото-  
рым следуют один или более аргументов (точно так же, как список аргументов   
в круглых скобках в заголовке инструкции def) и далее, вслед за двоеточием,   
находится выражение:  
lambda argument1, argument2,... argumentN : выражение, использующее аргументы  
В качестве результата lambda-выражения возвращают точно такие же объекты   
функций, которые создаются инструкциями def, но здесь есть несколько раз-  
личий, которые делают lambda-выражения удобными в некоторых специализи-  
рованных случаях:  
 •  
lambda – это выражение, а не инструкция. По этой причине ключевое сло-  
во lambda может появляться там, где синтаксис языка Python не позволя-  
Python не позволя-  
 не позволя-  
ет использовать инструкцию def, – внутри литералов или в вызовах функ-  
ций, например. Кроме того, lambda-выражение возвращает значение (новую   
функцию), которое при желании можно присвоить переменной, в противо-  
вес инструкции def, которая всегда связывает функцию с именем в заголов-  
ке, а не возвращает ее в виде результата.  
 •  
Тело lambda – это не блок инструкций, а единственное выражение. Тело   
lambda-выражения сродни тому, что вы помещаете в инструкцию return вну-  
три определения def, – вы просто вводите результат в виде выражения вме-  
сто его явного возврата. Вследствие этого ограничения lambda-выражения   
менее универсальны, чем инструкция def – в теле lambda-выражения может   
быть реализована только логика, не использующая такие инструкции, как   
if. Такая реализация предусмотрена заранее – она ограничивает возмож-  
ность создания большого числа уровней вложенности программ: lambda-  
выражения предназначены для создания простых функций, а инструкции   
def – для решения более сложных задач.  
1   
Название lambda отпугивает многих программистов, хотя в нем нет ничего страшно-  
го. По всей видимости, такая реакция вызвана самим словом «lambda». Это название   
происходит из языка программирования LISP, в котором это название было заимство-  
вано из лямбда-исчисления – разновидности символической логики. Однако в языке   
Python это просто ключевое слово, которое вводит выражение синтаксически. Если   
не оглядываться на математические истоки, окажется, что lambda-выражения проще,   
чем кажется.

Анонимные функции: lambda   
549  
Если отвлечься от этих различий, def и lambda выполняют одну и ту же работу.   
Например, мы уже видели, как создаются функции с помощью инструкции   
def:  
>>> def func(x, y, z): return x + y + z  
...  
>>> func(2, 3, 4)  
9  
Но того же эффекта можно достигнуть с помощью lambda-выражения, явно   
присвоив результат имени, которое позднее будет использоваться для вызова   
функции:  
>>> f = lambda x, y, z: x + y + z  
>>> f(2, 3, 4)  
9  
Здесь имени f присваивается объект функции, созданный lambda-выражением, –   
инструкция def работает точно так же, но присваивание выполняет автомати-  
чески.  
В lambda-выражениях точно так же можно использовать аргументы со значе-  
ниями по умолчанию:  
>>> x = (lambda a=”fee”, b=”fie”, c=”foe”: a + b + c)  
>>> x(“wee”)  
‘weefiefoe’  
Для lambda-выражений используются те же самые правила поиска переменных   
в областях видимости, что и для вложенных инструкций def. lambda-выражения   
создают локальную область видимости, как и вложенные инструкции def, и ав-  
томатически получают доступ к именам в объемлющих функциях, в модуле   
и во встроенной области видимости (в соответствии с правилом LEGB):  
>>> def knights():  
... title = ‘Sir’  
... action = (lambda x: title + ‘ ‘ + x) # Заголовок в объемлющей def  
... return action # Возвращает функцию  
...  
>>> act = knights()  
>>> act(‘robin’)  
‘Sir robin’  
В этом примере до версии Python 2.2 значение для переменной title передава-  
лось бы в виде значения по умолчанию – если вы забыли, почему, вернитесь   
к главе 17, где рассматривались области видимости.  
Когда можно использовать lambda-выражения?  
Вообще говоря, lambda-выражения очень удобны для создания очень малень-  
ких функций, к тому же они позволяют встраивать определения функций   
в программный код, который их использует. Они не являются предметом пер-  
вой необходимости (вы всегда сможете вместо них использовать инструкции   
def), но они позволяют упростить сценарии, где требуется внедрять небольшие   
фрагменты программного кода.

550   
Глава 19. Расширенные возможности функций   
Например, позднее мы увидим, что функции обратного вызова часто реализу-  
ются в виде lambda-выражений, встроенных непосредственно в список аргумен-  
тов, вместо инструкций def где-то в другом месте в модуле и передаваемых по   
имени (примеры вы найдете во врезке «Придется держать в уме: функции об-  
ратного вызова» ниже в этой главе).  
lambda-выражения также часто используются для создания таблиц переходов,   
которые представляют собой списки или словари действий, выполняемых по   
требованию. Например:  
L = [lambda x: x\*\*2, # Встроенные определения функций  
 lambda x: x\*\*3,   
 lambda x: x\*\*4] # Список из трех функций  
   
for f in L:  
 print(f(2)) # Выведет 4, 8, 16  
   
print(L[0](3)) # Выведет 9  
lambda-выражения наиболее полезны в качестве сокращенного варианта ин-  
струкции def, когда необходимо вставить маленькие фрагменты исполняемо-  
го программного кода туда, где использование инструкций недопустимо. На-  
пример, этот фрагмент программного кода создает список из трех функций,   
встраивая lambda-выражения в литерал списка. Инструкция def не может быть   
вставлена в литерал, потому что это – инструкция, а не выражение. Для реа-  
лизации эквивалентной таблицы переходов с применением инструкций def   
потребовалось бы создать именованные функции за пределами контекста их   
использования:  
def f1(x): return x \*\* 2  
def f2(x): return x \*\* 3 # Определения именованных функций  
def f3(x): return x \*\* 4  
   
L = [f1, f2, f3] # Ссылка по имени  
   
for f in L:  
 print(f(2)) # Выведет 4, 8, 16  
   
print(L[0](3)) # Выведет 9  
В действительности, подобные таблицы действий в языке Python можно созда-  
Python можно созда-  
 можно созда-  
вать с помощью словарей и других структур данных. Ниже приводится другой   
пример, выполненный в интерактивном сеансе:  
>>> key = ‘got’  
>>> {‘already’: (lambda: 2 + 2),  
... ‘got’: (lambda: 2 \* 4),  
... ‘one’: (lambda: 2 \*\* 6)}[key]()  
8  
В данном случае, когда интерпретатор создает словарь, каждое из вложенных   
lambda-выражений генерирует и оставляет после себя функцию для последую-  
щего использования – обращение по ключу извлекает одну из этих функций,   
а круглые скобки обеспечивают вызов извлеченной функции. При таком под-  
ходе словарь превращается в более универсальное средство множественного   
выбора, чем то, что я смог реализовать на основе инструкции if в главе 12.

Анонимные функции: lambda   
551  
Чтобы реализовать то же самое без использования lambda-выражений, при-  
шлось бы написать три отдельные инструкции def за пределами словаря, в ко-  
тором эти функции используются, и ссылаться на функции по их именам:  
>>> def f1(): return 2 + 2  
...  
>>> def f2(): return 2 \* 4  
...  
>>> def f3(): return 2 \*\* 6  
...  
>>> key = ‘one’  
>>> {‘already’: f1, ‘got’: f2, ‘one’: f3}[key]()  
64  
Этот прием тоже будет работать, но ведь инструкции def могут располагать-  
ся в файле модуля достаточно далеко, несмотря на то, что они очень корот-  
кие. Близость программного кода, которую обеспечивают lambda-выражения,   
особенно полезна, когда функции используются в единственном месте – если   
три функции в этом фрагменте не используются где-то еще, определенно име-  
ет смысл встроить их в определение словаря в виде lambda-выражений. Кроме   
того, инструкции def требуют даже для маленьких функций указывать имена,   
а они могут вступить в конфликт с другими именами в файле модуля (возмож-  
но, хотя и маловероятно).  
lambda-выражения также очень удобно использовать в списках аргументов   
функций – для определения временных функций, которые больше нигде в про-  
грамме не используются, – мы увидим примеры такого использования ниже,   
в этой главе, когда будем изучать функцию map.  
Как (не) запутать программный код на языке Python  
Тот факт, что lambda должно быть единственным выражением (а не серией ин-  
струкций), казалось бы, устанавливает серьезное ограничение на объем логи-  
ки, которую можно упаковать в lambda-выражение. Однако если вы понимае-  
те, что делаете, большую часть инструкций языка Python можно представить   
в форме выражений.  
Например, представим, что необходимо вывести некоторую информацию   
из тела lambda-выражения, тогда достаточно просто записать sys.stdout.  
write(str(x)+’\n’) вместо print(x) (в главе 11 объяснялось, что это именно то дей-  
ствие, которое выполняет инструкция print). Точно так же в lambda-выражение   
можно заложить логику в виде трехместного выражения if/else, представ-  
ленного в главе 12, или использовать эквивалентную, хотя и более сложную   
комбинацию операторов and/or, описанную там же. Как уже говорилось ранее,   
следующую инструкцию:  
if a:  
 b  
else:  
 c  
можно представить в виде одного из следующих примерно эквивалентных вы-  
ражений:

552   
Глава 19. Расширенные возможности функций   
b if a else c  
((a and b) or c)  
Так как выражения, подобные этим, допустимо помещать внутрь lambda-  
выражения, они могут использоваться для реализации логики выбора внутри   
lambda-функций:  
>>> lower = (lambda x, y: x if x < y else y)  
>>> lower(‘bb’, ‘aa’)  
‘aa’  
>>> lower(‘aa’, ‘bb’)  
‘aa’  
Кроме того, если внутри lambda-выражения потребуется выполнять циклы, их   
можно заменить вызовами функции map и генераторами списков (с ними мы   
уже познакомились в предыдущих главах и вернемся к ним еще раз в следую-  
щей главе):  
>>> import sys  
>>> showall = lambda x: list(map(sys.stdout.write, x)) # Функция list   
 # необходима в 3.0  
   
>>> t = showall([‘spam\n’, ‘toast\n’, ‘eggs\n’])  
spam  
toast  
eggs  
   
>>> showall = lambda x: [sys.stdout.write(line) for line in x]  
   
>>> t = showall((‘bright\n’, ‘side\n’, ‘of\n’, ‘life\n’))  
bright  
side  
of  
life  
Теперь, когда я продемонстрировал вам некоторые уловки, я должен просить   
вас использовать их только в случае крайней необходимости. Без должной   
осторожности они могут сделать программный код нечитабельным (запутан-  
ным). Вообще, простое лучше сложного, явное лучше неявного, а понятные ин-  
струкции лучше заумных выражений. Именно поэтому lambda ограничивается   
выражениями. Если необходимо реализовать сложную логику, используйте   
инструкцию def – lambda-выражения должны использоваться только для не-  
больших фрагментов программного кода. С другой стороны, при умеренном   
использовании эти приемы могут быть полезны.  
Вложенные lambda-выражения и области видимости  
lambda-выражения чаще других используют возможность поиска в области ви-  
димости вложенной функции (символ E в названии правила LEGB, с которым   
мы познакомились в главе 17). Например, следующее ниже lambda-выражение   
находится внутри инструкции def – типичный случай – и потому получает   
значение имени x из области видимости объемлющей функции, имевшееся на   
момент ее вызова:  
>>> def action(x):  
... return (lambda y: x + y) # Создать и вернуть ф-цию, запомнить x  
...

Анонимные функции: lambda   
553  
>>> act = action(99)  
>>> act  
<function <lambda> at 0x00A16A88>  
>>> act(2) # Вызвать функцию, созданную ф-цией action  
101  
Ранее, когда обсуждались области видимости вложенных функций, не говори-  
лось о том, что lambda-выражения обладают доступом к именам во всех объем-  
лющих lambda-выражениях. Это сложно себе вообразить, но представьте, что   
мы записали предыдущую инструкцию def в виде lambda-выражения:  
>>> action = (lambda x: (lambda y: x + y))  
>>> act = action(99)  
>>> act(3)  
102  
>>> ((lambda x: (lambda y: x + y))(99))(4)  
103  
Эта структура lambda-выражений создает функцию, которая при вызове соз-  
дает другую функцию. В обоих случаях вложенное lambda-выражение имеет   
доступ к переменной x в объемлющем lambda-выражении. Этот фрагмент будет   
работать, но программный код выглядит весьма замысловато, поэтому в инте-  
ресах соблюдения удобочитаемости лучше избегать использования вложенных   
друг в друга lambda-выражений.  
Придется держать в уме: функции обратного вызова  
Другое распространенное применение lambda-выражений состоит в опре-  
делении функций обратного вызова для tkinter GUI API (в Python 2.6   
этот модуль называется Tkinter). Например, следующий фрагмент со-  
здает кнопку, по нажатию которой на консоль выводится сообщение�   
при этом предполагается, что модуль tkinter доступен на вашем ком-  
пьютере (в �indows и в других операционных системах он доступен по   
умолчанию):  
import sys  
from tkinter import Button, mainloop # Tkinter в Python 2.6  
x = Button(  
 text =’Press me’,  
 command=(lambda:sys.stdout.write(‘Spam\n’)))  
x.pack()  
mainloop()  
Здесь в качестве обработчика события регистрируется функция, сгене-  
рированная lambda-выражением в аргументе command. Преимущество lambda-  
выражения перед инструкцией def в данном случае состоит в том, что   
обработчик события нажатия на кнопку находится прямо здесь же,   
в вызове функции, создающей эту кнопку.  
В действительности lambda-выражение откладывает исполнение обра-  
ботчика до того момента, пока не произойдет событие: вызов метода write   
произойдет, когда кнопка будет нажата, а не когда она будет создана.

554   
Глава 19. Расширенные возможности функций   
Поскольку правила областей видимости вложенных функций применя-  
ются и к lambda-выражениям, их проще использовать в качестве функ-  
ций обратного вызова. Начиная с версии Python 2.2, они автоматически   
получают доступ к переменным объемлющих функций и в большинстве   
случаев не требуют передачи параметров со значениями по умолчанию.   
Это особенно удобно при обращении к специальному аргументу экзем-  
пляра self, который является локальной переменной в объемлющих   
методах классов (подробнее о классах рассказывается в шестой части   
книги):  
class MyGui:  
 def makewidgets(self):  
 Button(command=(lambda: self.onPress(“spam”)))  
 def onPress(self, message):  
 ...использовать текст сообщения...  
В предыдущих версиях даже self приходилось передавать в виде аргу-  
мента со значением по умолчанию.  
Отображение функций   
на последовательности: map  
Одна из наиболее часто встречающихся задач, которые решаются в програм-  
мах, состоит в применении некоторой операции к каждому элементу в списке   
или в другой последовательности и сборе полученных результатов. Например,   
обновление всех счетчиков в списке может быть выполнено с помощью просто-  
го цикла for:  
>>> counters = [1, 2, 3, 4]  
>>>  
>>> updated = []  
>>> for x in counters:  
... updated.append(x + 10) # Прибавить 10 к каждому элементу  
...  
>>> updated  
[11, 12, 13, 14]  
Но так как такие операции встречаются достаточно часто, язык Python предо-  
Python предо-  
 предо-  
ставляет встроенную функцию, которая выполняет большую часть этой ра-  
боты. Функция map применяет указанную функцию к каждому элементу по-  
следовательности и возвращает список, содержащий результаты всех вызовов   
функции. Например:  
>>> def inc(x): return x + 10 # Функция, которая должна быть вызвана  
...  
>>> map(inc, counters) # Сбор результатов  
[11, 12, 13, 14]  
Функция map была представлена в главах 13 и 14 как средство применения   
встроенной функции к элементам в итерируемом объекте. Здесь мы будем пере-  
давать пользовательскую функцию, которая будет применяться ко всем эле-  
ментам списка – функция map вызывает функцию inc для каждого элемента

Отображение функций на последовательности: map   
555  
списка и собирает полученные результаты в новый список. Не забывайте, что   
функция map в Python 3.0 возвращает итерируемый объект, поэтому для выво-  
Python 3.0 возвращает итерируемый объект, поэтому для выво-  
 3.0 возвращает итерируемый объект, поэтому для выво-  
да всех результатов в интерактивной оболочке мы используем функцию list�   
этого не требуется в Python 2.6.  
Функция map ожидает получить в первом аргументе функцию, поэтому здесь   
часто можно встретить lambda-выражения:  
>>> list(map((lambda x: x + 3), counters)) # Выражение-функция  
[4, 5, 6, 7]  
В данном случае функция прибавляет число 3 к каждому элементу списка   
counters, а так как эта функция нигде в другом месте больше не используется,   
она оформлена в виде lambda-выражения. Такой вариант использования функ-  
ции map представляет собой эквивалент цикла for, поэтому, написав несколько   
строк, эту утилиту в общем виде вы можете реализовать самостоятельно:  
>>> def mymap(func, seq):  
... res = []  
... for x in seq: res.append(func(x))  
... return res  
Предположим, что функция inc осталась прежней, как было показано выше,   
тогда мы можем отобразить ее на последовательность с помощью встроенной   
функции map или нашей версии mymap:  
>>> list(map(inc, [1, 2, 3])) # Встроенная функция map возвращает итератор  
[11, 12, 13]  
>>> mymap(inc, [1, 2, 3]) # Наша функция возвращает список (см. генераторы)  
[11, 12, 13]  
Однако функция map является встроенной функцией, поэтому она доступ-  
на всегда, всегда работает одним и тем же способом и обладает некоторыми   
преимуществами производительности (как мы узнаем в следующей главе,   
она выполняется быстрее, чем любой цикл for). Кроме того, функция map мо-  
жет использоваться в более сложных ситуациях, чем показано здесь. Напри-  
мер, в данном случае имеется несколько аргументов с последовательностями,   
а функция map извлекает их параллельно и передает как отдельные аргументы   
в функцию:  
>>> pow(3, 4) # 3 \*\* 4  
81  
>>> map(pow, [1, 2, 3], [2, 3, 4]) # 1\*\*2, 2\*\*3, 3\*\*4 # 1\*\*2, 2\*\*3, 3\*\*4  
[1, 8, 81]  
При передаче нескольких последовательностей функция map предполагает, что   
ей будет передана функция, принимающая N аргументов для N последователь-  
ностей. Здесь функция pow при каждом вызове принимает от функции map два   
аргумента – по одному из каждой последовательности. Мы могли бы реализо-  
вать свою собственную функцию, имитирующую это действие, но вполне оче-  
видно, что в этом нет никакой необходимости, так как имеется высокопроизво-  
дительная встроенная функция.  
Этот вызов функции map напоминает генераторы списков, которые рассматри-  
вались в главе 14, и с которыми мы встретимся еще раз в следующей главе.   
Основное отличие состоит в том, что map применяет к каждому элементу после-  
довательности не произвольное выражение, а функцию. Вследствие этого огра-

556   
Глава 19. Расширенные возможности функций   
ничения она обладает меньшей гибкостью. Однако, современная реализация   
map в некоторых случаях обладает более высокой производительностью, чем ге-  
нераторы списков (например, когда отображается встроенная функция), и ис-  
пользовать ее проще.   
Средства функционального программирования:   
filter и reduce  
Функция map – это простейший представитель класса встроенных функций   
в языке Python, используемых в функциональном программировании, то есть   
функций, которые применяют другие функции к последовательностям и к   
другим итерируемым объектам. Родственные ей функции отфильтровывают   
элементы с помощью функций, выполняющих проверку (filter), и применяют   
функции к парам элементов, накапливая результаты (reduce). Например, сле-  
дующий вызов функции filter отбирает элементы последовательности больше   
нуля:  
>>> list(range(-5, 5)) # Итератор в Python 3.0  
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]  
   
>>> filter((lambda x: x > 0), range(-5, 5)) # Итератор в Python 3.0  
[1, 2, 3, 4]  
Элементы последовательности, для которых применяемая функция возвраща-  
ет истину, добавляются в список результатов. Как и map, функция filter явля-  
ется примерным эквивалентом цикла for, только она – встроенная функция   
и обладает высокой скоростью выполнения:  
>>> res = [ ]  
>>> for x in range(-5, 5):  
... if x > 0:  
... res.append(x)  
...  
>>> res  
[1, 2, 3, 4]  
Функция reduce в Python 2.6 была простой встроенной функцией, но в вер-  
Python 2.6 была простой встроенной функцией, но в вер-  
 2.6 была простой встроенной функцией, но в вер-  
сии 3.0 она была перемещена в модуль functools и стала более сложной. Она   
принимает итератор, но сама возвращает не итератор, а одиночный объект.   
Ниже приводятся два вызова функции reduce, которые вычисляют сумму   
и произведение элементов списка:  
>>> from functools import reduce # В 3.0 требуется выполнить импортирование  
   
>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])  
10  
>>> reduce((lambda x, y: x \* y), [1, 2, 3, 4])  
24  
На каждом шаге функция reduce передает текущую сумму или произведение   
вместе со следующим элементом списка lambda-функции. По умолчанию пер-  
вый элемент последовательности принимается в качестве начального значе-  
ния. Ниже приводится цикл for, эквивалентный первому вызову, с жестко за-  
данной операцией сложения внутри цикла:

В заключение   
557  
>>> L = [1,2,3,4]  
>>> res = L[0]  
>>> for x in L[1:]:  
... res = res + x  
...  
>>> res  
10  
Написать свою версию функции reduce достаточно просто. Следующая функ-  
ция имитирует большинство особенностей встроенной функции и помогает по-  
нять принцип ее действия:  
>>> def myreduce(function, sequence):  
... tally = sequence[0]  
... for next in sequence[1:]:  
... tally = function(tally, next)  
... return tally  
...  
>>> myreduce((lambda x, y: x + y), [1, 2, 3, 4, 5])  
15  
>>> myreduce((lambda x, y: x \* y), [1, 2, 3, 4, 5])  
120  
Кроме того, встроенная функция reduce может принимать третий необязатель-  
ный аргумент, который используется в качестве начального значения и слу-  
жит значением по умолчанию, когда передаваемая последовательность не со-  
держит ни одного элемента, но я оставлю тестирование этой особенности вам   
в качестве упражнения.  
Если этот пример разжег ваш интерес, загляните также во встроенный модуль   
operator, содержащий функции, соответствующие встроенным выражениям,   
которые могут пригодиться при использовании некоторых функциональных   
инструментов (за дополнительными подробностями об этом модуле обращай-  
тесь к руководству по стандартной библиотеке Python):  
>>> import operator, functools  
>>> functools.reduce(operator.add, [2, 4, 6]) # Оператор сложения в виде ф-ции  
12  
>>> functools.reduce((lambda x, y: x + y), [2, 4, 6])  
12  
Вместе с функцией map, filter и reduce поддерживают мощные приемы функ-  
ционального программирования. Некоторые программисты могут также до-  
полнить комплект средств функционального программирования языка Python   
lambda-выражениями, рассматривавшимися выше, и генераторами списков,   
которые рассматриваются в следующей главе.  
В заключение  
В этой главе мы рассмотрели расширенные понятия, связанные с функциями:   
рекурсивные функции� аннотации функций� lambda-выражения� инструменты   
функционального программирования, такие как map, filter и reduce� и общие   
правила проектирования функций. В следующей главе мы продолжим обсуж-  
дение расширенных тем, связанных с функциями, и рассмотрим функции-  
генераторы и выражения-генераторы, итераторы и генераторы списков, – эти

558   
Глава 19. Расширенные возможности функций   
инструменты так же связаны с функциональным программированием, как и с   
инструкциями циклов. Но прежде чем двинуться дальше, проверьте, насколь-  
ко вы овладели основами концепциями, представленными здесь, ответив на   
контрольные вопросы к главе.  
Закрепление пройденного  
Контрольные вопросы  
1. Как связаны между собой lambda-выражение и инструкция def?   
2. Какие замечания к использованию lambda-выражений вы можете сделать?  
3. Как узнать, является ли функция функцией-генератором?  
4. Сравните функции map, filter и reduce. В чем их сходства и различия?  
5. Что такое аннотации функций и для чего они используются?  
6. Что такое рекурсивные функций и для чего они используются?  
7. Назовите наиболее общие рекомендации по проектированию функций.  
Ответы  
1. И lambda-выражения, и инструкция def создают объекты функций для по-  
следующего вызова. Однако lambda-выражения – это именно выражения,   
и поэтому они могут использоваться для вложения определений функций   
там, где инструкция def синтаксически недопустима. Синтаксически lamb-  
da позволяет возвращать значение единственного выражения� так как эта   
конструкция не поддерживает блоки инструкций, она плохо подходит для   
создания больших функций.  
2. lambda-выражения позволяют «встраивать» небольшие фрагменты про-  
граммного кода, откладывая их выполнение. Они способны работать с пе-  
ременными в объемлющей области видимости и позволяют определять   
значения по умолчанию для аргументов. Нет таких случаев, когда нельзя   
было бы обойтись без lambda-выражений, – всегда можно определить иден-  
тичную инструкцию def и ссылаться на функцию по имени. Однако lambda-  
выражения удобно использовать для встраивания небольших фрагментов   
программного кода, которые в программе больше нигде не используются.   
Их часто можно встретить в программах, широко использующих функции   
обратного вызова, например в приложениях с графическим интерфейсом,   
и они по своей природе близки к функциональным инструментам, таким   
как функции map и filter, принимающим функции обработки.  
3. Все эти три функции применяют другую функцию к элементам последова-  
тельности (итерируемого объекта) и собирают результаты. Функция map пе-  
редает указанной функции каждый элемент исходной последовательности   
и собирает все результаты� функция filter отбирает только элементы, для   
которых указанная функция возвращает значение True� и функция reduce   
вычисляет единственное значение, применяя указанную функцию к нако-  
пленному значению и последующим элементам последовательности. Функ-  
ция reduce, в отличие от двух других, в версии 3.0 находится не во встроен-  
ной области видимости, а в модуле functools.

Закрепление пройденного   
559  
4. Аннотации функций, доступные в версии 3.0 и выше, являются синтак-  
сическим украшением аргументов и результатов функций. Аннотации со-  
бираются в словаре, который сохраняется в атрибуте \_\_annotations\_\_ функ-  
ции. Интерпретатор никак не использует эти аннотации, он просто подго-  
тавливает их для возможного использования другими инструментами.  
5. Рекурсивными называются функции, которые сами вызывают себя прямо   
или косвенно. Они могут использоваться для обхода структур данных про-  
извольной формы, но они также могут использоваться и для реализации   
итераций (хотя итерации зачастую более просто и эффективно реализуются   
с помощью инструкций циклов).  
6. Вообще функции должны иметь небольшой размер, они должны быть мак-  
симально автономными, решать единственную задачу и взаимодействовать   
с другими компонентами посредством входных аргументов и возвращае-  
мых значений. Кроме того, они могут использовать изменяемые объекты   
в аргументах, чтобы вернуть результат, если такое изменение ожидаемо.   
Некоторые типы программ подразумевают дополнительные механизмы   
взаимодействий.

Глава 20.  
   
Итераторы и генераторы  
Эта глава продолжает обсуждение расширенных тем, связанных с функциями.   
Здесь мы поближе познакомимся с концепциями генераторов и итераторов,   
представленными в главе 14. Генераторы списков, будучи по своей природе ци-  
клами for, очень тесно связаны с инструментами функционального программи-  
рования (такими как функции map и filter), поэтому мы снова вернемся к ним   
в этой главе. Кроме того, мы еще раз вернемся к итераторам и познакомимся   
с функциями-генераторами и родственными им выражениями-генераторами –   
средствами воспроизведения результатов по требованию.  
Итерации в языке Python также могут поддерживаться пользовательскими   
классами, но мы отложим обсуждение этой темы до шестой части книги, когда   
мы будем изучать прием перегрузки операторов. Так как эта глава завершает   
обсуждение встроенных инструментов итераций, мы коротко вспомним раз-  
личные инструменты, с которыми уже встречались выше, и сравним произво-  
дительность некоторых из них. Наконец, так как это последняя глава в данной   
части, ее завершает раздел с описанием типичных ошибок и упражнения, ко-  
торые помогут вам приступить к использованию отраженных здесь идей.  
Еще раз о генераторах списков:   
функциональные инструменты  
В предыдущей главе мы изучили такие инструменты функционального про-  
граммирования, как функции map и filter, выполняющие отображение опера-  
ций на последовательности и сбор результатов. Это стало настолько распро-  
страненной задачей в программировании на языке Python, что в нем появилась   
новая особенность – генераторы списков, которые упрощают решение задач   
еще больше, чем только что рассмотренные функции. Проще говоря, генера-  
торы списков применяют к элементам итерируемых объектов произвольные   
выражения – вместо применения функций. Кроме того, они могут быть более   
универсальными инструментами.  
Мы уже встречались с генераторами списков в главе 14 при изучении инструк-  
ций циклов. Но так как они относятся к средствам функционального програм-

Еще раз о генераторах списков: функциональные инструменты   
561  
мирования, таким как функции map и filter, здесь мы вернемся к этой теме еще   
раз. С технической точки зрения эта особенность не привязана к функциям�   
как мы увидим, генераторы списков – более универсальные инструменты, чем   
map и filter, но иногда их проще понять, проводя аналогии с альтернативами,   
основанными на применении функций.  
Генераторы списков и функция map  
Рассмотрим несколько примеров, демонстрирующих самые основы. Как было   
показано в главе 7, встроенная функция ord в языке Python возвращает цело-  
Python возвращает цело-  
 возвращает цело-  
численный код ASCII единственного символа (обратной к ней является встро-  
ASCII единственного символа (обратной к ней является встро-  
 единственного символа (обратной к ней является встро-  
енная функция chr – она возвращает символ, соответствующий коду ASCII):  
>>> ord(‘s’)  
115  
Теперь предположим, что нам необходимо получить коды ASCII всех символов   
в строке. Пожалуй, самый простой подход заключается в использовании цик-  
ла for, в котором полученные результаты добавляются в список:  
>>> res = []  
>>> for x in ‘spam’:  
... res.append(ord(x))  
...  
>>> res  
[115, 112, 97, 109]  
Однако теперь, когда мы уже познакомились с функцией map, тех же результа-  
тов мы можем достичь с помощью единственного вызова функции без необхо-  
димости заботиться о создании и заполнении списка:  
>>> res = map(ord, ‘spam’) # Применить функцию к последовательности  
>>> res  
[115, 112, 97, 109]  
Но те же результаты можно получить с помощью генератора списка – если map   
отображает функцию на последовательность, то генератор списков отображает   
на последовательность выражение:  
>>> res = [ord(x) for x in ‘spam’] # Применит выражение к последовательности  
>>> res  
[115, 112, 97, 109]  
Генераторы списков собирают результаты применения произвольного выраже-  
ния к элементам последовательности и возвращают их в виде нового списка.   
Синтаксически генераторы списков заключаются в квадратные скобки (чтобы   
показать, что они конструируют списки). В простейшем виде генератор списков   
представляет собой выражение, оперирующее переменной, за которым следует   
конструкция, напоминающая заголовок цикла for, в котором используется та   
же переменная. Во время выполнения интерпретатор Python собирает резуль-  
Python собирает резуль-  
 собирает резуль-  
таты выражения для каждой итерации подразумеваемого списка.  
Предыдущий пример дает тот же результат, что цикл for и вызов функции map   
выше. Однако генераторы списков более удобны, особенно, когда требуется   
применить к последовательности произвольное выражение:  
>>> [x \*\* 2 for x in range(10)]  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

562   
Глава 20. Итераторы и генераторы   
Здесь создается список квадратов чисел от 0 до 9 (здесь мы позволили инте-  
рактивной оболочке автоматически вывести список – если вам необходимо   
сохранить список, присвойте его переменной). Чтобы выполнить аналогич-  
ные действия с помощью функции map, потребовалось бы написать отдельную   
функцию, реализующую операцию возведения в квадрат. Так как эта функ-  
ция нам не потребуется в другом месте программы, ее можно было бы (хотя это   
и не обязательно) реализовать не с помощью инструкции def, а в виде lambda-  
выражения:  
>>> map((lambda x: x \*\* 2), range(10))  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  
Этот вызов выполняет ту же работу, и он всего на несколько символов длиннее   
эквивалентной реализации на базе генератора списка. Кроме того, он ненамно-  
го сложнее (по крайней мере, для тех, кто разбирается в lambda-выражениях).   
Однако в случае более сложных выражений генераторы списков часто выгля-  
дят проще. В следующем разделе будет показано – почему.  
Добавление проверок и вложенных циклов:   
функция filter  
Генераторы списков обладают даже еще большей гибкостью, чем было показа-  
но до сих пор. Например, как мы узнали в главе 14, после цикла for можно до-  
бавить оператор if для реализации логики выбора. Генераторы списков с опе-  
ратором if можно представить как аналог встроенной функции filter, пред-  
ставленной в предыдущем разделе, – они пропускают элементы, для которых   
условное выражение в операторе if возвращает ложь.   
Ниже приводятся две версии реализации выбора четных чисел в диапазоне от   
0 до 4 – с помощью генератора списка и с помощью функции filter, которая   
использует небольшое lambda-выражение для выполнения проверки. Для срав-  
нения здесь также показана реализация на основе цикла for:  
>>> [x for x in range(5) if x % 2 == 0]  
[0, 2, 4]  
   
>>> list(filter((lambda x: x % 2 == 0), range(5)))  
[0, 2, 4]  
   
>>> res = []  
>>> for x in range(5):  
... if x % 2 == 0:  
... res.append(x)  
...  
>>> res  
[0, 2, 4]  
Во всех этих версиях используется оператор деления по модулю (остаток от   
деления) %, с помощью которого определяются четные числа: если остаток от   
деления на два равен нулю, следовательно, число четное. Вызов функции fil-  
ter здесь также выглядит ненамного длиннее, чем генератор списка. Однако   
генераторы списков дают возможность объединять оператор if и произвольные   
выражения, позволяя добиться эффекта действия функций filter и map в един-  
ственном выражении:

Еще раз о генераторах списков: функциональные инструменты   
563  
>>> [x \*\* 2 for x in range(10) if x % 2 == 0]  
[0, 4, 16, 36, 64]  
На этот раз создается список квадратов четных чисел в диапазоне от 0 до 9:   
цикл for пропускает числа, для которых условное выражение, присоединен-  
ное справа, возвращает ложь, а выражение слева вычисляет квадраты. Эк-  
вивалентный вызов функции map потребовал бы от нас больше работы – нам   
пришлось бы объединить выбор элементов с помощью функции filter и обход   
списка с помощью map, что в результате дает более сложное выражение:  
>>> list(map((lambda x: x\*\*2), filter((lambda x: x % 2 == 0), range(10))))  
[0, 4, 16, 36, 64]  
В действительности, генераторы списков обладают еще большей гибкостью.   
Они дают возможность запрограммировать любое число вложенных циклов   
for, каждый из которых может сопровождаться собственным оператором if   
с условным выражением. В общем виде генераторы списков выглядят следую-  
щим образом:  
[ expression for target1 in sequence1 [if condition]  
 for target2 in sequence2 [if condition] ...  
 for targetN in sequenceN [if condition] ]  
Вложенные операторы for в генераторах списков действуют точно так же, как   
вложенные инструкции for. Например, следующий фрагмент:  
>>> res = [x + y for x in [0, 1, 2] for y in [100, 200, 300]]  
>>> res  
[100, 200, 300, 101, 201, 301, 102, 202, 302]  
дает тот же результат, что и более объемный эквивалент:  
>>> res = []  
>>> for x in [0, 1, 2]:  
... for y in [100, 200, 300]:  
... res.append(x + y)  
...  
>>> res  
[100, 200, 300, 101, 201, 301, 102, 202, 302]  
Генераторы списков конструируют списки, однако итерации могут выпол-  
няться по любым последовательностям и итерируемым объектам. Следующий,   
немного похожий, фрагмент выполняет обход уже не списков чисел, а строк,   
и возвращает результаты конкатенации:  
>>> [x + y for x in ‘spam’ for y in ‘SPAM’]  
[‘sS’, ‘sP’, ‘sA’, ‘sM’, ‘pS’, ‘pP’, ‘pA’, ‘pM’,  
‘aS’, ‘aP’, ‘aA’, ‘aM’, ‘mS’, ‘mP’, ‘mA’, ‘mM’]  
В заключение приведу более сложный генератор списка, который иллюстриру-  
ет действие оператора if, присоединенного к вложенному оператору for:  
>>> [(x, y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]  
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]  
Это выражение возвращает возможные комбинации четных и нечетных чисел   
в диапазоне от 0 до 4. Условные выражения отфильтровывают элементы в каж-  
дой из последовательностей. Ниже приводится эквивалентная реализация на   
базе инструкций:

564   
Глава 20. Итераторы и генераторы   
>>> res = []  
>>> for x in range(5):  
... if x % 2 == 0:  
... for y in range(5):  
... if y % 2 == 1:  
... res.append((x, y))  
...  
>>> res  
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]  
Не забывайте, что в случае, когда генератор списков становится слишком   
сложным для понимания, вы всегда можете развернуть вложенные операторы   
for и if (добавляя отступы), чтобы получить эквивалентные инструкции. Про-  
граммный код при этом получится более длинным, но более понятным.  
Эквивалентные реализации на основе функций map и filter оказались бы   
чрезвычайно сложными и имели бы глубокую вложенность вызовов, поэто-  
му я даже не буду пытаться продемонстрировать их. Оставлю эту задачу в ка-  
честве упражнения мастерам Дзен, бывшим программистам на языке LISP   
и просто безумцам.  
Генераторы списков и матрицы  
Конечно, генераторы списков не всегда выглядят настолько искусственно. Рас-  
смотрим еще одно, более сложное применение генераторов списков, в качестве   
упражнения для ума. Основной способ реализации матриц (они же – многомер-  
ные массивы) в языке Python заключается в использовании вложенных спи-  
сков. В следующем примере определяются две матрицы 3x3 в виде вложенных   
списков:  
>>> M = [[1, 2, 3],  
... [4, 5, 6],  
... [7, 8, 9]]  
   
>>> N = [[2, 2, 2],  
... [3, 3, 3],  
... [4, 4, 4]]  
При такой организации всегда можно использовать обычную операцию индек-  
сирования для обращения к строкам и элементам внутри строк:  
>>> M[1]  
[4, 5, 6]  
   
>>> M[1][2]  
6  
Генераторы списков являются мощным средством обработки таких струк-  
тур данных, потому что они позволяют автоматически сканировать строки   
и столбцы матриц. Например, несмотря на то, что при такой организации ма-  
трицы хранятся в виде списка строк, мы легко можем извлечь второй столбец,   
просто обходя строки матрицы и выбирая элементы из требуемого столбца или   
выполняя обход требуемых позиций в строках:   
>>> [row[1] for row in M]  
[2, 5, 8]

Еще раз о генераторах списков: функциональные инструменты   
565  
>>> [M[row][1] for row in (0, 1, 2)]  
[2, 5, 8]  
Пользуясь позициями, мы так же легко можем извлечь элементы, лежащие   
на диагонали. В следующем примере используется функция range – она созда-  
ет список смещений, который затем используется для индексирования строк   
и столбцов одним и тем же значением. В результате сначала выбирается M[0][0],   
затем M[1][1] и так далее (здесь предполагается, что матрица имеет одинаковое   
число строк и столбцов):  
>>> [M[i][i] for i in range(len(M))]  
[1, 5, 9]  
Наконец, проявив немного изобретательности, генераторы списков можно ис-  
пользовать для объединения нескольких матриц. Первый пример ниже созда-  
ет простой список, содержащий результаты умножения соответствующих эле-  
ментов двух матриц, а второй создает структуру вложенных списков, с теми же   
самыми значениями:  
>>> [M[row][col] \* N[row][col] for row in range(3) for col in range(3)]  
[2, 4, 6, 12, 15, 18, 28, 32, 36]  
   
>>> [[M[row][col] \* N[row][col] for col in range(3)] for row in range(3)]  
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]  
В последнем выражении итерации по строкам выполняются во внешнем ци-  
кле: для каждой строки запускается итерация по столбцам, которая создает   
одну строку в матрице с результатами. Это выражение эквивалентно следую-  
щему фрагменту:  
>>> res = []  
>>> for row in range(3):  
... tmp = []  
... for col in range(3):  
... tmp.append(M[row][col] \* N[row][col])  
... res.append(tmp)  
...  
>>> res  
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]  
В отличие от этого фрагмента, версия на базе генератора списков умещается   
в единственную строку и, вероятно, работает значительно быстрее в случае   
больших матриц, но, правда, может и взорвать ваш мозг! На этом перейдем   
к следующему разделу.  
Понимание генераторов списков  
При такой степени гибкости генераторы списков очень быстро могут стать не-  
постижимыми, особенно при наличии вложенных конструкций. Поэтому на-  
чинающим осваивать язык Python я рекомендую в большинстве случаев ис-  
пользовать простые циклы for и функцию map, а генераторы – в отдельных   
случаях (если они получаются не слишком сложными). Здесь также действует   
правило «чем проще, тем лучше»: лаконичность программного кода – намного   
менее важная цель, чем его удобочитаемость.  
Однако в настоящее время усложнение программного кода обеспечивает бо-  
лее высокую его производительность: проведенные тесты свидетельствуют,

566   
Глава 20. Итераторы и генераторы   
что функция map работает практически в два раза быстрее, чем эквивалентные   
циклы for, а генераторы списков обычно немного быстрее, чем функция map.1   
Это различие в скорости выполнения обусловлено тем фактом, что функция   
map и генераторы списков реализованы на языке C, что обеспечивает более   
высокую скорость, чем выполнение циклов for внутри виртуальной машины   
Python (PVM).  
Применение циклов for делает логику программы более явной, поэтому я реко-  
мендую использовать их для обеспечения большей простоты. Однако функция   
map и генераторы списков стоят того, чтобы знать и применять их для реали-  
зации простых итераций, а также в случаях, когда скорость работы прило-  
жения имеет критически важное значение. Кроме того, функция map и гене-  
раторы списков являются выражениями и синтаксически могут находиться   
там, где недопустимо использовать инструкцию for, например в теле lambda-  
выражений, в литералах списков и словарей и во многих других случаях. То   
есть вы должны стараться писать простые функции map и генераторы списков,   
а в более сложных случаях использовать полные инструкции.  
Придется держать в уме: генераторы списков и map  
Ниже приводится более реалистичный пример использования генера-  
торов списков и функции map (мы решали эту задачу с помощью гене-  
раторов списков в главе 14, а здесь мы снова вернемся к ней, чтобы про-  
демонстрировать альтернативную реализацию на базе функции map).   
Вспомните, что метод файлов readlines возвращает строки с символом   
конца строки (\n) в конце:  
>>> open(‘myfile’).readlines()  
[‘aaa\n’, ‘bbb\n’, ‘ccc\n’]  
Если требуется удалить символы конца строки, их можно отсечь сра-  
зу во всех строках за одно действие с помощью генератора списков или   
функции map (в Python 3.0 функция map возвращает итерируемый объ-  
ект, поэтому нам пришлось использовать функцию list, чтобы полу-  
чить весь список с результатами в интерактивном сеансе):  
>>> [line.rstrip() for line in open(‘myfile’).readlines()]  
[‘aaa’, ‘bbb’, ‘ccc’]  
   
1   
Производительность этих тестов может зависеть от вида решаемой задачи, а также   
от изменений и оптимизаций в самом интерпретаторе языка Python. Например, в по-  
следних версиях Python была увеличена скорость выполнения инструкции цикла   
for. Тем не менее генераторы списков обычно показывают более высокую скорость   
работы, чем циклы for, и даже более высокую, чем функция map (хотя функция map   
может выйти победителем в состязании среди встроенных функций). Чтобы прове-  
рить скорость работы альтернативных реализаций, можно использовать функции   
time.clock и time.time в модуле time. В версии Python 2.4 появился новый модуль   
timeit, который рассматривается в разделе «Хронометраж итерационных альтерна-  
тив» далее в этой главе.

Еще раз об итераторах: генераторы   
567  
>>> [line.rstrip() for line in open(‘myfile’)]  
[‘aaa’, ‘bbb’, ‘ccc’]  
   
>>> list(map((lambda line: line.rstrip()), open(‘myfile’)))  
[‘aaa’, ‘bbb’, ‘ccc’]  
В последних двух случаях используются файловые итераторы (по сути   
это означает, что вам не требуется вызывать метод, который будет чи-  
тать строки из файла). Вызов функции map выглядит немного длиннее,   
чем генератор списков, но ни в одном из этих двух случаев не требуется   
явно управлять конструированием списка результатов.  
Кроме того, генераторы списков могут играть роль операции извлечения   
столбца. Стандартный прикладной интерфейс доступа к базам данных   
в языке Python возвращает результаты запроса в виде списка кортежей,   
как показано ниже. Список – это таблица, кортежи – это строки, а эле-  
менты кортежей – это значения столбцов:  
listoftuple = [(‘bob’, 35, ‘mgr’), (‘mel’, 40, ‘dev’)]  
Выбрать все значения из определенного столбца можно и вручную, с по-  
мощью цикла for, но функция map и генераторы списков сделают это бы-  
стрее и за один шаг:  
>>> [age for (name, age, job) in listoftuple]  
[35, 40]  
   
>>> list(map((lambda (name, age, job): age), listoftuple))  
[35, 40]  
В первом случае используется операция присваивания кортежа – чтобы   
извлечь значения в список, а во втором – операция доступа к элементу   
последовательности по его индексу. В Python 2.6 (но не в 3.0� смотрите   
примечание к операции распаковывания аргументов в главе 18) для рас-  
паковывания кортежей может также использоваться функция map:  
# Только в версии 2.6  
>>> list(map((lambda (name, age, job): age), listoftuple))  
[35, 40]  
За дополнительной информацией о прикладных интерфейсах языка   
Python обращайтесь к другим книгам и источникам информации.  
Помимо обычных отличий между функцией и выражением, главное от-  
личие между функцией map и генератором списков в Python 3.0 состоит   
в том, что функция map возвращает итератор, воспроизводящий резуль-  
таты по требованию� чтобы добиться такой же экономии памяти, гене-  
раторы списков должны оформляться в виде выражений-генераторов   
(одна из тем этой главы).  
Еще раз об итераторах: генераторы  
На сегодняшний день язык Python более широко поддерживает отложенные   
операции, чем в прошлом, – он предоставляет инструменты, позволяющие реа-

568   
Глава 20. Итераторы и генераторы   
лизовать воспроизведение результатов по требованию, а не всех сразу. В част-  
ности, существуют две языковые конструкции, откладывающие создание ре-  
зультатов, когда это возможно:  
 •  
Функции-генераторы – выглядят как обычные инструкции def, но для воз-  
врата результатов по одному значению за раз используют инструкцию yield,   
которая приостанавливает выполнение функции.  
 •  
Выражения-генераторы – напоминают генераторы списков, о которых рас-  
сказывалось в предыдущем разделе, но они не конструируют список с ре-  
зультатами, а возвращают объект, который будет воспроизводить результа-  
ты по требованию.  
Поскольку ни одна из этих конструкций не создает сразу весь список с резуль-  
татами, они позволяют экономить память и производить дополнительные вы-  
числения между операциями получения результатов. Как мы увидим далее,   
обе конструкции поддерживают такую возможность возврата результатов по   
требованию за счет реализации протокола  итераций, который мы изучили   
в главе 14.  
Функции-генераторы: инструкция yield вместо return  
В этой части книги мы уже познакомились с обычными функциями, которые   
получают входные параметры и возвращают единственный результат. Однако   
точно так же возможно написать функцию, которая может возвращать значе-  
ние, а позднее продолжить свою работу с того места, где она была приостанов-  
лена. Такие функции известны как функции-генераторы, потому что они гене-  
рируют последовательность значений с течением времени.  
Функции-генераторы во многом похожи на обычные функции и в действитель-  
ности создаются с помощью инструкции def. Единственное отличие состоит   
в том, что они автоматически поддерживают протокол итераций и могут ис-  
пользоваться в контексте итераций. Мы рассмотрели итераторы в главе 14,   
а здесь взглянем на них еще раз, чтобы увидеть, какое отношение они имеют   
к генераторам.  
Замораживание состояния  
В отличие от обычных функций, которые возвращают значение и завершают   
работу, функции-генераторы автоматически приостанавливают и возобнов-  
ляют свое выполнение, при этом сохраняя информацию, необходимую для   
генерации значений. Вследствие этого они часто представляют собой удобную   
альтернативу вычислению всей серии значений заранее, с ручным сохранени-  
ем и восстановлением состояния в классах. Функции-генераторы при приоста-  
новке автоматически сохраняют информацию о своем состоянии, под которым   
понимается вся локальная область видимости, со всеми локальными перемен-  
ными, которая становится доступной сразу же, как только функция возобнов-  
ляет работу.  
Главное отличие функций-генераторов от обычных функций состоит в том, что   
функция-генератор поставляет значение, а не возвращает его – инструкция   
yield приостанавливает работу функции и передает значение вызывающей   
программе, при этом сохраняется информация о состоянии, необходимая,   
чтобы возобновить работу с того места, где она была приостановлена. Когда

Еще раз об итераторах: генераторы   
569  
функция-генератор возобновляет работу, ее выполнение продолжается с пер-  
вой инструкции, следующей за инструкцией yield. Это позволяет функциям   
воспроизводить последовательности значений в течение долгого времени, вме-  
сто того чтобы создавать всю последовательность сразу и возвращать ее в виде   
некоторой конструкции, такой как список.  
Интеграция с протоколом итераций  
Чтобы по-настоящему понять функции-генераторы, вы должны знать, что они   
тесно связаны с понятием протокола итераций в языке Python. Как мы уже   
знаем, итерируемые объекты определяют метод \_\_next\_\_, который либо воз-  
вращает следующий элемент в итерации, либо возбуждает особое исключение   
StopIteration по окончании итераций. Доступ к итератору объекта можно по-  
лучить с помощью встроенной функции iter.   
Циклы for и другие итерационные механизмы в языке Python используют   
такой итерационный протокол, если он поддерживается, для обхода последо-  
вательностей или значений генераторов. Если протокол не поддерживается,   
инструкция for терпит неудачу и возвращается к операции индексирования   
последовательности.  
Чтобы обеспечить поддержку этого протокола, функции, содержащие ин-  
струкцию yield, компилируются именно как генераторы. При вызове такой   
функции она возвращает объект генератора, поддерживающий интерфейс ите-  
раций с помощью автоматически созданного метода \_\_next\_\_, позволяющего   
продолжить выполнение. Функции-генераторы точно так же могут включать   
инструкцию return, которая не только служит окончанием блока def, но и за-  
вершает генерацию значений, возбуждая исключение StopIteration после вы-  
полнения обычного выхода из функции. С точки зрения вызывающей програм-  
мы, метод \_\_next\_\_ генератора возобновляет выполнение функции, работа ко-  
торой продолжается, пока она не встретит следующую инструкцию yield или   
пока не возбудит исключение StopIteration.  
Таким образом, функции-генераторы оформляются как обычные инструкции   
def, но содержащие инструкции yield, за счет чего они автоматически получа-  
ют поддержку протокола итераций и могут использоваться в любых разновид-  
ностях итераций для воспроизведения результатов по требованию.  
Как отмечалось в главе 14, в Python 2.6 и в более ранних версиях   
итерируемые объекты объявляют метод next, а не \_\_next\_\_. То же   
относится и к объектам генераторов, которые мы обсуждаем.   
В версии 3.0 метод next был переименован в \_\_next\_\_. Для обе-  
спечения удобства и совместимости предоставляется встроен-  
ная функция next: вызов next(I) – это то же самое, что вызов   
I.\_\_next\_\_() в версии 3.0 и I.next() в версии 2.6. Для реализации   
итераций вручную в более ранних версиях Python программы   
должны вызывать метод I.next().  
Пример функции-генератора  
Чтобы проиллюстрировать основные моменты, рассмотрим следующий фраг-  
мент, где определяется функция-генератор, которая может использоваться для   
генерации серии квадратов чисел:

570   
Глава 20. Итераторы и генераторы   
>>> def gensquares(N):  
... for i in range(N):  
... yield i \*\* 2 # Позднее продолжить работу с этого места  
...  
Эта функция поставляет значение и тем самым возвращает управление вызы-  
вающей программе на каждой итерации цикла – когда она возобновляет ра-  
боту, восстанавливается ее предыдущее состояние и управление передается   
непосредственно в точку, находящуюся сразу же за инструкцией yield. Напри-  
мер, при использовании в заголовке цикла for управление возвращается функ-  
ции на каждой итерации в точку, находящуюся сразу же за инструкцией yield:  
>>> for i in gensquares(5): # Возобновить работу функции  
... print(i, end = ‘ : ‘) # Вывести последнее полученное значение  
...  
0 : 1 : 4 : 9 : 16 :  
>>>  
Чтобы завершить генерацию значений, функция может либо воспользоваться   
инструкцией return без значения, либо просто позволить потоку управления   
достичь конца функции.  
Если вам интересно узнать, что происходит внутри цикла for, вызовите   
функцию-генератор напрямую:  
>>> x = gensquares(4)  
>>> x  
<generator object at 0x0086C378>  
Здесь функция вернула объект-генератор, который поддерживает протокол   
итераций, с которым мы познакомились в главе 14, – то есть имеет метод   
\_\_next)), который запускает функцию или возобновляет ее работу с места, от-  
куда было поставлено последнее значение, а также возбуждает исключение   
StopIteration по достижении конца последовательности значений. Для удобства   
в языке Python была создана встроенная функция next, вызов next(X) которой   
производит вызов метода X.\_\_next\_\_() объекта:  
>>> next(x) # То же, что и x.\_\_next\_\_() в версии 3.0  
0  
>>> next(x) # В версии 2.6 используйте вызов x.next() или next()  
1  
>>> next(x)  
4  
>>> next(x)  
9  
>>> next(x)  
Traceback (most recent call last):  
 ...остальной текст опущен...  
StopIteration  
Как мы узнали в главе 14, циклы for (и другие итерационные контексты) рабо-  
тают с генераторами точно так же – вызывают метод \_\_next\_\_ в цикле, пока не   
будет получено исключение. Если итерируемый объект не поддерживает этот   
протокол, вместо него цикл for использует протокол доступа к элементам по   
индексам.

Еще раз об итераторах: генераторы   
571  
Обратите внимание, что в этом примере мы могли бы просто сразу создать спи-  
сок всех значений:  
>>> def buildsquares(n):  
... res = []  
... for i in range(n): res.append(i\*\*2)  
... return res  
...  
>>> for x in buildsquares(5): print(x, end=’ : ‘)  
...  
0 : 1 : 4 : 9 : 16 :  
В такой ситуации мы могли бы использовать любой из приемов: цикл for, функ-  
цию map или генератор списков:  
>>> for x in [n\*\*2 for n in range(5)]:  
... print(x, end=’ : ‘)  
...  
0 : 1 : 4 : 9 : 16 :  
   
>>> for x in map((lambda x:x\*\*2), range(5)):  
... print(x, end=’ : ‘)  
...  
0 : 1 : 4 : 9 : 16 :  
Однако генераторы предлагают лучшее решение с точки зрения использования   
памяти и производительности. Они дают возможность избежать необходимо-  
сти выполнять всю работу сразу, что особенно удобно, когда список результатов   
имеет значительный объем или когда вычисление каждого значения занима-  
ет продолжительное время. Генераторы распределяют время, необходимое на   
создание всей последовательности значений, по отдельным итерациям цикла.   
Кроме того, в более сложных случаях использования они обеспечивают про-  
стую альтернативу сохранению состояния вручную между вызовами в объ-  
ектах классов – в случае с генераторами переменные функций сохраняются   
и восстанавливаются автоматически.1 Подробнее об итераторах на базе клас-  
сов рассказывается в шестой части книги.  
1   
Интересно отметить, что функции-генераторы до определенной степени могут слу-  
жить средством, обеспечивающим многопоточный  режим  работы, – выполнение   
функций чередуется с периодами работы вызывающей программы, в результате чего   
создается эффект пошагового выполнения операций между вызовами инструкции   
yields. Впрочем, генераторы не обеспечивают истинный многопоточный режим ра-  
боты: программа явно выполняет вход и выход из функции в пределах единственного   
потока управления. В некотором смысле многопоточность представляет собой нечто   
большее (поставщики информации действительно могут выполняться независимо   
от основной программы и помещать данные в очередь), но работать с генераторами   
намного проще. Краткое введение в средства поддержки многопоточного режима ра-  
боты программ на языке Python приводится в главе 17, во второй сноске. Обратите   
внимание: вследствие того, что управление выполнением производится явно с помо-  
щью инструкции yield и последующим вызовом функции next, генераторы не имеют   
собственного стека возвратов и более тесно связаны с сопрограммами – формаль-  
ной концепцией, обсуждение которой выходит далеко за рамки этой главы.

572   
Глава 20. Итераторы и генераторы   
Расширенный протокол функций-генераторов: send и next   
В версии Python 2.5 в протокол функций-генераторов был добавлен метод send.   
Метод send не только выполняет переход к следующему элементу в последова-  
тельности результатов, как это делает метод next, но еще и обеспечивает для   
вызывающей программы способ взаимодействия с генератором, влияя на его   
работу.  
С технической точки зрения yield в настоящее время является не инструкци-  
ей, а выражением, которое возвращает элемент, передаваемый методу send (не-  
смотря на то, что его можно использовать любым из двух способов – как yield   
X или как A = (yield X)). Когда выражение yield помещается справа от оператора   
присваивания, оно должно заключаться в круглые скобки, за исключением   
случая, когда оно не является составной частью более крупного выражения.   
Например, правильно будет написать X = yield Y, а также X = (yield Y) + 42.   
При использовании расширенного протокола значения передаются генератору   
G вызовом метода G.send(value). После этого программный код генератора возоб-  
новляет работу, и выражение yield возвращает значение, полученное от метода   
send. Когда вызывается обычный метод G.\_\_next\_\_() (или выполняется эквива-  
лентный вызов next(G)), выражение yield возвращает None. Например:  
>>> def gen():  
... for i in range(10):  
... X = yield i  
... print(X)  
...  
>>> G = gen()  
>>> next(G) # Чтобы запустить генератор, необходимо сначала вызвать next()  
0  
>>> G.send(77) # Переход к следующему значению   
77 # и передача значения выражению yield  
1  
>>> G.send(88)  
88  
2  
>>> next(G) # next() и X.\_\_next\_\_() передают значение None  
None  
3  
Метод send может использоваться, например, чтобы реализовать генератор, ко-  
торый можно будет завершать из вызывающей программы или переустанавли-  
вать в нем текущую позицию в последовательности результатов. Кроме того,   
генераторы в версии 2.5 поддерживают метод throw(type) для возбуждения ис-  
ключения внутри генератора в последнем выражении yield и метод close, ко-  
торый возбуждает специальное исключение GeneratorExit внутри генератора,   
чтобы вынудить его завершить итерации. Мы не будем углубляться здесь в эти   
расширенные возможности – за дополнительной информацией обращайтесь   
к стандартным руководствам по языку Python.  
Обратите внимание, что в Python 3.0 имеется удобная встроенная функция   
next(X), выполняющая вызов метода X.\_\_next\_\_() объекта, тогда как другие   
методы генератора, такие как send, должны вызываться непосредственно, как   
методы объекта генератора (например, G.send(X)). Это объясняется тем, что все   
дополнительные методы реализованы только во встроенных объектах гене-

Еще раз об итераторах: генераторы   
573  
раторов, тогда как метод \_\_next\_\_ имеется у всех итерируемых объектов (как   
встроенных типов, так и пользовательских классов).  
Выражения-генераторы:   
итераторы и генераторы списков  
Во всех последних версиях Python понятия итератора и генератора списков   
были объединены в новую языковую конструкцию – выражения-генераторы.   
Синтаксически выражения напоминают обычные генераторы списков, но они   
заключаются не в квадратные, а в круглые скобки:  
>>> [x \*\* 2 for x in range(4)] # Генератор списков: создает список  
[0, 1, 4, 9]  
   
>>> (x \*\* 2 for x in range(4)) # Выражение-генератор: создает   
<generator object at 0x011DC648> # итерируемый объект  
Фактически, по крайней мере с функциональной точки зрения, генератор спи-  
сков является эквивалентом генератора-выражения, обернутого в вызов встро-  
енной функции list для принудительного получения сразу всего списка с ре-  
зультатами:  
>>> list(x \*\* 2 for x in range(4)) # Эквивалент генератора списков  
[0, 1, 4, 9]  
Однако с другой стороны, выражения-генераторы кардинально отличаются от   
генераторов списков – вместо того, чтобы создавать в памяти список с резуль-  
татами, они возвращают объект-генератор, который в свою очередь поддержи-  
вает протокол итераций, поставляя по одному элементу списка за раз в любом   
итерационном контексте:  
>>> G = (x \*\* 2 for x in range(4))  
>>> next(G)  
0  
>>> next(G)  
1  
>>> next(G)  
4  
>>> next(G)  
9  
>>> next(G)  
   
Traceback (most recent call last):  
 ...остальной текст опущен...  
StopIteration  
Как правило, нам не приходится наблюдать итерационную механику действий   
выражений-генераторов, как в данном примере, потому что циклы for делают   
это автоматически:  
>>> for num in (x \*\* 2 for x in range(4)):  
... print(‘%s, %s’ % (num, num / 2.0))  
...  
0, 0.0  
1, 0.5  
4, 2.0  
9, 4.5

574   
Глава 20. Итераторы и генераторы   
Как мы уже знаем, именно таким способом работает любой итерационный   
контекст, включая встроенные функции sum, map и sorted, генераторы списков   
и другие итерационные инструменты, которые мы рассматривали в главе 14,   
такие как встроенные функции all, any и list.  
Обратите внимание, что круглые скобки вокруг выражения-генератора можно   
опустить, если оно является единственным элементом, заключенным в другие   
круглые скобки, например в вызове функции. Однако круглые скобки необхо-  
димы во втором вызове функции sorted:  
>>> sum(x \*\* 2 for x in range(4))  
14  
   
>>> sorted(x \*\* 2 for x in range(4))  
[0, 1, 4, 9]  
   
>>> sorted((x \*\* 2 for x in range(4)), reverse=True)  
[9, 4, 1, 0]  
   
>>> import math  
>>> list(map(math.sqrt, (x \*\* 2 for x in range(4))))  
[0.0, 1.0, 2.0, 3.0]  
Выражения-генераторы в первую очередь оптимизируют использование памя-  
ти – они не требуют создания в памяти полного списка с результатами, как   
это делают генераторы списков в квадратных скобках. Кроме того, на прак-  
тике они могут работать несколько медленнее, поэтому их лучше использо-  
вать, только когда объем результатов очень велик. Более точно проверить это   
утверждение нам поможет сценарий хронометража, который мы напишем   
ниже, в этой главе.  
Функции-генераторы и выражения-генераторы  
Интересно отметить, что одни и те же итерации часто можно реализовать как   
в виде функции-генератора, так и в виде выражения-генератора. Следующее   
выражение-генератор, например, четырежды повторяет каждый символ в ис-  
ходной строке:  
>>> G = (c \* 4 for c in ‘SPAM’) # Выражение-генератор  
>>> list(G) # Принудительно получить сразу все результаты  
[‘SSSS’, ‘PPPP’, ‘AAAA’, ‘MMMM’]  
Эквивалентная функция-генератор содержит чуть больше программного кода,   
но, будучи функцией, при необходимости способна вместить в себя больший   
объем логики и использовать больший объем информации:  
>>> def timesfour(S): # Функция-генератор  
... for c in S:  
... yield c \* 4  
...  
>>> G = timesfour(‘spam’)  
>>> list(G) # Выполнит итерации автоматически  
[‘ssss’, ‘pppp’, ‘aaaa’, ‘mmmm’]  
Обе разновидности генераторов поддерживают автоматическое и ручное управ-  
ление итерациями – в предыдущем примере вызов функции list обеспечивает   
автоматическое выполнение итераций, а в следующем примере демонстрирует-  
ся выполнение итераций вручную:

Еще раз об итераторах: генераторы   
575  
>>> G = (c \* 4 for c in ‘SPAM’)  
>>> I = iter(G) # Итерации выполняются вручную  
>>> next(I)  
‘SSSS’  
>>> next(I)  
‘PPPP’  
>>> G = timesfour(‘spam’)  
>>> I = iter(G)  
>>> next(I)  
‘ssss’  
>>> next(I)  
‘pppp’  
Обратите внимание, что для повторного получения результатов нам пришлось   
создавать новые генераторы – как описывается в следующем разделе, генерато-  
ры являются итераторами однократного применения.  
Генераторы – это объекты итераторов   
однократного применения  
И функции-генераторы, и выражения-генераторы имеют свои собственные   
итераторы и потому поддерживают возможность лишь однократного выполне-  
ния итераций – в отличие от некоторых встроенных типов, нет никакой воз-  
можности получить несколько итераторов или выполнять позиционирование   
итератора в множестве результатов. Например, для выражения-генератора из   
предыдущего раздела итератором является сам генератор (фактически вызов   
метода iter генератора не выполняет никаких действий):  
>>> G = (c \* 4 for c in ‘SPAM’)  
>>> iter(G) is G # Итератором генератора является сам генератор:   
True # G имеет метод \_\_next\_\_  
Если попробовать выполнить обход результатов вручную, с помощью несколь-  
ких итераторов, окажется, что все они ссылаются на одну и ту же позицию   
в последовательности результатов:  
>>> G = (c \* 4 for c in ‘SPAM’) # Создать новый генератор  
>>> I1 = iter(G) # Выполнить итерацию вручную  
>>> next(I1)  
‘SSSS’  
>>> next(I1)  
‘PPPP’  
>>> I2 = iter(G) # Второй итератор   
>>> next(I2) # ссылается на ту же позицию!  
‘AAAA’  
Кроме того, как только итерации достигнут конца, все результаты окажутся   
исчерпаны – чтобы выполнить повторный обход результатов, нам придется   
создать новый генератор:  
>>> list(I1) # Выберет остатки результаов в I1  
[‘MMMM’]  
>>> next(I2) # Другие итераторы также окажутся исчерпанными  
StopIteration  
   
>>> I3 = iter(G) # То же относится и к вновь созданным итераторам  
>>> next(I3)

576   
Глава 20. Итераторы и генераторы   
StopIteration  
   
>>> I3 = iter(c \* 4 for c in ‘SPAM’) # Создать новый генератор, чтобы   
>>> next(I3) # выполнить повторный обход результатов  
‘SSSS’  
То же относится и к функциям-генераторам – следующий эквивалентный гене-  
ратор, реализованный в виде функции, также обеспечивает возможность одно-  
кратного выполнения итераций:  
>>> def timesfour(S):  
... for c in S:  
... yield c \* 4  
...  
>>> G = timesfour(‘spam’) # Функция-генератор действует точно так же  
>>> iter(G) is G  
True  
>>> I1, I2 = iter(G), iter(G)  
>>> next(I1)  
‘ssss’  
>>> next(I1)  
‘pppp’  
>>> next(I2) # I2 находится в той же позиции, что и I1  
‘aaaa’  
Некоторые встроенные типы имеют отличное поведение, поддерживают воз-  
можность создания множества независимых итераторов и способны отражать   
изменения объекта во всех активных итераторах:  
>>> L = [1, 2, 3, 4]  
>>> I1, I2 = iter(L), iter(L)  
>>> next(I1)  
1  
>>> next(I1)  
2  
>>> next(I2) # Списки поддерживают множество независимых итераторов  
1  
>>> del L[2:] # Изменения отражаются на всех итераторах  
>>> next(I1)  
StopIteration  
Когда в шестой части книги мы начнем создавать итераторы на основе классов,   
то увидим, что мы в состоянии сами решать, какой тип итераций будут под-  
держивать наши объекты, если они будут итерационными.   
Имитация функций zip и map   
с помощью инструментов итераций  
Чтобы продемонстрировать всю мощь инструментов итераций в действии, рас-  
смотрим несколько более сложных примеров. Освоив генераторы списков, ге-  
нераторы и другие инструменты итераций, вы обнаружите, что реализация   
многих встроенных функций языка Python проста и наглядна.  
Например, мы уже видели, как встроенные функции zip и map позволяют объ-  
единять итерируемые объекты и отображать на них функции соответственно.   
При вызове с несколькими аргументами функция map отображает заданную

Еще раз об итераторах: генераторы   
577  
функцию на элементы, взятые из каждой последовательности, практически   
тем же способом, каким функция zip объединяет их:  
>>> S1 = ‘abc’  
>>> S2 = ‘xyz123’  
>>> list(zip(S1, S2)) # zip объединяет элементы итерируемых объектов  
[(‘a’, ‘x’), (‘b’, ‘y’), (‘c’, ‘z’)]  
   
# zip объединяет элементы, усекая результирующую последовательность  
# по длине кратчайшей исходной последовательности  
   
>>> list(zip([-2, -1, 0, 1, 2])) # Единственная последовательность:   
[(-2,), (-1,), (0,), (1,), (2,)] # 1-мерные кортежи  
   
>>> list(zip([1, 2, 3], [2, 3, 4, 5])) # N последовательностей:   
[(1, 2), (2, 3), (3, 4)] # N-мерные кортежи  
   
# map передает объединенные элементы последовательностей указанной функции,   
# усекая результирующую последовательность  
# по длине кратчайшей исходной последовательности  
   
>>> list(map(abs, [-2, -1, 0, 1, 2])) # Единственная последовательность:   
[2, 1, 0, 1, 2] # 1-мерная функция  
   
>>> list(map(pow, [1, 2, 3], [2, 3, 4, 5])) # N последовательностей:   
[1, 8, 81] # N-мерная функция  
Разумеется, эти функции имеют различное назначение, но если внимательно   
изучить эти примеры, можно заметить определенную связь между результа-  
тами функции zip и отображаемыми аргументами функции map, которую мы   
будем использовать в следующем примере.  
Создание собственной версии функции map(func, ...)  
Встроенные функции map и zip показывают высокую производительность   
и удобны в использовании, тем не менее мы всегда можем добиться той же   
функциональности, написав несколько строк самостоятельно. В предыдущей   
главе, например, мы видели реализацию функции, имитирующей поведение   
встроенной функции map для случая, когда ей передается единственная после-  
довательность. Нам не составит большого труда распространить эту версию на   
случай множества последовательностей:  
# map(func, seqs...) на основе использования zip  
def mymap(func, \*seqs):  
 res = []  
 for args in zip(\*seqs):  
 res.append(func(\*args))  
 return res  
   
print(mymap(abs, [-2, -1, 0, 1, 2]))  
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))  
Эта версия в значительной степени опирается на конструкцию \*args передачи   
аргументов – она получает множество аргументов-последовательностей (в дей-  
ствительности – итерируемых объектов), распаковывает их в аргументы функ-  
ции zip, для последующего объединения, и затем распаковывает результаты   
вызова функции zip в аргументы указанной функции func. То есть мы исполь-

578   
Глава 20. Итераторы и генераторы   
зуем тот факт, что объединение элементов последовательностей, по сути, явля-  
ется промежуточной операцией при отображении. Проверочный программный   
код, находящийся внизу сценария, применяет эту функцию сначала к одной   
последовательности, а потом к двум и воспроизводит следующие результаты   
(те же результаты дает применение встроенной функции map):  
[2, 1, 0, 1, 2]  
[1, 8, 81]  
Однако в действительности предыдущая версия функции представляет собой   
классический шаблон генератора списков, конструируя список с результатами   
с помощью цикла for. Учитывая это обстоятельство, мы можем сделать нашу   
версию функции map более компактной, реализовав все необходимое в виде од-  
нострочного генератора списков:  
# С использованием генератора списков  
   
def mymap(func, \*seqs):  
 return [func(\*args) for args in zip(\*seqs)]  
   
print(mymap(abs, [-2, -1, 0, 1, 2]))  
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))  
Запустив этот пример, мы получим те же результаты, что и выше, но сама реа-  
лизация получилась более компактной и, возможно, более быстрой (подробнее   
о производительности рассказывается в разделе «Хронометраж итерационных   
альтернатив», ниже). Обе предыдущие версии функции mymap конструируют   
сразу весь список с результатами, что для крупных списков может привести   
к существенному увеличению потребления памяти. Теперь, когда мы позна-  
комились с функциями-генераторами и выражениями-генераторами, мы лег-  
ко можем создать две альтернативные версии, возвращающие результаты по   
требованию:  
# С использованием генераторов: yield и (...)  
   
def mymap(func, \*seqs):  
 res = []  
 for args in zip(\*seqs):  
 yield func(\*args)  
   
def mymap(func, \*seqs):  
 return (func(\*args) for args in zip(\*seqs))  
Эти версии воспроизводят те же самые результаты, но возвращают генераторы,   
поддерживающие протокол итераций, – первая версия поставляет по одному   
результату за раз, а вторая возвращает результат выражения-генератора, кото-  
рый имеет тот же эффект. Если обернуть вызовы этих функций в вызов функ-  
ции list, мы сможем получить сразу все результаты:  
print(list(mymap(abs, [-2, -1, 0, 1, 2])))  
print(list(mymap(pow, [1, 2, 3], [2, 3, 4, 5])))  
В действительности здесь не выполняется никакой работы, пока вызов функ-  
ции list не запустит генераторы, активируя их посредством протокола ите-  
раций. Генераторы, возвращаемые этими функциями, как и генераторы, воз-  
вращаемые встроенной функцией zip в Python 3.0, воспроизводят результаты   
только по требованию.

Еще раз об итераторах: генераторы   
579  
Создание собственных версий функций   
zip(...) и map(None, ...)   
Конечно, тайной пружиной предыдущих примеров является использование   
встроенной функции zip для объединения элементов из нескольких последо-  
вательностей. Можно также заметить, что наши версии функции map на самом   
деле имитируют поведение функции map в Python 3.0, – они усекают длину   
последовательности результатов по кратчайшей исходной последовательно-  
сти и не поддерживают возможность подстановки значений, когда длины ис-  
ходных последовательностей отличаются, как это позволяет функция map в Py-  
Py-  
thon 2.X, когда ей передается аргумент None:  
C:\misc> c:\python26\python  
>>> map(None, [1, 2, 3], [2, 3, 4, 5])  
[(1, 2), (2, 3), (3, 4), (None, 5)]  
>>> map(None, ‘abc’, ‘xyz123’)  
[(‘a’, ‘x’), (‘b’, ‘y’), (‘c’, ‘z’), (None, ‘1’), (None, ‘2’), (None, ‘3’)]  
Используя инструменты итераций, мы можем создать версии, имитирующие   
усечение, как это делает функция zip, и дополнение, как это делает функция   
map в версии 2.6. Как оказывается, эти версии почти не отличаются от предыду-  
щих:  
# Версии zip(seqs...) и map(None, seqs...) в Python 2.6  
   
def myzip(\*seqs):  
 seqs = [list(S) for S in seqs]  
 res = []  
 while all(seqs):  
 res.append(tuple(S.pop(0) for S in seqs))  
 return res  
   
def mymapPad(\*seqs, pad=None):  
 seqs = [list(S) for S in seqs]  
 res = []  
 while any(seqs):  
 res.append(tuple((S.pop(0) if S else pad) for S in seqs))  
 return res  
   
S1, S2 = ‘abc’, ‘xyz123’  
print(myzip(S1, S2))  
print(mymapPad(S1, S2))  
print(mymapPad(S1, S2, pad=99))  
Обе функции, представленные здесь, способны работать с любыми итерируе-  
мыми объектами, потому что они применяют к своим аргументам встроенную   
функцию list, вынуждая их сгенерировать все результаты (что, к примеру, по-  
зволяет передавать в качестве аргументов не только простые последовательно-  
сти, такие как строки, но и файлы). Обратите внимание, что здесь используют-  
ся встроенные функции all и any, – они возвращают True, если все или хотя бы   
один элемент итерируемого объекта соответственно имеет истинное значение   
(то есть непустой). Эти встроенные функции позволяют остановить итерации,   
когда один или все аргументы, пропущенные через функцию list, превраща-  
ются в пустые списки после удаления очередного элемента.  
Обратите также внимание на использование аргумента pad, который в Py-  
Py-  
thon 3.0 может передаваться только  по  имени, – в отличие от функции map

580   
Глава 20. Итераторы и генераторы   
в версии 2.6, наша версия позволяет указывать любые объекты, которые будут   
использоваться вместо отсутствующих элементов (если вы пользуетесь верси-  
ей 2.6, для поддержки этой возможности используйте форму передачи имено-  
ванных аргументов \*\*kargs вместо pad=None� подробности ищите в главе 18). За-  
пустив этот сценарий, вы получите следующие результаты – результат вызова   
функции zip и двух вызовов функции map, поддерживающей дополнение недо-  
стающих элементов:  
[(‘a’, ‘x’), (‘b’, ‘y’), (‘c’, ‘z’)]  
[(‘a’, ‘x’), (‘b’, ‘y’), (‘c’, ‘z’), (None, ‘1’), (None, ‘2’), (None, ‘3’)]  
[(‘a’, ‘x’), (‘b’, ‘y’), (‘c’, ‘z’), (99, ‘1’), (99, ‘2’), (99, ‘3’)]  
Эти функции не могут быть преобразованы в генераторы списков из-за того,   
что в них используются слишком специфичные циклы. Однако, как и прежде,   
текущие версии функций zip и map конструируют и возвращают сразу весь спи-  
сок с результатами и их точно так же легко можно превратить в генераторы,   
поставляющие результаты по одному элементу за раз. Результаты получаются   
теми же самыми, что и прежде, но нам снова придется использовать функцию   
list, чтобы получить сразу все значения для отображения:  
# С использованием генераторов: yield  
   
def myzip(\*seqs):  
 seqs = [list(S) for S in seqs]  
 while all(seqs):  
 yield tuple(S.pop(0) for S in seqs)  
   
def mymapPad(\*seqs, pad=None):  
 seqs = [list(S) for S in seqs]  
 while any(seqs):  
 yield tuple((S.pop(0) if S else pad) for S in seqs)  
   
S1, S2 = ‘abc’, ‘xyz123’  
print(list(myzip(S1, S2)))  
print(list(mymapPad(S1, S2)))  
print(list(mymapPad(S1, S2, pad=99)))  
Наконец, ниже приводятся альтернативные реализации наших версий функ-  
ций zip и map – вместо того, чтобы удалять элементы из списков с помощью   
метода pop, они выполняют свою работу за счет нахождения аргументов с ми-  
нимальной и максимальной длиной. Вооружившись этими значениями, легко   
можно реализовать вложенные генераторы списков для обхода диапазонов ин-  
дексов в аргументах:  
# Альтернативные реализации с вычислением длин исходных последовательностей  
   
def myzip(\*seqs):  
 minlen = min(len(S) for S in seqs)  
 return [tuple(S[i] for S in seqs) for i in range(minlen)]  
   
def mymapPad(\*seqs, pad=None):  
 maxlen = max(len(S) for S in seqs)  
 index = range(maxlen)  
 return [tuple((S[i] if len(S) > i else pad) for S in seqs) for i in index]  
   
S1, S2 = ‘abc’, ‘xyz123’  
print(myzip(S1, S2))  
print(mymapPad(S1, S2))

Еще раз об итераторах: генераторы   
581  
print(mymapPad(S1, S2, pad=99))  
Поскольку в этих версиях используются функция len и операция извлечения   
элементов по индексам, они уже не предполагают, что их аргументы могут   
быть произвольными итерируемыми объектами. Внешний генератор списков   
здесь выполняет обход диапазона индексов в аргументе, а внутренний (пере-  
дается функции tuple) выполняет обход полученных функцией последователь-  
ностей, одновременно извлекая из каждой по одному элементу. Если запустить   
этот пример, вы получите те же результаты, что и прежде.  
Больше всего в этом примере поражает широкое использование генераторов   
и итераторов. Аргументы, которые передаются функциям min и max, являют-  
ся выражениями-генераторами, которые заканчивают свои итерации еще до   
того, как вложенный генератор списков приступит к выполнению итераций.   
Кроме того, вложенные генераторы списков содержат два уровня отложенных   
операций – итератор, возвращаемый встроенной функцией range в Python 3.0,   
и выражение-генератор, которое передается, как аргумент функции tuple.  
В действительности, пока поток управления не достигнет квадратных скобок   
генератора списков, здесь не воспроизводится никаких результатов – именно   
в этом месте происходит запуск генераторов списков и выражений-генераторов.   
Чтобы эти версии функций возвращали не списки с результатами, а генерато-  
ры, вместо квадратных скобок следует использовать круглые скобки. Ниже   
приводится преобразованная версия нашей функции zip:  
# С использованием генераторов: (...)  
   
def myzip(\*seqs):  
 minlen = min(len(S) for S in seqs)  
 return (tuple(S[i] for S in seqs) for i in range(minlen))  
   
print(list(myzip(S1, S2)))  
Чтобы получить сразу весь список результатов, воспроизводимых этой версией   
функции, ее необходимо обернуть в вызов функции list, которая активирует   
работу генераторов и итераторов, воспроизводящих результаты. Поэкспери-  
ментируйте с этими версиями самостоятельно, чтобы лучше понять, как они   
действуют. Разработку других альтернатив я оставляю вам в качестве само-  
стоятельного упражнения (смотрите также врезку «Придется держать в уме:   
однократные итерации»).  
Придется держать в уме: однократные итерации  
В главе 14 мы видели, что некоторые встроенные функции (такие как   
map) возвращают итераторы, поддерживают возможность только одно-  
кратного обхода элементов и опустошаются по достижении конца ите-  
раций. Там же я обещал привести пример, насколько значимым может   
оказаться это обстоятельство на практике. Теперь, когда мы знаем об   
итерациях чуть больше, я готов выполнить свое обещание. Рассмотрим   
следующую альтернативную реализацию нашей функции zip, которая   
является адаптированной версией функции, взятой из руководства по   
языку Python:

582   
Глава 20. Итераторы и генераторы   
def myzip(\*args):  
 iters = map(iter, args)  
 while iters:  
 res = [next(i) for i in iters]  
 yield tuple(res)  
Так как в этой версии используются функции iter и next, она способна   
принимать любые итерируемые объекты. Обратите внимание, что нет   
никаких причин перехватывать исключение StopIteration, возбуждае-  
мое вызовом функции next(i) внутри генератора списков по исчерпании   
любого из аргументов-итераторов, – внутри функции-генератора это   
исключение дает тот же эффект, что и инструкция return. Инструкции   
while iters: вполне достаточно, чтобы организовать обход, когда функ-  
ции был передан хотя бы один аргумент, и избежать зацикливания   
в противном случае (генератор списков в этом случае всегда будет воз-  
вращать пустой список).  
Эта функция прекрасно действует в Python 2.6, например:  
>>> list(myzip(‘abc’, ‘lmnop’))  
[(‘a’, ‘l’), (‘b’, ‘m’), (‘c’, ‘n’)]  
Но она попадает в бесконечный цикл в Python 3.0, потому что функция   
map в версии 3.0 возвращает объект-итератор однократного пользования,   
а не список, как в версии 2.6. В версии 3.0, как только внутри цикла   
будет выполнен генератор списков, переменная iters навсегда останется   
пустой (и переменная res будет ссылаться на пустой список []). Чтобы   
обеспечить работоспособность функции в версии 3.0, нам необходимо   
с помощью встроенной функции list создать объект, который поддер-  
живает многократные итерации:  
def myzip(\*args):  
 iters = list(map(iter, args))  
 ...остальная часть функции осталась без изменений...  
Опробуйте эту версию самостоятельно, чтобы понять, как она действу-  
ет. Главный урок, который следует запомнить: обертывать вызовы   
функции map в вызовы функции list в версии 3.0 приходится не только   
для отображения списков результатов в интерактивном сеансе!  
Генерирование значений во встроенных типах и классах  
Наконец, хоть мы и сосредоточились в этом разделе на реализации собствен-  
ных генераторов значений, не забывайте, что многие встроенные типы ведут   
себя похожим образом, – как мы видели в главе 14, словари, например, имеют   
собственные итераторы, которые во время итераций воспроизводят ключи:  
>>> D = {‘a’:1, ‘b’:2, ‘c’:3}  
>>> x = iter(D)  
>>> next(x)  
‘a’  
>>> next(x)  
‘c’

Краткая сводка по синтаксису генераторов в 3.0   
583  
Подобно значениям, которые воспроизводятся нашими собственными генера-  
торами, ключи словарей можно обойти вручную или с помощью автоматизиро-  
ванных инструментов итераций, включая циклы for, функцию map, генераторы   
списков и других, с которыми мы встречались в главе 14:  
>>> for key in D:  
... print(key, D[key])  
...  
a 1  
c 3  
b 2  
Мы также видели, что при использовании итераторов файлов интерпретатор   
Python просто загружает строки из файла по мере необходимости:  
>>> for line in open(‘temp.txt’):  
... print(line, end=’’)  
...  
Tis but  
a flesh wound.  
Хотя итераторы встроенных типов воспроизводят значения определенно-  
го типа, тем не менее в них используются те же концепции, что и в наших   
выражениях-генераторах и функциях-генераторах. Инструменты итераций,   
такие как циклы for, принимают любые итерируемые объекты, как определяе-  
мые пользователем, так и встроенные.  
Хотя это и выходит за рамки данной главы, тем не менее замечу, что существу-  
ет возможность реализовать произвольные объекты-генераторы с помощью   
классов, которые поддерживают протокол итераций, и поэтому могут исполь-  
зоваться в циклах for и в других итерационных контекстах. Такие классы   
определяют специальный метод \_\_iter\_\_, вызываемый функцией iter и воз-  
вращающий объект-итератор, обладающий методом \_\_next\_\_, который вызы-  
вается встроенной функцией next (при этом метод \_\_getitem\_\_ также остается   
доступным, как крайнее средство обеспечения итераций через доступ к эле-  
ментам по индексу).  
Экземпляры таких классов считаются итерируемыми объектами и могут ис-  
пользоваться в циклах for и во всех остальных итерационных контекстах.   
Однако при использовании классов мы получаем доступ к более богатым воз-  
можностям реализации логики выполнения и структурирования данных, чем   
могут предложить другие конструкции генераторов.  
На этом история итераторов не заканчивается и нам еще предстоит узнать, как   
они могут быть связаны с классами. А пока нам придется отложить эту тему,   
пока мы не познакомимся с итераторами на базе классов в главе 29.  
Краткая сводка по синтаксису генераторов в 3.0  
В этой главе наше внимание было сосредоточено на генераторах списков и на   
объектах-генераторах, но не следует забывать о существовании еще двух форм   
генераторов: генераторов множеств и словарей, появившихся в Python 3.0. Мы   
встречались с ними в главах 5 и 8, а теперь, с учетом накопленных представ-  
лений о генераторах, мы в состоянии полностью охватить эти разновидности   
генераторов:

584   
Глава 20. Итераторы и генераторы   
 •  
В случае с множествами литеральная форма {1, 3, 2} эквивалентна вызову   
set([1, 3, 2]), а новый синтаксис генераторов множеств {f(x) for x in S if P(x)}   
напоминает синтаксис выражений-генераторов set(f(x) for x in S if P(x)), где   
f(x) – произвольное выражение.  
 •  
В случае со словарями новая конструкция генераторов словарей {key: val for   
(key, val) in zip(keys, vals)} действует точно так же, как dict(zip(keys, vals)),   
and {x: f(x) for x in items}, и напоминает выражение-генератор dict((x, f(x))   
for x in items).  
Ниже приводится краткая сводка по всем альтернативным генераторам в вер-  
сии 3.0. Последние две конструкции являются новыми и недоступны в вер-  
сии 2.6:  
>>> [x \* x for x in range(10)] # Генератор списков: конструирует список  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81] # подобно вызову list(generator expr)  
   
>>> (x \* x for x in range(10)) # Выражение-генератор: воспроизводит   
<generator object at 0x009E7328> # элементы. Скобки часто необязательны  
   
>>> {x \* x for x in range(10)} # Генератор множеств, новинка в 3.0  
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36} # {x, y} – литерал множества в 3.0  
   
>>> {x: x \* x for x in range(10)} # Генератор словарей, новинка в 3.0  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}  
Понимание генераторов множеств и словарей  
В некотором смысле, конструкции генераторов множеств и словарей являют-  
ся всего лишь синтаксическим подсластителем для выражений-генераторов,   
которые передаются конструкторам этих типов данных. Поскольку обе кон-  
струкции способны принимать любые итерируемые объекты, следующие гене-  
раторы являются вполне допустимыми:  
>>> {x \* x for x in range(10)} # Генератор  
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}  
   
>>> set(x \* x for x in range(10)) # Генератор и конструктор типа  
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}  
   
>>> {x: x \* x for x in range(10)}  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}  
   
>>> dict((x, x \* x) for x in range(10))  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}  
Как и в случае с генераторами списков, мы всегда можем конструировать объ-  
екты результата вручную. Ниже приводятся реализации, эквивалентные двум   
последним генераторам:  
>>> res = set()  
>>> for x in range(10): # Эквивалент генератора множеств  
... res.add(x \* x)  
...  
>>> res  
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}  
   
>>> res = {}

Краткая сводка по синтаксису генераторов в 3.0   
585  
>>> for x in range(10): # Эквивалент генератора словарей  
... res[x] = x \* x  
...  
>>> res  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}  
Обратите внимание, что хотя обе формы принимают итераторы, но они не вос-  
производят результаты по требованию – обе формы конструируют результат   
целиком. Если вам потребуется воспроизводить ключи и значения по запросу,   
используйте выражение-генератор:  
>>> G = ((x, x \* x) for x in range(10))  
>>> next(G)  
(0, 0)  
>>> next(G)  
(1, 1)  
Расширенный синтаксис   
генераторов множеств и словарей  
Как и генераторы списков, генераторы множеств и словарей поддерживают   
вложенные условные инструкции if, позволяющие отфильтровывать элемен-  
ты из результата, – следующие конструкции воспроизводят квадраты четных   
чисел (то есть чисел, которые делятся на 2 без остатка) в заданном диапазоне:  
>>> [x \* x for x in range(10) if x % 2 == 0] # Списки упорядочены  
[0, 4, 16, 36, 64]  
   
>>> {x \* x for x in range(10) if x % 2 == 0} # А множества - нет  
{0, 16, 4, 64, 36}  
   
>>> {x: x \* x for x in range(10) if x % 2 == 0} # Как и ключи словаря  
{0: 0, 8: 64, 2: 4, 4: 16, 6: 36}  
Вложенные циклы for также являются допустимыми, хотя неупорядоченная   
природа обоих типов, не допускающая появления дубликатов, может несколь-  
ко осложнять интерпретацию результатов:  
>>> [x + y for x in [1, 2, 3] for y in [4, 5, 6]] # Списки сохраняют дубликаты  
[5, 6, 7, 6, 7, 8, 7, 8, 9]  
   
>>> {x + y for x in [1, 2, 3] for y in [4, 5, 6]} # А множества - нет  
{8, 9, 5, 6, 7}  
   
>>> {x: y for x in [1, 2, 3] for y in [4, 5, 6]} # Как и ключи словарей  
{1: 6, 2: 6, 3: 6}  
Подобно генераторам списков, генераторы множеств и словарей способны вы-  
полнять итерации по итераторам любых типов – спискам, строкам, файлам,   
диапазонам целых чисел и любым другим объектам, поддерживающим про-  
токол итераций:  
>>> {x + y for x in ‘ab’ for y in ‘cd’}  
{‘bd’, ‘ac’, ‘ad’, ‘bc’}  
   
>>> {x + y: (ord(x), ord(y)) for x in ‘ab’ for y in ‘cd’}  
{‘bd’: (98, 100), ‘ac’: (97, 99), ‘ad’: (97, 100), ‘bc’: (98, 99)}

586   
Глава 20. Итераторы и генераторы   
>>> {k \* 2 for k in [‘spam’, ‘ham’, ‘sausage’] if k[0] == ‘s’}  
{‘sausagesausage’, ‘spamspam’}  
   
>>> {k.upper(): k \* 2 for k in [‘spam’, ‘ham’, ‘sausage’] if k[0] == ‘s’}  
{‘SAUSAGE’: ‘sausagesausage’, ‘SPAM’: ‘spamspam’}  
Поэкспериментируйте с этими инструментами самостоятельно, чтобы полнее   
усвоить принцип их действия. Они могут обладать или не обладать более вы-  
сокой производительностью по сравнению с генераторами или циклами for, но   
для такой оценки мы должны иметь возможность явно оценить их производи-  
тельность, что естественным образом ведет нас к следующему разделу.  
Хронометраж итерационных альтернатив  
В этой книге нам встретилось несколько итерационных альтернатив. Чтобы   
подвести итог, коротко проанализируем ситуацию, соединив все, что мы узна-  
ли об итерациях и функциях.  
Несколько раз я упоминал, что генераторы списков обладают более высокой   
скоростью выполнения, чем циклы for, а скорость работы функции map мо-  
жет быть выше или ниже, в зависимости от конкретной решаемой задачи.   
Выражения-генераторы, рассматривавшиеся в предыдущих разделах, обычно   
немного медленнее, чем генераторы списков, но при этом они минимизируют   
требования к объему используемой памяти.  
Все это справедливо на сегодняшний день, но относительная производитель-  
ность может измениться со временем, так как интерпретатор Python постоянно   
оптимизируется. Если вам захочется проверить это самим, попробуйте запу-  
стить следующий сценарий на своем компьютере, со своей версией интерпре-  
татора.  
Модуль time  
К счастью, Python существенно упрощает хронометраж выполнения про-  
Python существенно упрощает хронометраж выполнения про-  
 существенно упрощает хронометраж выполнения про-  
граммного кода. Чтобы увидеть, как различные варианты выполнения итера-  
ций выстраиваются по производительности, рассмотрим простую, но достаточ-  
но универсальную утилиту timer, которую мы сохраним в файле модуля, чтобы   
потом ее можно было использовать в различных программах:  
# Файл mytimer.py  
   
import time  
reps = 1000  
repslist = range(reps)  
   
def timer(func, \*pargs, \*\*kargs):  
 start = time.clock()  
 for i in repslist:  
 ret = func(\*pargs, \*\*kargs)  
 elapsed = time.clock() - start  
 return (elapsed, ret)   
Функционально эта утилита способна вызывать любую функцию с любым ко-  
личеством позиционных и именованных аргументов. Она фиксирует началь-  
ное время, вызывает функцию фиксированное число раз и вычитает время на-  
чала из времени конца. Важно отметить следующие обстоятельства:

Хронометраж итерационных альтернатив   
587  
 •  
Модуль time из стандартной библиотеки языка Python позволяет получить   
текущее время с точностью, зависящей от платформы. В �indows, как   
утверждается, этот модуль позволяет получить время с точностью до ми-  
кросекунды и, следовательно, обеспечивает высокую точность.   
 •  
Вызов функции range был вынесен за пределы цикла измерения времени,   
благодаря чему время конструирования диапазона не накладывается на по-  
лучаемые результаты в Python 2.6. В Python 3.0 функция range возвращает   
итератор, поэтому в версии 3.0 данный шаг можно считать излишним (хотя   
он и не мешает).  
 •  
Счетчик reps оформлен как глобальная переменная, благодаря чему она   
может изменяться импортирующим модулем при необходимости: mytimer.   
reps = N.  
По окончании измерений утилита возвращает общее время выполнения всех   
вызовов функции внутри кортежа, наряду с последним значением, получен-  
ным от исследуемой функции, чтобы вызывающая программа могла проверить   
результат ее деятельности.  
Глядя вперед, можно отметить, что так как эта функция находится в отдель-  
ном файле модуля, она может рассматриваться как удобный инструмент обще-  
го пользования, доступный для импортирования любой программой. Подроб-  
нее с модулями и с операцией импортирования вы познакомитесь в следующей   
части книги, но вы знаете уже достаточно, чтобы понять этот программный   
код, – просто импортируйте модуль и вызовите функцию (если вам необходимо   
освежить знания об атрибутах модулей, обращайтесь к главе 3).  
Сценарий хронометража  
Теперь можно приступать к измерению производительности инструментов ите-  
раций. Запустите следующий сценарий – он использует модуль mytimer, кото-  
рый мы только что написали, для измерения относительной скорости выполне-  
ния различных приемов конструирования списков, изученных нами:  
# Файл timeseqs.py  
   
import sys, mytimer # Импортирует функцию timer   
reps = 10000  
repslist = range(reps) # Вызов функции range вынесен за пределы цикла в 2.6  
   
def forLoop():  
 res = []  
 for x in repslist:  
 res.append(abs(x))  
 return res  
   
def listComp():  
 return [abs(x) for x in repslist]  
   
def mapCall():  
 return list(map(abs, repslist)) # Вызов list необходим только в 3.0  
   
def genExpr():  
 return list(abs(x) for x in repslist) # Функция list вынуждает   
 # вернуть сразу все результаты  
def genFunc():

588   
Глава 20. Итераторы и генераторы   
 def gen():  
 for x in repslist:  
 yield abs(x)  
 return list(gen())  
   
print(sys.version)  
for test in (forLoop, listComp, mapCall, genExpr, genFunc):  
 elapsed, result = mytimer.timer(test)  
 print (‘-’ \* 33)  
 print (‘%-9s: %.5f => [%s...%s]’ %  
 (test.\_\_name\_\_, elapsed, result[0], result[-1]))  
Этот сценарий тестирует все пять альтернативных способов создания списков и,   
как видно из листинга, выполняет по 10 миллионов итераций каждым из спо-  
собов, то есть каждый из тестов создает список из 10 000 элементов 1 000 раз.  
Обратите внимание, как выражения и функции передаются встроенной функ-  
ции list, чтобы вынудить их выдать все значения, – если бы этого не было сде-  
лано, мы бы просто создали генератор, который не выполняет никакой работы.   
В Python 3.0 (только) мы также должны передать функции list результат вызо-  
ва функции map, поскольку в этой версии она возвращает итератор. Кроме того,   
заметьте, как программный код в самом конце сценария выполняет обход кор-  
тежа из четырех функций и выводит значение атрибута \_\_name\_\_ для каждой из   
них: это встроенный атрибут, который возвращает имя функции.  
Результаты хронометража  
Когда я запустил сценарий из предыдущего раздела в �indows Vista на своем   
ноутбуке, где установлен Python 3.0, я обнаружил, что функция map оказалась   
немного быстрее, чем генератор списков� оба они оказались существенно бы-  
стрее эквивалентной инструкции цикла for, а функция-генератор и выражение-  
генератор заняли промежуточное положение:  
C:\misc> c:\python30\python timeseqs.py  
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]  
---------------------------------  
forLoop : 2.64441 => [0...9999]  
---------------------------------  
listComp : 1.60110 => [0...9999]  
---------------------------------  
mapCall : 1.41977 => [0...9999]  
---------------------------------  
genExpr : 2.21758 => [0...9999]  
---------------------------------  
genFunc : 2.18696 => [0...9999]  
Если внимательно изучить программный код и полученные результаты,   
можно заметить, что выражения-генераторы выполняются медленнее, чем   
генераторы списков. Несмотря на то что обертывание выражения-генератора   
в вызов функции list превращает его в функциональный эквивалент генера-  
тора списка в квадратных скобках, тем не менее результаты показывают, что   
внутренние реализации этих двух видов выражений отличаются (даже если   
учесть время, затрачиваемое на вызов функции list в тесте производительно-  
сти выражения-генератора):

Хронометраж итерационных альтернатив   
589  
return [abs(x) for x in range(size)] # 1.6 секунды  
return list(abs(x) for x in range(size)) # 2.2 секунды: качественно отличаются  
Интересно отметить, что когда я запускал этот сценарий в �indows XP, где   
установлен Python 2.5, при подготовке предыдущего издания книги, были   
получены похожие результаты – генератор списков оказался почти в два раза   
быстрее эквивалентного цикла for, а функция map оказалась немного быстрее   
генератора списков при отображении встроенной функции abs (возвращает аб-  
солютное значение). Тогда я не тестировал функцию-генератор и формат выво-  
да результатов был не такой симпатичный:  
2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)]  
forStatement => 6.10899996758  
listComprehension => 3.51499986649  
mapFunction => 2.73399996758  
generatorExpression => 4.11600017548  
То обстоятельство, что по результатам хронометража тесты под управлением   
Python 2.5 выполняются более чем в два раза медленнее, обусловлено тем, что   
более свежие тесты я выполнял на более быстром ноутбуке, а вовсе не улучше-  
ниями производительности в Python 3.0. В действительности, производитель-  
Python 3.0. В действительности, производитель-  
 3.0. В действительности, производитель-  
ность тестов в версии 2.6 на том же самом компьютере оказывается чуть выше,   
чем в версии 3.0, если из теста функции map убрать вызов функции list, чтобы   
избежать двойного создания списка с результатами (попробуйте сами выпол-  
нить тестирование, чтобы проверить правоту моих слов).  
Но вот как изменилось положение дел, когда сценарий был изменен так, чтобы   
он выполнял настоящую операцию, такую как сложение, вместо вызова три-  
виальной встроенной функции abs (опущенные части сценария остались без   
изменений):   
# Файл timeseqs.py  
...  
...  
def forLoop():  
 res = []  
 for x in repslist:  
 res.append(x + 10)  
 return res  
   
def listComp():  
 return [x + 10 for x in repslist]  
   
def mapCall():  
 return list(map((lambda x: x + 10), repslist)) # Вызов list необходим   
 # только в 3.0  
def genExpr():  
 return list(x + 10 for x in repslist) # Вызов list необходим в 2.6 + 3.0  
   
def genFunc():  
 def gen():  
 for x in repslist:  
 yield x + 10  
 return list(gen())  
...  
...

590   
Глава 20. Итераторы и генераторы   
Теперь присутствие вызова пользовательской функции сделало вызов map бо-  
лее медленным, чем цикл for, несмотря на то, что инструкция цикла содержит   
больше программного кода. В Python 3.0:  
C:\misc> c:\python30\python timeseqs.py  
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]  
---------------------------------  
forLoop : 2.60754 => [10...10009]  
---------------------------------  
listComp : 1.57585 => [10...10009]  
---------------------------------  
mapCall : 3.10276 => [10...10009]  
---------------------------------  
genExpr : 1.96482 => [10...10009]  
---------------------------------  
genFunc : 1.95340 => [10...10009]  
Результаты, полученные в Python 2.5 на более медленном компьютере при под-  
Python 2.5 на более медленном компьютере при под-  
 2.5 на более медленном компьютере при под-  
готовке предыдущего издания, демонстрируют ту же тенденцию, хотя произ-  
водительность и оказывается в два раза ниже, что обусловлено различиями   
в быстродействии компьютеров, на которых выполнялось тестирование:  
2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)]  
forStatement => 5.25699996948  
listComprehension => 2.68400001526  
mapFunction => 5.96900010109  
generatorExpression => 3.37400007248  
Так как внутренние механизмы интерпретатора сильно оптимизированы, ана-  
лиз производительности, как в данном случае, становится очень непростым   
делом. В действительности невозможно заранее утверждать, какой метод луч-  
ше, – лучшее, что можно сделать, это провести хронометраж своего программ-  
ного кода, на своем компьютере, со своей версией Python. В этом случае все,   
что можно сказать наверняка, – это то, что в данной версии Python использова-  
Python использова-  
 использова-  
ние пользовательской функции в вызове map может привести к снижению про-  
изводительности по крайней мере в 2 раза, и что в этом испытании генератор   
списков оказался самым быстрым.  
Однако, как уже говорилось ранее, производительность не должна быть глав-  
ной целью при создании программ на языке Python. Первый шаг на пути   
к оптимизации программ на языке Python – это отказаться от всякой опти-  
Python – это отказаться от всякой опти-  
 – это отказаться от всякой опти-  
мизации! Основное внимание должно уделяться удобочитаемости и просто-  
те программного кода, и только потом код можно будет оптимизировать, если   
это действительно необходимо. Вполне возможно, что все пять вариантов обла-  
дают достаточной скоростью обработки имеющихся наборов данных, – в этом   
случае основной целью должна быть ясность программного кода.  
Альтернативные реализации модуля хронометража  
Модуль хронометража из предыдущего раздела вполне работоспособен, но   
в некоторых отношениях он слишком примитивен:  
 •  
В нем всегда используется функция time.clock. В операционной системе   
�indows она является лучшим выбором, однако на некоторых платформах   
в системе UNIX более высокую точность можно получить с помощью функ-  
UNIX более высокую точность можно получить с помощью функ-  
 более высокую точность можно получить с помощью функ-  
ции time.time.

Хронометраж итерационных альтернатив   
591  
 •  
Для изменения количества повторений требуется изменять глобальную   
переменную модуля – не самое идеальное решение, если функция timer им-  
портируется и одновременно используется в нескольких модулях.  
 •  
Функция timer выполняет тестовую функцию большое число раз. Чтобы   
учесть случайные флуктуации, вызванные различными уровнями нагруз-  
ки на систему, можно было бы отбирать наилучшие результаты из серии   
тестов вместо того, чтобы рассчитывать общее время выполнения.  
Ниже приводится более сложная, альтернативная реализация модуля mytimer,   
в которой учтены все три пожелания: производится выбор функции опреде-  
ления времени в зависимости от платформы, счетчик повторений передается   
функции timer в виде именованного аргумента \_reps и дополнительно предо-  
ставляется функция хронометража, возвращающая лучший результат из се-  
рии по N испытаниям:  
# Файл mytimer.py (2.6 и 3.0)  
   
“””  
timer(spam, 1, 2, a=3, b=4, \_reps=1000) вызывает и измеряет время работы функции   
spam(1, 2, a=3) \_reps раз, и возвращает общее время, затраченное на все вызовы,   
с результатом вызова испытуемой функции;  
   
best(spam, 1, 2, a=3, b=4, \_reps=50) многократно вызывает функцию timer, чтобы   
исключить влияние флуктуаций в нагрузке на систему, и возвращает лучший результат из   
серии по \_reps испытаниям  
“””  
   
import time, sys  
if sys.platform[:3] == ‘win’:  
 timefunc = time.clock # В Windows использовать time.clock   
else:  
 timefunc = time.time # На некоторых платформах Unix дает   
 # лучшее разрешение  
   
def trace(\*args): pass # Заглушка: вывод аргументов  
   
def timer(func, \*pargs, \*\*kargs):  
 \_reps = kargs.pop(‘\_reps’, 1000) # Полученное число повторов   
 # или значение по умолчанию  
 trace(func, pargs, kargs, \_reps)  
 repslist = range(\_reps) # Вызов range вынесен за пределы   
 # цикла for для версии 2.6  
 start = timefunc()  
 for i in repslist:  
 ret = func(\*pargs, \*\*kargs)  
 elapsed = timefunc() - start  
 return (elapsed, ret)  
   
def best(func, \*pargs, \*\*kargs):  
 \_reps = kargs.pop(‘\_reps’, 50)  
 best = 2 \*\* 32  
 for i in range(\_reps):  
 (time, ret) = timer(func, \*pargs, \_reps=1, \*\*kargs)  
 if time < best: best = time  
 return (best, ret)  
Строка документирования в самом начале модуля описывает порядок его ис-  
пользования. Здесь используется метод pop словарей, чтобы удалить аргу-

592   
Глава 20. Итераторы и генераторы   
мент \_reps из списка аргументов, предназначенных для испытуемой функции,   
и подставить значение по умолчанию в случае необходимости. Кроме того,   
модуль позволяет выводить значения аргументов на этапе отладки, для чего   
достаточно заменить вызов функции trace вызовом функции print. Чтобы про-  
верить работу нового модуля в Python 3.0 или 2.6, измените тестовый сцена-  
Python 3.0 или 2.6, измените тестовый сцена-  
 3.0 или 2.6, измените тестовый сцена-  
рий, как показано ниже (опущенный программный код в функциях остался   
таким же, как в предыдущей версии, где в каждом испытании используется   
операция x + 1):  
# Файл timeseqs.py  
   
import sys, mytimer  
reps = 10000  
repslist = range(reps)  
   
def forLoop(): ...  
   
def listComp(): ...  
   
def mapCall(): ...  
   
def genExpr(): ...  
   
def genFunc(): ...  
   
print(sys.version)  
for tester in (mytimer.timer, mytimer.best):  
 print(‘<%s>’ % tester.\_\_name\_\_)  
 for test in (forLoop, listComp, mapCall, genExpr, genFunc):  
 elapsed, result = tester(test)  
 print (‘-’ \* 35)  
 print (‘%-9s: %.5f => [%s...%s]’ %  
 (test.\_\_name\_\_, elapsed, result[0], result[-1]))  
При запуске сценария в Python 3.0 результаты хронометража остались прак-  
Python 3.0 результаты хронометража остались прак-  
 3.0 результаты хронометража остались прак-  
тически такими же. И относительно такими же – при выявлении наилучших   
показателей. Многократный запуск тестирования, похоже, позволяет отфиль-  
тровывать флуктуации, вызванные изменением нагрузки на систему, так же   
хорошо, как и прием выбора лучшего значения из серии, однако последний   
лучше подходит для тестирования функций с продолжительным временем ра-  
боты. На моем компьютере были получены следующие результаты:  
C:\misc> c:\python30\python timeseqs.py  
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]  
<timer>  
-----------------------------------  
forLoop : 2.35371 => [10...10009]  
-----------------------------------  
listComp : 1.29640 => [10...10009]  
-----------------------------------  
mapCall : 3.16556 => [10...10009]  
-----------------------------------  
genExpr : 1.97440 => [10...10009]  
-----------------------------------  
genFunc : 1.95072 => [10...10009]  
<best>  
-----------------------------------  
forLoop : 0.00193 => [10...10009]

Хронометраж итерационных альтернатив   
593  
-----------------------------------  
listComp : 0.00124 => [10...10009]  
-----------------------------------  
mapCall : 0.00268 => [10...10009]  
-----------------------------------  
genExpr : 0.00164 => [10...10009]  
-----------------------------------  
genFunc : 0.00165 => [10...10009]  
Значения времени, которые вернула функция best, оказались очень маленьки-  
ми, но они могут существенно возрасти при тестировании программы, которая   
выполняет множество итераций по большим массивам данных. В большинстве   
случаев, по крайней мере в смысле относительной производительности, гене-  
раторы списков показывают наилучшую производительность – функция map   
оказывается чуть быстрее лишь при использовании встроенных функций.  
Использование аргументов,   
которые могут передаваться только по именам  
Чтобы упростить реализацию модуля mytimer, мы можем также использовать   
аргументы,  которые  могут  передаваться  только  по  именам, появившиеся   
в Python 3.0. Как мы узнали в главе 19, аргументы, которые могут передавать-  
Python 3.0. Как мы узнали в главе 19, аргументы, которые могут передавать-  
 3.0. Как мы узнали в главе 19, аргументы, которые могут передавать-  
ся только по именам, идеально подходят для передачи параметров настройки,   
таких как аргумент \_reps. Они должны указываться в заголовках функций по-  
сле формы представления аргументов \* и перед формой \*\*, а в вызовах функ-  
ций они должны передаваться в виде именованных аргументов и указывать-  
ся перед формой \*\*, если она используется. Ниже приводится альтернативная   
версия предыдущего модуля, где используются аргументы, которые могут пе-  
редаваться только по именам. Реализация модуля стала выглядеть проще, но   
теперь он может работать только под управлением Python 3.X:  
# Файл mytimer.py (только для 3.X)  
   
“””  
Вместо формы \*\* и метода pop словарей используются аргументы, которые могут   
передаваться только по именам, появившиеся в версии 3.0.  
В версии 3.0 нет необходимости выносить вызов range() за пределы цикла, так как эта   
функция возвращает генератор, а не список  
“””  
   
import time, sys  
trace = lambda \*args: None # or print  
timefunc = time.clock if sys.platform == ‘win32’ else time.time  
   
def timer(func, \*pargs, \_reps=1000, \*\*kargs):  
 trace(func, pargs, kargs, \_reps)  
 start = timefunc()  
 for i in range(\_reps):  
 ret = func(\*pargs, \*\*kargs)  
 elapsed = timefunc() - start  
 return (elapsed, ret)  
   
def best(func, \*pargs, \_reps=50, \*\*kargs):  
 best = 2 \*\* 32  
 for i in range(\_reps):  
 (time, ret) = timer(func, \*pargs, \_reps=1, \*\*kargs)  
 if time < best: best = time

594   
Глава 20. Итераторы и генераторы   
 return (best, ret)  
Эта версия модуля используется точно так же и возвращает результаты, иден-  
тичные предыдущей версии, не считая незначительных расхождений, харак-  
терных для разных испытательных серий:  
C:\misc> c:\python30\python timeseqs.py  
...результаты те же, что и прежде...  
Для разнообразия мы можем также опробовать данную версию модуля в ин-  
терактивном сеансе. Это универсальный инструмент – он никак не зависит от   
сценария хронометража:  
C:\misc> c:\python30\python  
>>> from mytimer import timer, best  
>>>  
>>> def power(X, Y): return X \*\* Y # Испытуемая функция  
...  
>>> timer(power, 2, 32) # Общее время, последний результат  
(0.002625403507987747, 4294967296)  
>>> timer(power, 2, 32, \_reps=1000000) # Переопределить количество повторов  
(1.1822605247314932, 4294967296)  
>>> timer(power, 2, 100000)[0] # 2 \*\* 100000: общее время для 1000 повторов  
2.2496919999608878  
   
>>> best(power, 2, 32) # Лучшее время, последний результат  
(5.58730229727189e-06, 4294967296)  
>>> best(power, 2, 100000)[0] # 2 \*\* 100000: лучшее время  
0.0019937589833460834  
>>> best(power, 2, 100000, \_reps=500)[0] # Переопределить количество повторов  
0.0019845399345541637  
Для таких тривиальных функций, как та, что использовалась в этом интерак-  
тивном сеансе, затраты времени на выполнение программного кода хрономе-  
тража, вероятно, сопоставимы с затратами на выполнение самой испытуемой   
функции, поэтому не следует рассматривать результаты, как некоторый абсо-  
лютный показатель (здесь мы измеряли не только производительность опера-  
ции X \*\* Y). Результаты хронометража смогут помочь вам судить об относитель-  
ной производительности альтернативных решений и могут оказаться более   
осмысленными при тестировании длительных операций, таких как показано   
ниже, – операция возведения числа 2 в степень 1000000 выполняется на поря-  
док (в 10 раз) дольше, чем предыдущая операция 2\*\*100000:  
>>> timer(power, 2, 1000000, \_reps=1)[0] # 2 \*\* 1000000: общее время  
0.088112804839710179  
>>> timer(power, 2, 1000000, \_reps=10)[0]  
0.40922470593329763  
   
>>> best(power, 2, 1000000, \_reps=1)[0] # 2 \*\* 1000000: лучшее время  
0.086550036387279761  
>>> best(power, 2, 1000000, \_reps=10)[0] # Иногда 10 ничуть не хуже 50  
0.029616752967200455  
>>> best(power, 2, 1000000, \_reps=50)[0] # Лучшее разрешение  
0.029486918030102061  
Напомню еще раз: хотя здесь получаются достаточно маленькие времена, они   
могут увеличиваться существенно в программах, часто вычисляющих степень   
числа.

Хронометраж итерационных альтернатив   
595  
Подробнее об аргументах, которые можно передавать только по именам, по-  
явившихся в Python 3.0, рассказывается в главе 19, – они могут упростить   
реализацию настраиваемых инструментов, таких как наши функции хроно-  
метража, но они не могут использоваться в Python 2.X. Если вам нужно срав-  
Python 2.X. Если вам нужно срав-  
 2.X. Если вам нужно срав-  
X. Если вам нужно срав-  
. Если вам нужно срав-  
нить производительность 2.X и 3.X, например, или обеспечить возможность   
тестирования в любой из двух версий Python, лучшим выбором для вас будет   
предыдущая версия. Если вы используете Python 2.6, предыдущая версия мо-  
Python 2.6, предыдущая версия мо-  
 2.6, предыдущая версия мо-  
дуля хронометража будет действовать точно так же, как показано в листинге   
интерактивного сеанса выше.  
Другие предложения  
Чтобы еще глубже вникнуть в ситуацию, попробуйте изменить количество   
повторений в начале сценария или рассмотрите возможность использования   
новейшего модуля timeit, который автоматизирует хронометраж кода и позво-  
ляет избежать проблем, связанных с используемой платформой. Порядок его   
использования описывается в руководстве по языку Python.  
Кроме того, обратите внимание на модуль profile из стандартной библиотеки,   
где вы найдете полные исходные тексты инструментов профилирования про-  
граммного кода, – поближе с ними мы познакомимся в главе 35, когда будем   
рассматривать инструменты, используемые при разработке крупных проек-  
тов. Вообще говоря, профилирование программного кода с целью выявления   
узких мест должно проводиться перед проведением хронометража, который   
мы рассматривали здесь.  
Кроме того, было бы интересно поэкспериментировать с новым методом str.  
format, появившимся в Python 2.6 и 3.0, и использовать его вместо оператора %   
форматирования (который в будущем, возможно, будет удален!), заменив стро-  
ки вывода результатов в сценарии хронометража следующими инструкциями:  
print(‘<%s>’ % tester.\_\_name\_\_) # Выражение  
   
print(‘<{0}>’.format(tester.\_\_name\_\_)) # Вызов метода  
   
 print (‘%-9s: %.5f => [%s...%s]’ %  
 (test.\_\_name\_\_, elapsed, result[0], result[-1]))  
   
 print(‘{0:<9}: {1:.5f} => [{2}...{3}]’.format(  
 test.\_\_name\_\_, elapsed, result[0], result[-1]))  
Оцените различия между этими двумя подходами самостоятельно.  
Если вам интересно, можете также попробовать изменить или написать новый   
сценарий хронометража для измерения производительности генераторов мно-  
жеств и словарей в версии 3.0, о которых рассказывалось в этой главе, и экви-  
валентных им циклов for. Поскольку эти конструкции в программах на языке   
Python встречаются значительно реже, чем генераторы списков, я оставляю   
решение этой задачи в качестве самостоятельного упражнения (и, пожалуй-  
ста, не заключайте никаких пари заранее...).  
И напоследок, сохраните модуль хронометража, который мы написали здесь,   
для обращения к нему в будущем, – мы еще будем использовать его для измере-  
ния производительности альтернативных операций вычисления квадратного   
корня в упражнениях, в конце этой главы. Для тех, кому это интересно, от-  
мечу, что мы также будем использовать этот прием для сравнительного хроно-

596   
Глава 20. Итераторы и генераторы   
метража производительности генераторов словарей и циклов for в интерактив-  
ном сеансе.  
Типичные ошибки при работе с функциями  
Теперь, когда мы закончили изучение функций, рассмотрим некоторые наи-  
более распространенные ошибки. При работе с функциями вас поджидают под-  
водные камни, о которых вы можете не догадываться. Они не всегда видны,   
некоторые из них исчезли в последних версиях, но большая часть оставшихся   
продолжает ставить в тупик начинающих программистов.  
Локальные имена определяются статически  
Как известно, имена, которым выполняется присваивание внутри функции,   
по умолчанию рассматриваются как локальные – они располагаются в области   
видимости функции и существуют только во время работы функции. Но я еще   
не говорил, что локальные переменные определяются статически, во время   
компиляции программного кода в инструкции def, а не в соответствии с опера-  
циями присваивания, производимыми во время выполнения. Эта особенность   
становится причиной появления самых причудливых сообщений в группе но-  
востей, получаемых от начинающих программистов.  
Обычно, если внутри функции имени не присваивается какое-либо значение,   
поиск его будет производиться в области видимости объемлющего модуля:   
>>> X = 99  
>>> def selector(): # Переменная X используется, но ей ничего не присваивается  
... print(X) # Переменная X будет найдена в глобальной области вид-ти  
...  
>>> selector()  
99  
В этом фрагменте переменная X внутри функции определяется как переменная   
X модуля. Но посмотрите, что произойдет, если добавить инструкцию присваи-  
вания переменной X после ее использования:  
>>> def selector():  
... print(X) # Переменная еще не существует!  
... X = 88 # X классифицируется как локальная переменная  
... # То же самое происходит при “import X”, “def X”...  
>>> selector()  
Traceback (most recent call last):  
...текст сообщения об ошибке опущен...  
UnboundLocalError: local variable ‘X’ referenced before assignment  
Было получено сообщение о том, что переменная не определена, но причина его   
появления не очевидна. Этот программный код компилируется интерпретато-  
ром во время ввода в интерактивной оболочке или во время импорта модуля. Во   
время компиляции Python обнаруживает операцию присваивания переменной   
X и делает вывод, что X – это локальное имя везде в теле функции. Но во время   
выполнения функции, из-за того, что к моменту вызова инструкции print опе-  
рация присваивания еще не производилась, интерпретатор сообщает о том, что   
имя не определено. Согласно этому правилу использования имен, он говорит,

Типичные ошибки при работе с функциями   
597  
что обращение к локальной переменной X произведено до того, как ей было при-  
своено значение. Фактически любая операция присваивания внутри функции   
создает локальное имя. Операция импортирования, =, вложенные инструкции   
def, вложенные определения классов и так далее – все трактуются именно та-  
ким образом.  
Проблема возникает из-за того, что операция присваивания делает имена ло-  
кальными для всей функции, а не только для той ее части, которая следует за   
инструкцией присваивания. На самом деле предыдущий пример далеко неод-  
нозначен: что имелось в виду – требовалось вывести глобальную переменную X   
и затем создать локальную переменную или это просто ошибка программиста?   
Так как Python интерпретирует имя X как локальное во всей функции, то это   
ошибка – если вы действительно хотите вывести значение глобальной перемен-  
ной X, объявите ее глобальной с помощью инструкции global:  
>>> def selector():  
... global X # Принудительное объявление X глобальным (везде)  
... print(X)  
... X = 88  
...  
>>> selector()  
99  
При этом следует помнить, что в этом случае операция присваивания изменит   
глобальную переменную X, а не локальную. Внутри функции можно использо-  
вать как локальную, так и глобальную версии одного и того же имени. Если вы   
действительно предполагаете вывести значение глобальной переменной, а за-  
тем присвоить значение локальной версии того же самого имени, импортируй-  
те вмещающий модуль и обращайтесь к глобальной переменной как к атрибуту   
модуля:  
>>> X = 99  
>>> def selector():  
... import \_\_main\_\_ # Импортировать вмещающий модуль  
... print(\_\_main\_\_.X) # Квалифицированное обращение к глобальной версии   
имени  
... X = 88 # Неквалифицированное локальное имя X   
... print(X) # Вывести локальную версию имени  
...  
>>> selector()  
99  
88  
Обращение по квалифицированному имени (часть .X) приводит к извлечению   
значения из пространства имен объекта. Пространством имен интерактивной   
оболочки является модуль с именем \_\_main\_\_, поэтому при обращении по имени   
\_\_main\_\_.X извлекается глобальная версия X. Если что-то вам показалось непо-  
нятным, прочитайте главу 17.  
Положение дел с локальными переменными в последних версиях Python не-  
сколько улучшилось, потому что в данном случае выводится более определен-  
ное сообщение об ошибке «обращение к локальной переменной до присваива-  
ния», которое показано в листинге примера (теперь оно используется вместо   
более расплывчатого сообщения об ошибке, связанной с именем), впрочем, этот   
вид ошибки все еще встречается.

598   
Глава 20. Итераторы и генераторы   
Значения по умолчанию и изменяемые объекты  
Значения по умолчанию для аргументов функции вычисляются и запомина-  
ются в момент выполнения инструкции def, а не при вызове функции. Внутрен-  
няя реализация Python сохраняет по одному объекту для каждого аргумента   
со значением по умолчанию, присоединенного к функции.  
Вычисление значений по умолчанию в момент определения функции – это   
чаще именно то, что вам требуется� это позволяет, в случае необходимости, со-  
хранять значения из объемлющей области видимости. Но так как значения по   
умолчанию сохраняются между вызовами функции, следует быть вниматель-  
ным при воздействии на изменяемые значения по умолчанию. Например, сле-  
дующая функция использует пустой список в качестве значения по умолчанию   
своего аргумента, а затем изменяет его при каждом вызове:  
>>> def saver(x=[]): # Объект списка сохраняется  
... x.append(1) # При каждом вызове изменяется один и тот же объект!  
... print(x)  
...  
>>> saver([2]) # Значение по умолчанию не используется  
[2, 1]  
>>> saver() # Используется значение по умолчанию  
[1]  
>>> saver() # Список растет при каждом вызове!  
[1, 1]  
>>> saver()  
[1, 1, 1]  
Некоторые воспринимают такое поведение как достоинство – изменяемые ар-  
гументы по умолчанию сохраняют свое состояние между вызовами функции,   
поэтому они могут играть роль, подобную роли статических локальных пере-  
менных в языке C. В некотором смысле они ведут себя как глобальные пере-  
менные за исключением того, что их имена являются локальными по отноше-  
нию к функциям, вследствие чего исключается конфликт имен с переменны-  
ми, определенными в другом месте.  
Для большинства же это выглядит как недостаток, особенно для тех, кто впер-  
вые сталкивается с этой особенностью. В языке Python существует лучший   
способ сохранения состояния между вызовами функций (например, за счет ис-  
пользования классов, которые будут рассматриваться в шестой части книги).  
Кроме того, такое поведение аргументов по умолчанию сложно запомнить (и   
вообще понять). Они могут изменяться с течением времени. В предыдущем   
примере для значения по умолчанию существует единственный объект спи-  
ска – тот, что был создан в момент выполнения инструкции def. При каждом   
обращении к функции не будет создаваться новый список, поэтому он будет   
расти с каждым новым вызовом – он не опустошается при каждом вызове.  
Если такое поведение является неприемлемым, можно просто создавать копию   
аргумента по умолчанию в начале тела функции или переместить выражение,   
возвращающее значение по умолчанию, в тело функции. Поскольку в этом   
случае значение по умолчанию будет находиться в программном коде, кото-  
рый выполняется при каждом вызове функции, вы всякий раз будете получать   
новый объект:  
>>> def saver(x=None):  
... if x is None: # Аргумент отсутствует?

Типичные ошибки при работе с функциями   
599  
... x = [] # Создать новый список  
... x.append(1) # Изменить объект списка  
... print(x)  
...  
>>> saver([2])  
[2, 1]  
>>> saver() # Список больше не растет  
[1]  
>>> saver()  
[1]  
Между прочим, инструкцию if в этом примере в большинстве случаев можно   
было бы заменить выражением x = x or [], где используется тот факт, что опера-  
тор or в языке Python возвращает один из двух объектов: если аргумент отсут-  
ствует, имя x получит значение по умолчанию None, и тогда оператор or вернет   
новый пустой список справа от него.  
Однако это не совсем одно и то же. Если функции будет передан пустой список,   
оператор вернет вновь созданный список вместо полученного в аргументе, как   
это делает инструкция if. (Выражение примет вид [] or [], которое возвраща-  
ет новый пустой список справа, – вернитесь к разделу «Проверка истинности»   
в главе 12, если вам не понятно, почему так происходит.) В разных программах   
могут предъявляться разные требования к такому поведению.  
На сегодняшний день имеется еще один, менее запутанный, способ получения   
изменяемых значений по умолчанию, который заключается в использовании   
атрибутов функций, о которых рассказывается в главе 19:  
>>> def saver():  
... saver.x.append(1)  
... print(saver.x)  
...  
>>> saver.x = []  
>>> saver()  
[1]  
>>> saver()  
[1, 1]  
>>> saver()  
[1, 1, 1]  
Имя функции является глобальным для самой функции, но объявлять его гло-  
бальным внутри функции не требуется, так как мы фактически не изменяем   
его внутри функции. Такой прием использования атрибутов, присоединенных   
к объекту функции, оказывается более явным (и, вероятно, менее таинствен-  
ным).  
Функции, не возвращающие результат  
В языке Python функции могут не иметь инструкцию return (или yield). Когда   
функция не возвращает управление явно, выход из нее происходит, когда по-  
ток управления достигает конца тела функции. С технической точки зрения   
все функции возвращают некоторое значение – в отсутствие инструкции return   
функция автоматически возвращает объект None:  
>>> def proc(x):  
... print(x) # Нет возвращаемого значения, возвращается None   
...

600   
Глава 20. Итераторы и генераторы   
>>> x = proc(‘testing 123...’)  
testing 123...  
>>> print(x)  
None  
Такие функции, как эта, не имеющие инструкции return, представляют собой   
эквивалент того, что в других языках программирования называется «проце-  
дурами». Как правило, они вызываются как инструкции, а возвращаемое зна-  
чение None игнорируется, поскольку они выполняют свою работу, не вычисляя   
результат.  
Об этом следует помнить, потому что интерпретатор ничего не сообщит вам,   
если вы попытаетесь присвоить результат функции, которая ничего не возвра-  
щает. Например, присваивание результата метода списков append не вызыва-  
ет появление ошибки, но при этом вы получите объект None, а не обновленный   
список:  
>>> list = [1, 2, 3]  
>>> list = list.append(4) # метод append – это “процедура”  
>>> print(list) # метод append изменяет сам список  
None  
Как упоминалось в разделе «Типичные ошибки программирования» в гла-  
ве 15, действие таких функций проявляется как побочный эффект, и они обыч-  
но вызываются как инструкции, а не как выражения.  
Переменные цикла в объемлющей области видимости  
Эта ошибка была описана в главе 17, когда мы рассматривали области видимо-  
сти объемлющих функций, однако напомню еще раз: будьте внимательны при   
использовании переменных в области видимости объемлющей функции, кото-  
рые изменяются объемлющим циклом, – все ссылки на эту переменную будут   
запоминать значение, которое будет иметь переменная в последней итерации   
цикла. Чтобы сохранить значения переменной цикла в каждой итерации, ис-  
пользуйте аргументы со значениями по умолчанию (дополнительные сведения   
по этой теме вы найдете в главе 17).  
В заключение  
Эта глава завершает изучение встроенных генераторов и инструментов итера-  
ций. Здесь мы исследовали генераторы списков в контексте инструментов функ-  
ционального программирования и познакомились с функциями-генераторами   
и выражениями-генераторами как дополнительными инструментами под-  
держки протокола итераций. В завершение мы произвели измерения произво-  
дительности различных методов выполнения итераций. Наконец, мы коротко   
рассмотрели типичные ошибки, допускаемые при работе с функциями, чтобы   
помочь вам обойти потенциальные ловушки.  
Этой главой завершается часть книги, посвященная функциям. В следующей   
части мы рассмотрим модули – вершину организационной структуры языка   
Python� структуру, в которой всегда располагаются наши функции. После это-  
го мы займемся исследованием классов – инструментов, которые являются   
пакетами функций со специальным первым аргументом. Как вы увидите, с по-  
мощью пользовательских классов можно реализовать объекты, использующие

Закрепление пройденного   
601  
протокол итераций точно так же, как генераторы и итераторы, с которыми мы   
встретились в этой главе. Все, что мы здесь узнали, пригодится везде, где далее   
в книге будут появляться функции в виде методов классов.  
Но прежде чем двинуться дальше, проверьте, насколько вы овладели основами   
функций, ответив на контрольные вопросы к главе и выполнив упражнения   
для этой части.  
Закрепление пройденного  
Контрольные вопросы  
1. Чем отличаются генераторы списков в квадратных скобках и в круглых   
скобках?  
2. Как связаны между собой генераторы и итераторы?  
3. Как узнать, является ли функция функцией-генератором?  
4. Для чего служит инструкция yield?  
5. Как связаны между собой функция map и генераторы списков? В чем их   
сходства и различия?  
Ответы  
1. Генераторы списков в квадратных скобках воспроизводят сразу весь список   
целиком. Когда генераторы списков заключаются в круглые скобки, они   
фактически превращаются в выражения-генераторы, которые имеют похо-  
жее назначение, но не воспроизводят список результатов целиком. Вместо   
этого выражения-генераторы возвращают объект-генератор, который по-  
ставляет по одному значению при использовании в итерационном контексте.  
2. Генераторы – это объекты, поддерживающие итерационный протокол, –   
они обладают методом \_\_next\_\_, который выполняет переход к следующе-  
му элементу в последовательности результатов и возбуждает исключение   
по достижении конца последовательности. В языке Python существует   
возможность создавать функции-генераторы с помощью инструкции def,   
выражения-генераторы в виде генераторов списков, заключенных в кру-  
глые скобки, и объекты-генераторы с помощью классов, которые определя-  
ют специальный метод \_\_iter\_\_ (обсуждается далее в этой книге).  
3. Функция-генератор имеет в своем теле инструкцию yield. Во всем осталь-  
ном функции-генераторы ничем не отличаются от обычных функций, но   
интерпретатор компилирует их иначе, чтобы эти функции возвращали   
объект-итератор.  
4. При наличии этой инструкции интерпретатор Python компилирует функ-  
цию как генератор – при вызове она возвращает объект-генератор, который   
поддерживает итерационный протокол. Когда запускается инструкция   
yield, она возвращает результат вызывающей программе и приостанавли-  
вает работу функции, после этого, в ответ на вызов встроенной функции   
next или метода \_\_next\_\_ со стороны вызывающей программы, функция воз-  
обновляет свою работу с позиции после последней выполненной инструк-  
ции yield. Функции-генераторы также могут содержать инструкцию return,   
которая завершает работу генератора.

602   
Глава 20. Итераторы и генераторы   
5. Вызов функции map напоминает генератор списков тем, что обе конструк-  
ции создают новый список с результатами, применяя операцию к каждому   
элементу последовательности или другого итерируемого объекта, по одно-  
му за раз. Главное различие состоит в том, что map применяет к каждому   
элементу функцию, а генератор списков – произвольное выражение. Вслед-  
ствие этого генераторы списков обладают большей гибкостью – они могут   
применять функцию, как и map, а функция map требует, чтобы применяемое   
выражение было оформлено в виде функции. Кроме того, генераторы спи-  
сков поддерживают расширенный синтаксис. Например, вложенные ци-  
клы for и условные операторы if, что делает их похожими на встроенную   
функцию filter.  
Упражнения к четвертой части  
В этих упражнениях вам будет предложено написать более сложные програм-  
мы. Обязательно проверьте решения в разделе «Часть IV, Функции» в прило-  
жении B� оформляйте свои решения в виде файлов модулей. Если будет допу-  
щена ошибка, будет очень сложно повторно ввести эти упражнения с клавиа-  
туры в интерактивной оболочке.  
1. Основы. В интерактивной оболочке интерпретатора Python напишите   
функцию, которая выводит на экран единственный аргумент, и попробуй-  
те вызвать ее несколько раз, передавая объекты различных типов: строки,   
целые числа, списки, словари. Затем попробуйте вызвать ее без аргументов.   
Что произошло? Что произойдет, если передать функции два аргумента?  
2. Аргументы. Напишите функцию с именем adder в файле модуля. Функция   
должна принимать два аргумента и возвращать их сумму (или конкатена-  
цию). Затем добавьте в конец файла модуля вызовы функции adder с объ-  
ектами различных типов (две строки, два списка, два вещественных числа)   
и запустите этот файл как сценарий из командной строки операционной   
системы. Должны ли вы явно производить вывод результатов, чтобы они   
появились на экране?  
3. Переменное число аргументов. Обобщите функцию adder из предыдущего   
упражнения, чтобы она вычисляла сумму произвольного числа аргумен-  
тов, и измените вызовы функции так, чтобы ей передавалось больше или   
меньше двух аргументов. Какой тип имеет возвращаемое значение суммы?   
(Подсказка: срез, такой как S[:0], возвращает пустую последовательность   
того же типа, что и S, а с помощью встроенной функции type можно узнать   
тип объекта – смотрите примеры с функцией min в главе 18, где использу-  
ется подобный прием.) Что произойдет, если функции передать аргументы   
разных типов? Что произойдет, если ей передать словари?  
4. Именованные аргументы. Измените функцию adder из упражнения 2 так,   
чтобы она принимала и вычисляла сумму/конкатенацию трех аргументов:   
def adder(good, bad,ugly). После этого определите значения по умолчанию   
для каждого из аргументов и поэкспериментируйте с функцией в интерак-  
тивной оболочке. Попробуйте передавать ей один, два, три и четыре аргу-  
мента. Попробуйте передавать аргументы по именам. Будет ли работать та-  
кой вызов: adder(ugly=1, good=2)? Почему? Наконец, обобщите новую версию   
функции adder так, чтобы принимала и вычисляла сумму/конкатенацию   
произвольного числа именованных аргументов. Решение будет напоминать   
то, что было получено в упражнении 3, с той лишь разницей, что вам при-

Закрепление пройденного   
603  
дется выполнить обход словаря, а не кортежа. (Подсказка: метод dict.keys()   
возвращает список, который можно обойти с помощью цикла for или while,   
но не забудьте обернуть его вызовом функции list в Python 3.0, чтобы обе-  
спечить возможность обращения к элементам по индексам!)  
5. Напишите функцию с именем copyDict(dict), которая копирует словарь, по-  
лучаемый в виде аргумента. Она должна возвращать новый словарь, содер-  
жащий все элементы аргумента. Используйте метод keys для выполнения   
итераций (или, в Python 2.2, выполните обход ключей словаря без вызова   
метода keys). Копирование последовательностей выполняется достаточно   
просто (выражение X[:] выполняет поверхностное копирование)� будет ли   
этот метод работать со словарями?  
6. Напишите функцию с именем addDict(dict1, dict2), которая вычисляет объ-  
единение двух словарей. Она должна возвращать новый словарь, содержа-  
щий все элементы обоих аргументов (которые, как предполагается, являют-  
ся словарями). Если один и тот же ключ присутствует в обоих аргументах,   
вы можете выбрать значение из любого словаря. Проверьте свою функцию,   
добавив программный код проверки в файл и запустив его как сценарий.   
Что произойдет, если вместо словарей передать списки? Как можно было   
бы обобщить функцию, чтобы она обрабатывала и этот случай? (Подсказка:   
смотрите встроенную функцию type, использовавшуюся ранее.) Имеет ли   
значение порядок следования аргументов?  
7. Дополнительные примеры на сопоставление аргументов. Сначала опреде-  
лите следующие шесть функций (в интерактивной оболочке или в файле   
модуля, который затем можно будет импортировать):  
def f1(a, b): print(a, b) # Обычные аргументы  
   
def f2(a, \*b): print(a, b) # Переменное число позиционных аргументов  
   
def f3(a, \*\*b): print(a, b) # Переменное число именованных аргументов  
   
def f4(a, \*b, \*\*c): print(a, b, c) # Смешанный режим  
   
def f5(a, b=2, c=3): print(a, b, c)# Аргументы со значениями по умолчанию  
   
def f6(a, b=2, \*c): print(a, b, c) # Переменное число позиционных аргументов   
 # и аргументов со значениями по умолчанию  
Теперь протестируйте следующие вызовы в интерактивной оболочке и по-  
пробуйте объяснить полученные результаты – в некоторых случаях вам,   
возможно, придется вернуться к обсуждению алгоритмов сопоставления   
в главе 18. Как вы думаете, смешивание режимов сопоставления вообще   
можно считать удачным выбором? Можете ли вы придумать ситуации, ког-  
да это могло бы оказаться полезным?  
>>> f1(1, 2)  
>>> f1(b=2, a=1)  
   
>>> f2(1, 2, 3)  
>>> f3(1, x=2, y=3)  
>>> f4(1, 2, 3, x=2, y=3)  
   
>>> f5(1)  
>>> f5(1, 4)

604   
Глава 20. Итераторы и генераторы   
>>> f6(1)  
>>> f6(1, 3, 4)  
8. Снова простые числа. Вспомните следующий фрагмент из главы 13, кото-  
рый определяет – является ли целое положительное число простым:  
x = y // 2 # Для значений y > 1  
while x > 1:  
 if y % x == 0: # Остаток  
 print(y, ‘has factor’, x)  
 break # Обойти блок else  
 x = x-1  
else: # Обычный выход  
 print(y, ‘is prime’)  
Оформите этот фрагмент в виде функции в файле модуля (значение y долж-  
но передаваться как аргумент) и добавьте несколько вызовов функции в ко-  
нец этого файла. При этом замените оператор // на / в первой строке, чтобы   
увидеть, как изменилось действие оператора деления в Python 3.0 и как это   
изменение отрицательно влияет на работоспособность данного фрагмента   
(вернитесь к главе 5, если вам необходимо освежить свои знания). Что вы   
можете сказать об отрицательных значениях? Сумеете ли вы повысить ско-  
рость работы этой функции? Вывод из вашего модуля должен выглядеть   
примерно так, как показано ниже:  
13 is prime  
13.0 is prime  
15 has factor 5  
15.0 has factor 5.0  
9. Генераторы  списков. Напишите программный код, который будет созда-  
вать новый список, содержащий квадратные корни всех чисел из следую-  
щего списка: [2, 4, 9, 16, 25]. Начните с реализации на основе цикла for,   
затем на основе функции map и, наконец, в виде генератора списков. Для вы-  
числений используйте функцию sqrt из модуля math (то есть выполните im-  
port math и вызывайте функцию, как math.sqrt(x)). Какой из трех вариантов,   
на ваш взгляд, является лучшим?  
10. Хронометраж. В главе 5 мы видели три способа вычисления квадратных   
корней: math.sqrt(X), X \*\* .5 и pow(X, .5). Если в вашей программе часто воз-  
никает необходимость вычислять квадратные корни, сопоставление произ-  
водительности этих трех методов может оказаться значимым. Чтобы уви-  
деть, какой из способов является наиболее быстрым, перепишите сценарий   
timerseqs.py, который был создан нами в этой главе, так, чтобы он измерял   
производительность каждого из трех способов. Используйте функцию best   
из модуля mytimer.py (вы можете использовать версию для Python 3.0 с ар-  
гументами, которые могут передаваться только по именам, или версию, со-  
вместимую с 2.6/3.0). Для большего удобства вы можете также переписать   
реализацию испытаний в этом сценарии так, чтобы имелась возможность   
передавать испытуемые функции, например, в виде кортежа (для этого   
упражнения вполне подойдет прием копирования и вставки фрагментов   
кода). Какой из трех способов вычисления квадратных корней оказался са-  
мым быстрым на вашем компьютере и в вашей версии Python? Наконец,   
как можно было бы выполнить хронометраж производительности генерато-  
ров словарей и эквивалентных им циклов for в интерактивном сеансе?

Часть V.  
Модули

Глава 21.  
   
Модули: общая картина  
Начиная с этой главы, мы приступаем к детальному изучению модулей в язы-  
ке Python – самой крупной организационной программной единицы, кото-  
Python – самой крупной организационной программной единицы, кото-  
 – самой крупной организационной программной единицы, кото-  
рая вмещает в себя программный код и данные, готовые для многократного   
использования. Если говорить более точно, модули в языке Python обычно   
соответствуют файлам программ (или расширениям, написанным на других   
языках программирования, таких как C, Java или C#). Каждый файл – это   
отдельный модуль, и модули могут импортировать другие модули для доступа   
к именам, которые в них определены. Обработка модулей выполняется двумя   
инструкциями и одной важной функцией:   
import  
Позволяет клиентам (импортерам) получать модуль целиком.  
from  
Позволяет клиентам получать определенные имена из модуля.  
imp.reload  
Обеспечивает возможность повторной загрузки модуля без остановки ин-  
терпретатора Python.  
В главе 3 были представлены основные принципы, касающиеся модулей, и мы   
пользовались ими до сих пор. Эту часть книги мы начнем с подробного опи-  
сания базовых концепций, а затем перейдем к исследованию расширенных   
возможностей использования модулей. В этой первой главе вашему внима-  
нию предлагается общий взгляд на роль модулей в структуре всей программы.   
В последующих главах мы начнем рассматривать программный код, который   
основан на этой теории.  
Попутно мы подробно рассмотрим сведения о модулях, которые до сих пор   
были опущены: вы узнаете об операции перезагрузки модулей, об атрибутах   
\_\_name\_\_ и \_\_all\_\_, об импорте пакетов и так далее. Поскольку модули и клас-  
сы – это всего лишь пространства имен, здесь мы также формализуем понятие   
пространства имен.

608   
Глава 21. Модули: общая картина   
Зачем нужны модули?  
В двух словах, модули обеспечивают простой способ организации компонен-  
тов в систему автономных пакетов переменных, известных как пространства   
имен. Все имена, определяемые на верхнем уровне модуля, становятся атри-  
бутами объекта импортируемого модуля. Как мы видели в предыдущей части,   
операция импорта предоставляет доступ к именам в глобальной области види-  
мости модуля. Таким образом, в процессе импортирования глобальная область   
видимости модуля образует пространство имен атрибутов объекта модуля.   
В конечном счете модули позволяют связывать отдельные файлы в крупные   
программные системы.  
Если говорить более определенно, с точки зрения теории модули играют как   
минимум три роли:  
Повторное использование программного кода  
Как говорилось в главе 3, модули позволяют сохранять программный код   
в виде файлов. В отличие от программного кода, который вводится в ин-  
терактивной оболочке интерпретатора Python и исчезает безвозвратно по-  
Python и исчезает безвозвратно по-  
 и исчезает безвозвратно по-  
сле выхода из оболочки, программный код в файлах модулей хранится по-  
стоянно – его можно повторно загружать и запускать столько раз, сколько   
потребуется. Можно добавить, что модули – это место, где определяются   
имена, известные как атрибуты, на которые могут ссылаться множество   
внешних клиентов.  
Разделение системы пространств имен  
Модули в языке Python также являются самой высокоуровневой единицей   
организации программ. По существу, они – всего лишь пакеты имен. Моду-  
ли позволяют изолировать имена в замкнутые пакеты, которые позволяют   
избежать конфликтов имен – вы никогда не увидите имя в другом файле,   
если не импортируете его. Фактически все, что находится в модуле, – вы-  
полняемый программный код и создаваемые объекты – всегда неявно   
включается в модули. Вследствие этого модули являются естественными   
инструментами группировки компонентов системы.  
Реализация служб или данных для совместного пользования  
С функциональной точки зрения модули могут также использоваться для   
реализации компонентов, используемых системой, вследствие чего тре-  
буется только одна копия такого компонента. Например, если необходим   
глобальный объект, который используется более чем одной функцией или   
модулем, можно написать его в виде модуля, который затем может импор-  
тироваться множеством клиентов.  
Однако, чтобы понять действительную роль модулей в системе Python, нам не-  
Python, нам не-  
, нам не-  
обходимо отступить на шаг назад и исследовать общую структуру программы   
на языке Python.  
Архитектура программы на языке Python  
До сих пор в этой книге я избегал сложностей в описаниях программ на язы-  
ке Python. Обычно программы состоят более, чем из одного файла, – любые   
программы, за исключением самых простых сценариев, состоят из нескольких   
файлов. Даже если вам удастся поместить всю логику в один файл, вы почти

Архитектура программы на языке Python   
609  
наверняка будете использовать сторонние модули, которые уже кем-то были   
написаны.  
В этом разделе дается введение в общую архитектуру программ на языке   
Python – способ, которым программа делится на коллекцию файлов с исходны-  
ми текстами (то есть модулей) и увязывается в единое целое. Кроме того, мы   
попутно рассмотрим основные концепции модулей в языке Python, процедуру   
импортирования и атрибуты объектов.  
Как организована программа  
Как правило, программа на языке Python состоит из множества текстовых фай-  
лов, содержащих инструкции. Программа организована как один главный файл,   
к которому могут подключаться дополнительные файлы, известные как модули.   
Главный файл (или сценарий) определяет, как будет двигаться основной поток   
выполнения программы, – это тот файл, который необходимо запустить, чтобы   
начать работу приложения. Файлы модулей – это библиотеки инструменталь-  
ных средств, где содержатся компоненты, используемые главным файлом (и,   
возможно, где-то еще). Главный файл использует инструменты, определенные   
в файлах модулей, а модули используют инструменты, определенные в других   
модулях.  
Обычно файлы модулей ничего не делают, если попытаться запустить их от-  
дельно – в них определяются инструментальные средства, используемые в дру-  
гих файлах. Чтобы получить доступ к определенным в модуле инструментам,   
именующимся атрибутами модуля (имена переменных, связанные с такими   
объектами, как функции), в языке Python необходимо импортировать этот   
модуль. То есть мы импортируем модули и получаем доступ к их атрибутам,   
что дает нам возможность использовать их функциональные возможности.  
Импортирование и атрибуты  
Давайте сделаем наше обсуждение более конкретным. На рис. 21.1 схемати-  
чески изображена структура программы на языке Python, состоящей из трех   
файлов: a.py, b.py и c.py. Файл a.py является главным файлом программы – это   
простой текстовый файл, состоящий из инструкций, который при запуске вы-  
полняется от начала и до конца. Файлы b.py и c.py – это модули, они также яв-  
ляются простыми текстовыми файлами, содержащими инструкции, но обыч-  
но они не запускаются как самостоятельные программы. Вместо этого они, как   
уже говорилось выше, обычно импортируются другими файлами, использую-  
щими инструментальные средства, определяемые в этих файлах.   
Например, предположим, что файл b.py на рис. 21.1 определяет функцию с име-  
нем spam. Как мы уже знаем из четвертой части книги, чтобы определить функ-  
цию, которая затем сможет быть запущена за счет передачи ей нуля или более   
аргументов в круглых скобках, файл b.py должен содержать инструкцию def:  
def spam(text):  
 print(text, ‘spam’)  
Теперь предположим, что модуль a.py использует функцию spam. Для этого он   
мог бы содержать следующие инструкции:  
import b  
b.spam(‘gumby’)

610   
Глава 21. Модули: общая картина   
Верхний уровень:  
главный файл программы  
a.py  
b.py  
c.py  
Модули  
стандартной  
библиотеки  
Модули  
Рис. 21.1. Архитектура программы на языке Python. Программа – это сис-  
тема модулей. Она состоит из одного главного файла сценария (который   
требуется запустить, чтобы начать работу программы) и нескольких моду-  
лей (импортируемых библиотек инструментальных средств). И сценарии,   
и модули – это текстовые файлы, содержащие инструкции на языке Python,   
хотя инструкции в модулях обычно только создают объекты для последую-  
щего использования. Стандартная библиотека языка Python представляет   
собой коллекцию модулей, готовых к использованию  
В первой строке здесь располагается инструкция import, дающая файлу a.py   
доступ ко всему, что определено на верхнем уровне в файле b.py. В общих чер-  
тах это означает следующее: «загрузить файл b.py (если он еще не загружен)   
и предоставить доступ ко всем его атрибутам через имя модуля b». Инструкции   
import (и, как вы узнаете далее, from) загружают и запускают другие файлы на   
этапе времени выполнения.  
В языке Python невозможно обращаться к именам в других модулях, пока та-  
кие инструкции импорта не будут выполнены на этапе времени выполнения.   
Основная задача этих инструкций состоит в том, чтобы связать имена в моду-  
ле – простые переменные – с объектами загруженных модулей. Фактически   
имя модуля, используемое в инструкции import, во-первых, идентифицирует   
внешний файл и, во-вторых, становится именем переменной, которая будет   
представлять загруженный модуль. Объекты, определяемые модулем, также   
создаются во время выполнения, когда производится импорт модуля: инструк-  
ция import, в действительности, последовательно выполняет инструкции в ука-  
занном файле, чтобы воссоздать его содержимое.  
Вторая инструкция в файле a.py вызывает функцию spam, определенную в мо-  
дуле b, используя форму записи атрибутов объекта. Запись b.spam означает сле-  
дующее: «извлечь значение имени spam, расположенного в объекте b». В нашем   
примере – это вызываемая функция, поэтому далее ей передается строка в кру-  
глых скобках (‘gumby’). Если вы создадите эти файлы, сохраните их и запусти-  
те файл a.py, то будут выведены слова «gumby spam».  
Вы увидите, что повсюду в сценариях на языке Python используется нотация   
object.attribute – большинство объектов обладают атрибутами, доступ к кото-  
рым можно получить с помощью оператора «.». Некоторые атрибуты – имена   
вызываемых функций, а другие – простые значения, которые представляют   
свойства объекта (например, имя персоны).

Архитектура программы на языке Python   
611  
Импорт – широко используемое понятие в языке Python. Любой файл может   
импортировать функциональные возможности из любого другого файла. На-  
пример, файл a.py может импортировать файл b.py, чтобы иметь возможность   
вызывать его функцию, при этом файл b.py может в свою очередь импортиро-  
вать файл c.py, чтобы получить доступ к другим функциональным возмож-  
ностям, определенным в нем. Цепочка импортирования может уходить так   
глубоко, как это потребуется: в этом примере модуль a может импортировать   
модуль b, который импортирует модуль c, который в свою очередь может еще   
раз импортировать модуль b, и так далее.  
Помимо самой крупной единицы в организационной структуре программы   
модули (и пакеты модулей, которые описываются в главе 23) также играют   
роль самой крупной единицы программного кода, доступного для повторно-  
го использования. Оформив программные компоненты в виде файлов модулей,   
вы сможете использовать их не только в оригинальной программе, но и в лю-  
бых других программах, которые вам придется писать. Например, если после   
написания программы, структура которой изображена на рис. 21.1, вдруг об-  
наружится, что функция b.spam является универсальным инструментом, мы   
сможем использовать в других программах. Все, что нам потребуется, – это   
импортировать файл b.py из файлов другой программы.  
Модули стандартной библиотеки  
Обратите внимание на правую часть рис. 21.1. Некоторые из модулей, которые   
будут импортироваться вашими программами, входят непосредственно в со-  
став языка Python, и вам не придется писать их.  
Интерпретатор Python поставляется с обширной коллекцией дополнительных   
модулей, которая известна как стандартная библиотека. Эта коллекция на-  
считывает порядка 200 крупных модулей и содержит платформонезависимую   
поддержку распространенных задач программирования: интерфейсы опера-  
ционных систем, организацию хранилищ объектов, поиск по шаблону, сетевые   
взаимодействия, создание графического интерфейса и многих других. Ни один   
из этих инструментов не является непосредственной частью языка Python, но   
вы можете использовать их, импортируя соответствующие модули. Так как   
это модули стандартной библиотеки, можно быть уверенным, что они будут до-  
ступны и будут работать переносимым образом на большинстве платформ, на   
которых работает интерпретатор Python.  
В примерах этой книги вы увидите несколько модулей стандартной библиоте-  
ки в действии, но за полной информацией вам следует обратиться к справоч-  
ному руководству по стандартной библиотеке языка Python, которое можно   
найти в инсталляции Python (в IDLE или в меню кнопки Пуск (Start) в опера-  
ционной системе �indows) или в Интернете, по адресу: http://www.python.org.  
При таком большом количестве модулей это действительно единственный спо-  
соб получить представление о том, какие инструментальные средства имеют-  
ся в наличии. Кроме этого, описание библиотеки инструментов Python можно   
найти в некоторых книгах, посвященных прикладному программированию,   
таких как «Программирование на Python»1, но, в отличие от книг, руководства   
распространяются бесплатно, их можно просматривать в любом веб-броузере   
1   
Лутц М. «Программирование на Python», 2-е изд. – Пер. с англ. – СПб.: Символ-  
Плюс, 2002. Четвертое издание этой книги выйдет в 2011 году.

612   
Глава 21. Модули: общая картина   
(поскольку они распространяются в формате HTML) и к тому же они обновля-  
ются с выходом каждой новой версии Python.  
Как работает импорт  
В предыдущем разделе говорилось об импортировании модулей, но никак не   
объяснялось, что происходит во время импорта. Так как в языке Python ин-  
струкции импортирования составляют основу структуры программы, в этом   
разделе более подробно будет рассмотрена операция импорта, чтобы сделать   
представление об этом процессе менее абстрактным.  
Некоторые программисты на языке C любят сравнивать инструкцию import   
в языке Python с инструкцией #include, но они в корне неправы – импортиро-  
вание в языке Python – это не просто включение текста одного файла в другой.   
Это самые настоящие операции времени выполнения, которые выполняют сле-  
дующие действия, когда программа впервые импортирует заданный файл:  
1. Отыскивают файл модуля.  
2. Компилируют в байт-код (если это необходимо).  
3. Запускают программный код модуля, чтобы создать объекты, которые он   
определяет.  
Чтобы лучше понять, как протекает импорт модулей, мы исследуем все эти   
действия по порядку. Примите во внимание, что все три действия выполня-  
ются, только когда модуль впервые импортируется во время выполнения про-  
граммы, – все последующие операции импорта того же модуля пропускают эти   
действия и просто выбирают уже находящийся в памяти объект модуля. Тех-  
нически это обеспечивается за счет того, что интерпретатор сохраняет инфор-  
мацию о загруженных модулях в словаре с именем sys.modules и проверяет его   
при выполнении каждой операции импортирования. Если модуль отсутствует   
в словаре, выполняется трехэтапный процесс, описанный выше.  
1. Поиск  
Прежде всего, интерпретатор должен определить местонахождение файла мо-  
дуля, указанного в инструкции import. Обратите внимание, что имена файлов   
в инструкции import в примерах из предыдущих разделов указаны без рас-  
ширения .py и без пути к каталогу: вместо записи в виде, например, import c:\  
dir1\b.py, инструкция записывается просто – import b. Фактически допуска-  
ется указывать лишь простые имена – путь к каталогу и расширение файла   
должны быть опущены, потому что для поиска файла, соответствующего име-  
ни, указанному в инструкции import, интерпретатор использует стандартный   
путь поиска модулей.1 Поскольку это основная часть операции импорта, кото-  
рую необходимо знать программистам, мы вернемся к ней чуть ниже.  
1   
В действительности синтаксис стандартной инструкции import не позволяет вклю-  
чать путь к файлу и его расширение. Для операции импортирования пакетов, кото-  
рая будет рассматриваться в главе 23, инструкция import допускает указывать путь   
к файлу в виде последовательности имен, разделенных точкой� но при этом операция   
импортирования пакетов по-прежнему использует обычный путь поиска модулей,   
чтобы отыскать самый первый каталог в указанном пути к пакету (то есть пути к па-  
кетам указываются относительно одного из каталогов, находящегося в пути поиска).

Как работает импорт   
613  
2. Компиляция (если необходимо)   
После того как в пути поиска модулей будет найден файл, соответствующий   
имени в инструкции import, интерпретатор компилирует его в байт-код, если   
это необходимо. (Мы рассматривали байт-код в главе 2.)  
Интерпретатор проверяет время создания файла и пропускает этап компиля-  
ции исходного программного кода, если файл с байт-кодом .pyc не старше, чем   
соответствующий ему файл .py с исходным текстом. Кроме того, если Python   
обнаружит в пути поиска только файл с байт-кодом и не найдет файл с исход-  
ным текстом, он просто загрузит байт-код (это означает, что вы можете рас-  
пространять свою программу исключительно в виде файлов с байт-кодом и не   
передавать файлы с исходными текстами). Другими словами, этап компиля-  
ции пропускается, если можно ускорить запуск программы. Если вы измените   
исходный программный код, Python автоматически скомпилирует байт-код   
при следующем запуске программы.   
Обратите внимание, что компиляция выполняется в момент импортирования   
файла. По этой причине файл .pyc с байт-кодом для главного файла программы   
обычно не создается, если только он не был импортирован еще куда-нибудь –   
файлы .pyc создаются только при импортировании файлов. Байт-код главного   
файла программы создается в памяти компьютера, а байт-код импортирован-  
ных файлов сохраняется в файлах для ускорения будущих операций импорта.  
Главные файлы программ часто планируется исполнять непосредственно и ни-  
куда их не импортировать. Позднее мы увидим, что существует возможность   
создать файл, который будет играть роль как главного файла программы, так   
и модуля, доступного для импорта. Такие файлы могут и исполняться, и им-  
портироваться, поэтому для них создаются соответствующие файлы .pyc. Что-  
бы разобраться с тем, как это получается, читайте обсуждение специальных   
атрибутов \_\_name\_\_ и \_\_main\_\_ в главе 24.  
3. Запуск  
На последнем шаге операции импортирования производится запуск байт-кода   
модуля. Все инструкции в файле модуля выполняются по порядку, сверху   
вниз, и любые операции присваивания, которые встретятся на этом шаге, бу-  
дут создавать атрибуты конечного объекта модуля. Таким образом, этот этап   
выполнения создает все инструменты, которые определяются модулем. Напри-  
мер, во время импортирования выполняются инструкции def в файле, которые   
создают функции и присваивают их атрибутам модуля. После этого функции   
могут вызываться из программы, выполнившей импорт.  
На этом последнем шаге операции импортирования фактически запускается   
программный код модуля, поэтому если программный код верхнего уровня   
в файле модуля выполняет какие-нибудь действия, результаты этих действий   
можно будет наблюдать во время импорта. Например, при импорте файла мож-  
но будет наблюдать результат работы инструкций print на верхнем уровне мо-  
Кроме того, эта операция не позволяет использовать в инструкции import синтаксис   
путей к каталогам, характерный для определенной платформы – этот синтаксис   
может использоваться только при определении самого пути поиска. Следует также   
заметить, что проблемы поиска пути к файлам модулей отсутствуют при запуске   
фиксированных двоичных файлов (рассматривались в главе 2)� они обычно содержат   
в двоичном образе весь необходимый байт-код.

614   
Глава 21. Модули: общая картина   
дуля. Инструкции def просто определяют объекты для последующего исполь-  
зования.  
Как видите, во время импорта выполняется достаточно много работы – про-  
изводится поиск файла, в случае необходимости запускается компилятор   
и производится запуск программного кода. Вследствие этого любой заданный   
модуль по умолчанию импортируется только один раз за все время работы про-  
граммы. При повторных попытках импортировать модуль все три шага просто   
пропускаются, и повторно используется модуль, уже загруженный в память.   
Если вам потребуется еще раз импортировать файл, который уже был загру-  
жен (например, чтобы обеспечить поддержку настроек, выполняемых пользо-  
вателем), воспользуйтесь функцией imp.reload, с которой мы встретимся в сле-  
дующей главе.1  
Путь поиска модулей  
Как уже отмечалось выше, для большинства программистов наиболее важным   
в операции импортирования является первый этап – поиск файла импорти-  
руемого модуля (раздел «1. Поиск»). Вам может потребоваться сообщить ин-  
терпретатору, где следует искать импортируемые файлы, поэтому вы должны   
знать, как получить доступ к пути поиска, чтобы дополнить его.  
В большинстве случаев можно положиться на автоматически организованные   
пути поиска модулей и вообще не настраивать этот путь. Однако, если вам по-  
требуется импортировать модули из пользовательских каталогов, вам необхо-  
димо будет знать, как работает путь поиска файлов, чтобы настроить его соот-  
ветствующим образом. В общих чертах пути поиска модулей в языке Python   
выбираются из объединенных данных следующих основных источников. Не-  
которые из них предопределены, а некоторые можно настроить и тем самым   
сообщить интерпретатору, где выполнять поиск:  
1. Домашний каталог программы.  
2. Содержимое переменной окружения PYTHONPATH (если таковая определена).   
3. Каталоги стандартной библиотеки.   
4. Содержимое любых файлов с расширением .pht (если таковые имеются).   
В конечном итоге объединение этих четырех компонентов составляет sys.path –   
список строк с именами каталогов, о котором я расскажу подробнее в следую-  
щем разделе. Первый и третий компоненты пути поиска определяются авто-  
матически, но так как интерпретатор при поиске использует данные всех этих   
компонентов, от первого до последнего, то второй и четвертый компоненты   
можно использовать для расширения пути поиска, включая в него свои соб-  
ственные каталоги с исходными текстами. Далее описывается, как интерпре-  
татор Python использует эти компоненты пути:  
1   
Как уже говорилось выше, интерпретатор сохраняет импортированные модули во   
встроенном словаре sys.modules, благодаря чему он в состоянии определить, какие   
модули уже были загружены. В действительности, если вам потребуется получить   
перечень загруженных модулей, вы можете импортировать модуль sys и вывести ре-  
зультат инструкции list(sys.modules.keys()). Подробнее об использовании этой вну-  
тренней таблицы рассказывается в главе 24.

Путь поиска модулей   
615  
Домашний каталог  
В первую очередь интерпретатор ищет импортируемые файлы в домашнем   
каталоге. В зависимости от того, как была запущена программа, это может   
быть каталог, где находится главный файл программы. При работе в инте-  
рактивном сеансе этот элемент содержит путь к каталогу, откуда была за-  
пущена интерактивная оболочка (то есть текущий рабочий каталог).  
Поскольку поиск в первую очередь всегда производится в этом каталоге,   
если программа целиком располагается в одном каталоге, все операции   
импорта будут выполнены автоматически, без необходимости настраивать   
путь поиска. С другой стороны, из-за того, что поиск в первую очередь про-  
изводится в этом каталоге, файлы, находящиеся в нем, могут сделать недо-  
ступными модули с теми же именами, находящиеся в других каталогах, –   
будьте внимательны, чтобы случайно не сделать недоступными библиотеч-  
ные модули, которые могут потребоваться вашей программе.  
Каталоги в PYTHONPATH  
После этого поиск производится во всех каталогах, перечисленных в пере-  
менной окружения PYTHONPATH, слева направо (если эта переменная вообще   
установлена). В двух словах, переменная окружения PYTHONPATH – это про-  
сто список имен каталогов, определяемых пользователем и системой, в ко-  
торых располагаются файлы с программным кодом на языке Python. Вы   
можете добавить в эту переменную все каталоги, откуда предполагается   
импортировать модули, и интерпретатор будет использовать ваши настрой-  
ки при создании пути поиска модулей.  
Поскольку интерпретатор пытается отыскать файлы сначала в домашнем   
каталоге, настройка этой переменной приобретает большое значение, толь-  
ко когда необходимо импортировать модули, размещающиеся в разных ка-  
талогах, – то есть, когда импортируемый файл хранится в каталоге, отлич-  
ном от каталога, где располагается импортирующий файл. Вам наверняка   
потребуется настраивать переменную окружения PYTHONPATH, как только вы   
начнете писать большие программы, но на начальном этапе освоения языка   
храните файлы всех своих модулей в каталоге, в котором вы работаете (то   
есть в домашнем каталоге), и тогда операции импорта будут работать без не-  
обходимости выполнять какие-либо настройки.  
Каталоги стандартной библиотеки  
Далее интерпретатор автоматически выполняет поиск в каталогах, куда   
были установлены модули стандартной библиотеки. Так как эти каталоги   
всегда участвуют в поиске, их можно не добавлять в переменную окруже-  
ния PYTHONPATH, или включать в файлы с расширением .pth, о которых рас-  
сказывается ниже.  
Каталоги в файле .pth  
Наконец, относительно новая особенность языка Python дает пользова-  
телям возможность добавлять нужные каталоги в путь поиска модулей,   
просто перечисляя их по одному в строке в текстовом файле, имя которого   
оканчивается расширением .pth (от слова «path» – «путь»). Эти файлы пред-  
ставляют собой расширенную возможность, имеющую отношение к про-  
блеме установки, и мы не будем здесь подробно их обсуждать. Впрочем, они   
могут служить альтернативой настройки переменной PYTHONPATH.

616   
Глава 21. Модули: общая картина   
Текстовый файл со списком каталогов помещается в соответствующий ка-  
талог и может играть примерно ту же роль, что и переменная окружения   
PYTHONPATH. Например, если вы работаете в Python 3.0 под �indows, файл   
с именем myconfig.pth можно поместить в главный каталог, куда был уста-  
новлен Python 3.0 (например, C:\Python30), или в подкаталог site-packages   
C:\Python30\Lib\site-packages стандартной библиотеки, что позволит рас-  
ширить путь поиска модулей. В UNIX-подобных системах этот файл можно   
поместить в каталог /usr/local/lib/python3.0/site-packages или /usr/local/lib/  
site-python.  
Обнаружив этот файл, интерпретатор добавит в конец пути поиска модулей   
каталоги, перечисленные во всех строках файла, от первой до последней.   
Интерпретатор выберет все имена каталогов во всех файлах .pth, которые   
обнаружит, и отфильтрует повторяющиеся имена и имена несуществую-  
щих каталогов. Поскольку это файлы, а не параметры настройки команд-  
ной оболочки, они могут применяться ко всем пользователям системы, а не   
только к одному пользователю или одной командной оболочке. Кроме того,   
для некоторых пользователей процедура создания текстовых файлов вы-  
глядит проще, чем настройка переменных окружения.  
Эта особенность на практике более сложная, чем я описал. За дополни-  
тельной информацией обращайтесь к руководству по библиотеке языка   
Python, в частности к описанию модуля site, входящего в стандартную   
биб лиотеку, – этот модуль позволяет создавать файлы .pth и определять ме-  
стоположение библиотек языка Python, а в документации к нему описыва-  
ются каталоги, где вообще могут располагаться файлы .pth. Начинающим   
я рекомендую использовать переменную окружения PYTHONPATH или един-  
ственный файл .pth и только в том случае, если возникает необходимость   
импортировать файлы из других каталогов. Наиболее часто файлы .pth   
используются в сторонних библиотеках, которые обычно устанавливают   
файлы .pth в каталог site-packages, чтобы исключить необходимость допол-  
нительных настроек (система установки пакетов distutils, описываемая во   
врезке ниже, позволяет автоматизировать многие операции, выполняемые   
при установке).   
Настройка пути поиска  
Из всего вышесказанного следует, что переменная окружения PYTHONPATH   
и файлы .pth позволяют вам определять каталоги, где интерпретатор будет ис-  
кать файлы при выполнении операции импортирования. Способ настройки   
переменных окружения и имена каталогов, где могут храниться файлы .pth,   
зависит от типа платформы. Например, в �indows можно воспользоваться яр-  
�indows можно воспользоваться яр-  
 можно воспользоваться яр-  
лыком Система (System) в панели управления, чтобы записать в переменную PY-  
THONPATH список каталогов, разделенных точкой с запятой, как показано ниже:  
c:\pycode�tilities;d:\pycode\package1  
Или создать текстовый файл с именем C:\Python30\pydirs.pth, который выгля-  
дит примерно так:  
c:\pycode�tilities  
d:\pycode\package1

Путь поиска модулей   
617  
Аналогичным образом выполняются настройки и на других платформах, од-  
нако детали настроек могут изменяться в слишком широком диапазоне, чтобы   
осветить их все в этой главе. В приложении A вы найдете указания по расши-  
рению пути поиска файлов с помощью переменной окружения PYTHONPATH или   
файлов .pth на различных платформах.  
Автоматическое изменение пути поиска  
Это описание пути поиска модулей является верным, но достаточно общим –   
точная конфигурация пути поиска зависит от типа платформы и версии   
Python. В зависимости от используемой платформы в путь поиска модулей ав-  
томатически могут добавляться дополнительные каталоги.  
Например, в путь поиска вслед за каталогами из переменной окружения PY-  
THONPATH и перед каталогами стандартной библиотеки интерпретатор может   
добавлять текущий рабочий каталог – каталог, откуда была запущена про-  
грамма. Когда программа запускается из командной строки, текущий рабочий   
каталог может не совпадать с домашним каталогом, где находится главный   
файл программы (то есть с каталогом, где находится программа). Так как от за-  
пуска к запуску программы текущий рабочий каталог может изменяться, при   
обычных условиях рабочий каталог не должен иметь значения для операций   
импорта. Подробнее о запуске программ из командной строки рассказывается   
в главе 3.1   
Чтобы увидеть, как интерпретатор настраивает путь поиска модулей на вашей   
платформе, вы можете проверить содержимое переменной sys.path, обсужде-  
ние которой является темой следующего раздела.  
Список sys.path  
Если вам потребуется узнать, как выглядит путь поиска на вашей машине, вы   
всегда сможете сделать это, просмотрев содержимое встроенного списка sys.  
path (то есть содержимое атрибута path модуля sys, входящего в состав стан-  
дартной библиотеки). Этот список строк с именами каталогов представляет со-  
бой фактический путь поиска, используемый интерпретатором, – при выпол-  
нении операций импорта Python просматривает каждый каталог из списка,   
слева направо.  
Действительно, sys.path – это путь поиска модулей. Интерпретатор создает его   
во время запуска программы, автоматически объединяя в список домашний   
каталог (или пустую строку, что соответствует текущему рабочему каталогу)   
все каталоги, перечисленные в переменной окружения PYTHONPATH и в файлах   
.pth, и каталоги стандартной библиотеки. В результате получается список   
строк с именами каталогов, которые просматриваются интерпретатором при   
импортировании новых файлов.  
1   
В главе 23 дополнительно обсуждается новый  синтаксис  инструкции  импортиро-  
вания по относительному пути, добавленный в версии Python 3.0, – благодаря ему   
можно изменять путь поиска в инструкции from с помощью символов «.» (например,   
from . import string). По умолчанию при выполнении в Python 3.0 операции импортиро-  
вания интерпретатор не производит поиск в собственном каталоге пакета, если в фай-  
лах пакета не используется синтаксис импортирования по относительному пути.

618   
Глава 21. Модули: общая картина   
Представление языком Python этого списка имеет два основных полезных ре-  
зультата. Во-первых, он обеспечивает возможность проверить настройки пути   
поиска, которые вы выполнили, – если вы не видите свои настройки в этом   
списке каталогов, вам следует проверить, насколько правильно вы все проде-  
лали. Например, ниже показано, как выглядит путь поиска модулей у меня,   
в операционной системе �indows, в Python 3.0, с моими настройками пере-  
�indows, в Python 3.0, с моими настройками пере-  
, в Python 3.0, с моими настройками пере-  
Python 3.0, с моими настройками пере-  
 3.0, с моими настройками пере-  
менной окружения PYTHONPATH, куда записан каталог C:�sers, и с моим файлом   
C:\Python30\mypath.pth, содержащим путь к каталогу C:�sers\mark. Пустая   
строка в начале списка соответствует текущему рабочему каталогу, а мои на-  
стройки объединены с системными (остальные пути в списке – это каталоги   
стандартной библиотеки):  
>>> import sys  
>>> sys.path  
[‘’, ‘C:\\users’, ‘C:\\Windows\\system32\\python30.zip’, ‘c:\\Python30\\DLLs’,  
‘c:\\Python30\\lib’, ‘c:\\Python30\\lib\\plat-win’, ‘c:\\Python30’,  
‘C:\\Users\\Mark’, ‘c:\\Python30\\lib\\site-packages’]  
Во-вторых, если вы понимаете, как формируется список, вы можете обеспе-  
чить сценариям возможность самостоятельно задавать свои пути поиска. Как   
будет показано далее в этой части книги, изменяя список sys.path, вы можете   
изменить путь поиска для всех последующих операций импорта. Однако эти   
изменения продолжают действовать, только пока выполняется сценарий� пере-  
менная окружения PYTHONPATH и файлы .pth обеспечивают возможность более   
долговременного хранения измененного пути.1  
Выбор файла модуля  
Имейте в виду, что расширения имен файлов (например, .py) преднамеренно   
опущены в инструкции import. Интерпретатор выбирает первый найденный   
в пути поиска файл, который соответствует указанному имени. Например, ин-  
струкция import b могла бы загрузить:  
 •  
Файл с исходным текстом, имеющий имя b.py.  
 •  
Файл с байт-кодом, имеющий имя b.pyc.  
 •  
Содержимое каталога b при импортировании пакета (описывается в гла-  
ве 23).  
 •  
Скомпилированный модуль расширения, написанный, как правило, на   
языке C или C++ и скомпонованный в виде динамической библиотеки (на-  
пример, b.so в Linux и b.dll или b.pyd в Cygwin и в �indows).  
 •  
Скомпилированный встроенный модуль, написанный на языке C и стати-  
чески скомпонованный с интерпретатором Python.  
 •  
Файл ZIP-архива с компонентом, который автоматически извлекается при   
импорте.  
1   
Некоторым программам действительно требуется изменять sys.path. Сценарии, ко-  
торые выполняются на веб-сервере, например, обычно выполняются с привилегиями   
пользователя «nobody» с целью ограничить доступ к системе. Поскольку такие сце-  
нарии обычно не должны зависеть от значения переменной окружения PYTHONPATH для   
пользователя «nobody», они часто изменяют список sys.path вручную, чтобы вклю-  
чить в него необходимые каталоги до того, как будет выполнена какая-либо инструк-  
ция import. Обычно для этого бывает достаточно вызова sys.path.append(dirname).

Путь поиска модулей   
619  
 •  
Образ памяти для фиксированных двоичных исполняемых файлов.  
 •  
Класс Java в версии Jython.  
 •  
Компонент .NET в версии IronPython.  
Импортирование расширений, написанных на языке C, операция импорта   
в Jython и импортирование пакетов – это расширенные возможности импорти-  
рования компонентов, не являющихся простыми файлами модулей. Впрочем,   
для импортера различия в типах загружаемых файлов совершенно незаметны   
как при импорте, так и при обращении к атрибутам модуля. Инструкция im-  
port b загружает некоторый модуль b в соответствии с настройками пути поис-  
ка модулей, а инструкция b.attr извлекает элемент модуля, будь то перемен-  
ная или функция, написанная на языке C. Некоторые стандартные модули,   
которые мы будем использовать в этой книге, в действительности написаны на   
языке C, но благодаря прозрачности импортирования, это не имеет никакого   
значения для клиентов.  
Если у вас в различных каталогах имеются файлы b.py и b.so, интерпретатор   
всегда будет загружать тот, что будет найден в каталоге, который располага-  
ется раньше (левее) в пути поиска модулей, так как поиск в списке sys.path вы-  
полняется слева направо. Но что произойдет, если оба файла, b.py и b.so, на-  
ходятся в одном и том же каталоге? В этом случае интерпретатор будет следо-  
вать стандартному порядку выбора файлов, впрочем, нет никаких гарантий,   
что такой порядок будет оставаться неизменным с течением времени. Вообще   
вы должны избегать зависимости от порядка выбора файлов интерпретатором   
Python в одном и том же каталоге – давайте своим модулям различные имена   
или настраивайте путь поиска модулей, чтобы обеспечить более очевидный по-  
рядок выбора файлов.  
Дополнительные возможности выбора модуля   
Обычно операция импорта работает именно так, как описывается в данном раз-  
деле, – она отыскивает и загружает файлы, находящиеся на вашей машине.   
Однако вполне возможно переопределить большую часть того, что делает опе-  
рация импорта, используя то, что называется программными ловушками им-  
порта. Эти ловушки могут использоваться, чтобы придать операции импорта   
дополнительные полезные возможности, такие как загрузка файлов из архи-  
вов, расшифровывание и так далее.   
Фактически сам интерпретатор Python использует эти ловушки, чтобы обеспе-  
чить возможность извлечения импортируемых компонентов из ZIP-архивов, –   
заархивированные файлы автоматически извлекаются во время импорта,   
когда в пути поиска выбирается файл с расширением .zip. Например, один из   
каталогов стандартной библиотеки в списке sys.path, представленном выше, на   
сегодняшний день является файлом .zip. За дополнительной информацией об-  
ращайтесь к описанию встроенной функции \_\_import\_\_ в руководстве по стан-  
дартной библиотеке Python – настраиваемому инструменту, которым в дей-  
ствительности пользуется инструкция import.  
Кроме того, Python поддерживает понятие файлов с оптимизированным байт-  
кодом (.pyo), которые создаются и запускаются интерпретатором из командной   
строки с флагом –O, – однако они выполняются лишь немногим быстрее, чем   
обычные файлы .pyc (обычно на 5 процентов быстрее), поэтому они использу-

620   
Глава 21. Модули: общая картина   
ются достаточно редко. Система Psyco (глава 2) обеспечивает куда более суще-  
ственный прирост в скорости выполнения.  
Стороннее программное обеспечение: distutils  
Настройка пути поиска модулей, описание которой приводится в этой   
главе, в первую очередь касается программного кода, который вы пи-  
шете самостоятельно. Сторонние расширения для Python обычно ис-  
пользуют для автоматической установки самих себя такой инструмент,   
как distutils, входящий в состав стандартной библиотеки, поэтому для   
использования такого программного кода не требуется выполнять на-  
стройку пути поиска модулей.  
Системы, использующие distutils, обычно поставляются со сценарием   
setup.py, который запускается для установки таких систем, – этот сце-  
нарий импортирует и использует модуль distutils, чтобы поместить   
систему в каталог, который уже является частью пути поиска модулей   
(обычно в подкаталог Lib\sitepackages в каталоге, куда был установлен   
Python).  
За дополнительной информацией о распространении и установке про-  
грамм с помощью distutils обращайтесь к стандартному набору руко-  
водств по языку Python, потому что эта тема далеко выходит за рамки   
данной книги (например, этот инструмент дополнительно обеспечивает   
возможность компиляции расширений на языке C на машине, где про-  
изводится установка). Кроме того, обратите внимание на развивающую-  
ся систему eggs, распространяемую с открытыми исходными текстами,   
которая добавляет возможность проверки зависимостей для установ-  
ленного программного кода на языке Python.  
В заключение  
В этой главе были даны основные понятия, имеющие отношение к модулям,   
атрибутам и импорту, а также был исследован принцип действия инструкции   
import. Мы узнали, что во время операции импортирования производится по-  
иск файла модуля в пути поиска модулей, компиляция в байт-код и выпол-  
нение всех инструкций, создающих его содержимое. Мы также узнали, как   
настроить путь поиска, в первую очередь с помощью переменной окружения   
PYTHONPATH, чтобы иметь возможность импортировать модули из других катало-  
гов, отличных от домашнего каталога и от каталогов стандартной библиотеки.   
Как показала эта глава, операция импортирования и модули являются осно-  
вой архитектуры программ на языке Python. Крупные программы делятся на   
множество файлов, которые связываются между собой во время выполнения   
посредством импортирования. Местонахождение файлов модулей во время им-  
портирования определяется с помощью пути поиска модулей, а модули опреде-  
ляют атрибуты, использующиеся за пределами этих модулей.  
Конечно, основное назначение операции импорта и модулей состоит в том, что-  
бы образовать структуру программы, логика которой подразделяется на са-  
мостоятельные программные компоненты. Программный код в одном модуле

Закрепление пройденного   
621  
изолирован от программного кода в другом – фактически ни один модуль не   
может получить доступ к именам, определенным в другом модуле, если явно   
не выполнит инструкцию import. Благодаря этому модули дают возможность   
минимизировать конфликты имен между различными частями программы.  
В следующей главе вы увидите, что все это означает в терминах фактического   
программного кода. Но прежде чем двинуться дальше, постарайтесь ответить   
на контрольные вопросы к главе.  
Закрепление пройденного  
Контрольные вопросы  
1. Каким образом файл с исходным программным кодом модуля превращает-  
ся в объект модуля?  
2. Зачем может потребоваться настраивать значение переменной окружения   
PYTHONPATH?  
3. Назовите четыре основных компонента, составляющих путь поиска моду-  
лей.  
4. Назовите четыре типа файлов, которые могут загружаться операцией им-  
портирования.  
5. Что такое пространство имен, и что содержит пространство имен модуля?  
Ответы  
1. Файл с исходными текстами модуля автоматически превращается в объ-  
ект модуля в результате выполнения операции импортирования. С техни-  
ческой точки зрения исходный программный код модуля выполняется во   
время импортирования, инструкция за инструкцией, и все имена, которым   
по мере выполнения операций будут присвоены значения, превращаются   
в атрибуты объекта модуля.  
2. Настройка переменной PYTHONPATH может потребоваться только в случае не-  
обходимости импортировать модули, размещенные в каталогах, отличных   
от того, в котором вы работаете (то есть отличных от текущего каталога при   
работе в интерактивной оболочке или от каталога, где находится главный   
файл программы).  
3. Четырьмя основными компонентами, составляющими путь поиска моду-  
лей, являются: домашний каталог главного файла программы (каталог,   
в котором он находится), все каталоги, перечисленные в переменной окру-  
жения PYTHONPATH, каталоги стандартной библиотеки и все каталоги в фай-  
лах с расширением .pth, размещенных в стандартных местах. Из них до-  
ступны для настройки переменная окружения PYTHONPATH и файлы с расши-  
рением .pth.  
4. Интерпретатор Python может загружать файлы с исходными текстами   
(.py), файлы с байт-кодом (.pyc), файлы расширений, написанных на язы-  
ке C (например, файлы с расширением .so в Linux или с расширением .dll   
в �indows), или каталог с указанным именем, в случае импортирования   
пакета. Операция импортирования может также загружать менее обыч-  
ные файлы, такие как компоненты из архивов в формате ZIP, классы Java

622   
Глава 21. Модули: общая картина   
в Jython – версии Python, компоненты .NET в IronPython и статически   
скомпонованные расширения, написанные на языке C, которые вообще не   
представлены в виде файлов. С помощью программных ловушек, которые   
имеет реализация операции импорта, можно загрузить все, что угодно.  
5. Пространство имен – это независимый пакет переменных, известных как   
атрибуты пространства имен объекта. Пространство имен модуля содер-  
жит все имена, присваивание значений которым производится программ-  
ным кодом на верхнем уровне модуля (то есть не вложенным в инструкции   
def или class). С технической точки зрения глобальная область видимости   
трансформируется в пространство имен атрибутов объекта модуля. Про-  
странство имен модуля может изменяться с помощью операций присваива-  
ния из других файлов, которые импортируют данный модуль, хотя это и не   
приветствуется (подробнее об этом рассказывается в главе 17).

Глава 22.  
   
Основы программирования модулей  
Теперь, когда мы рассмотрели общие идеи, лежащие в основе модулей, обра-  
тимся к простому примеру модулей в действии. Модули в языке Python соз-  
даются очень просто – это всего лишь файлы с программным кодом на языке   
Python, которые создаются с помощью текстового редактора. Вам не требу-  
ется употреблять специальные инструкции, чтобы сообщить интерпретатору   
Python, что вы создаете модуль, – практически любой текстовый файл может   
играть эту роль. Интерпретатор сам заботится о поиске и загрузке модулей, по-  
этому их очень просто использовать – клиент просто импортирует модуль или   
определенные имена из модуля и использует объекты, на которые эти имена   
ссылаются.   
Создание модуля  
Чтобы определить модуль, достаточно воспользоваться текстовым редактором,   
с его помощью ввести некоторый программный код на языке Python в текстовый   
файл и сохранить его с расширением «.py» – любой такой файл автоматически   
будет считаться модулем Python. Все имена, которым будет выполнено присваи-  
вание на верхнем уровне модуля, станут его атрибутами (именами, ассоцииро-  
ванными с объектом модуля) и будут доступны для использования клиентами.  
Например, если ввести следующую инструкцию def в файл с именем module1.  
py и импортировать его, тем самым будет создан объект модуля с единствен-  
ным атрибутом – именем printer, которое ссылается на объект функции:  
def printer(x): # Атрибут модуля  
 print(x)  
Прежде чем мы двинемся дальше, следует сказать несколько слов об именах   
файлов модулей. Вы можете называть ваши модули, как вам будет угодно, при   
условии, что эти имена будут оканчиваться расширением .py, если вы собирае-  
тесь импортировать их. Для главных файлов программ, которые будут запу-  
скаться, но не будут импортироваться, имена не обязательно должны иметь рас-  
ширение .py, однако было бы желательно использовать это расширение в любом   
случае, потому что оно делает назначение файлов более очевидным и позволит   
в будущем импортировать любой из ваших файлов.

624   
Глава 22. Основы программирования модулей   
Поскольку имена модулей внутри программы превращаются в имена перемен-  
ных (без расширения .py), они также должны следовать правилам именования   
обычных переменных, которые приводились в главе 11. Например, можно   
создать файл модуля с именем if.py, но его невозможно будет импортировать,   
потому что if – это зарезервированное слово, и когда вы попытаетесь выпол-  
нить инструкцию import if, интерпретатор выдаст сообщение о синтаксической   
ошибке. Фактически и имена модулей, и имена каталогов, используемых при   
импортировании пакетов (рассматривается в следующей главе), должны соот-  
ветствовать требованиям, предъявляемым к именам переменных и представ-  
ленным в главе 11 – они могут, например, содержать только алфавитные сим-  
волы, цифры и символы подчеркивания. Имена каталогов с пакетами также   
не могут содержать посторонних символов, таких как пробелы, даже если они   
являются допустимыми для используемой платформы.  
Когда производится импорт модуля, интерпретатор Python преобразует имя   
модуля в имя внешнего файла, добавляя в начало путь к каталогу из пути по-  
иска модулей и добавляя .py или другое расширение в конец. Например, в ко-  
нечном итоге имя модуля M преобразуется в имя некоторого внешнего файла   
<каталог>\M.<расширение>, который содержит программный код модуля.  
Как упоминалось в предыдущей главе, существует возможность создать модуль   
для Python на другом языке программирования, таком как C или C++ (или Java   
в реализации Jython). Такие модули называются модулями расширений и обыч-  
но используются для создания библиотек, используемых сценариями на языке   
Python. Когда модули расширений импортируются программным кодом на язы-  
ке Python, они выглядят и ведут себя точно так же, как обычные модули, напи-  
санные на языке Python, – они импортируются инструкцией import и предостав-  
ляют функции и объекты в виде атрибутов объекта модуля. Обсуждение модулей   
расширений выходит далеко за рамки этой книги, поэтому за дополнительной   
информацией обращайтесь к стандартным руководствам по языку Python или   
к специализированным книгам, таким как «Программирование на Python».  
Использование модулей  
Клиенты могут использовать простой файл модуля, только что написанный   
нами, выполнив инструкцию import или from. Обе инструкции отыскивают,   
компилируют и запускают программный код модуля, если он еще не был за-  
гружен. Главное различие этих инструкций заключается в том, что инструк-  
ция import загружает модуль целиком, поэтому при обращении к именам в мо-  
дуле их необходимо дополнять именем модуля. Инструкция from, напротив,   
загружает (или копирует) из модуля отдельные имена.  
Все следующие примеры вызывают функцию printer, определенную во внеш-  
нем модуле module1.py, но делают это различными способами.  
Инструкция import  
В первом примере имя module1 служит двум различным целям – оно идентифи-  
цирует внешний файл, который должен быть загружен, и превращается в имя   
переменной, которая ссылается на объект модуля после загрузки файла:  
>>> import module1 # Загрузить модуль целиком  
>>> module1.printer(‘Hello world!’) # Имя дополняется именем модуля   
Hello world!

Использование модулей   
625  
Так как в результате выполнения инструкции import в сценарии появляется   
имя, ссылающееся на полный объект модуля, нам необходимо использовать   
имя модуля при обращении к его атрибутам (например, module1.printer).  
Инструкция from  
Инструкция from, напротив, копирует имена из области видимости одного фай-  
ла в область видимости другого, что позволяет непосредственно использовать   
скопированные имена, не предваряя их именем модуля (например, printer):  
>>> from module1 import printer # Копировать одну переменную  
>>> printer(‘Hello world!’) # Имя не требует дополнения  
Hello world!  
Этот пример дает тот же результат, что и предыдущий, но так как импортируе-  
мое имя копируется в область видимости, в которой находится сама инструк-  
ция from, можно напрямую обращаться к переменной в сценарии, не предваряя   
его именем вмещающего модуля.  
Как будет показано далее, инструкция from является всего лишь небольшим   
расширением инструкции import – она импортирует файл модуля как обычно,   
но выполняет дополнительный шаг, на котором копирует одно или более имен   
из импортируемого файла.  
Инструкция from \*  
Наконец, в следующем примере используется специальная форма инструкции   
from: когда используется символ \*, копируются все имена, которым присваива-  
ются значения на верхнем уровне указанного модуля. В этом случае точно так   
же можно использовать скопированное имя printer, не предваряя его именем   
модуля:  
>>> from module1 import \* # Скопировать все переменные  
>>> printer(‘Hello world!’)  
Hello world!  
С технической точки зрения обе инструкции, from и import, вызывают одну и ту   
же операцию импорта, просто форма from \* дополнительно выполняет копиро-  
вание всех имен в импортируемом модуле в область видимости, откуда произ-  
водится импорт. По сути происходит совмещение пространств имен модулей,   
что позволяет нам меньше вводить с клавиатуры.  
Как видите, модули действительно легко использовать. Чтобы еще лучше по-  
нять, что происходит в действительности, когда вы определяете и используете   
модули, рассмотрим некоторые их свойства более подробно.  
В Python 3.0 форма инструкции from ...\*, описанная здесь, мо-  
жет использоваться только на верхнем уровне модуля – она не   
может вызываться внутри функций. В Python 2.6 эту инструк-  
цию можно использовать внутри функций, но она вызывает по-  
явление предупреждения. На практике эта инструкция чрезвы-  
чайно редко встречается в функциях, так как ее присутствие   
лишает интерпретатор возможности определять переменные   
статически, до вызова функции.

626   
Глава 22. Основы программирования модулей   
Импорт выполняется только один раз  
Один из самых типичных вопросов, которые задают начинающие программи-  
сты, начав использовать модули: «Почему операция импорта перестает рабо-  
тать?» Они часто сообщают, что при первой попытке импортировать модуль все   
работает, но последующие попытки импорта в интерактивной оболочке (или   
во время работы программы) не дают должного эффекта. В действительности   
такой эффект и не предполагается, и вот почему.  
Модули загружаются и запускаются первой, и только первой инструкцией im-  
port или from. Реализовано такое поведение преднамеренно, потому что импор-  
тирование – это дорогостоящая операция и интерпретатор выполняет ее всего   
один раз за все время работы. Последующие операции импорта просто получа-  
ют объект уже загруженного модуля.  
Из этого следует: так как программный код на верхнем уровне модуля выпол-  
няется всего один раз, это обстоятельство можно использовать для инициали-  
зации переменных. Рассмотрим пример модуля simple.py:  
Print(‘hello’)  
spam = 1 # Инициализировать переменную  
В этом примере инструкции print и = выполняются, когда модуль импортиру-  
ется впервые, и переменная spam инициализируется во время импортирования:  
% python  
>>> import simple # Первая инструкция import: загружает и запускает код модуля  
hello  
>>> simple.spam # Операция присваивания создает атрибут  
1  
Вторая и все последующие операции импортирования не приводят к перезапу-  
ску программного кода модуля – они просто получают объект модуля из вну-  
тренней таблицы модулей интерпретатора. В результате повторная инициали-  
зация переменной spam не происходит:  
>>> simple.spam = 2 # Изменить атрибут модуля  
>>> import simple # Просто получает уже загруженный модуль  
>>> simple.spam # Код не перезапускается: атрибут не изменился  
2  
Конечно, иногда действительно бывает необходимо перезапустить программ-  
ный код модуля при повторных операциях импортирования. Позднее в этой   
главе мы увидим, как это можно сделать с помощью функции reload.  
Инструкции import и from – операции присваивания  
Так же, как и инструкция def, инструкции import и from являются выполняе-  
мыми инструкциями, а не объявлениями времени компиляции. Они могут   
вкладываться в условные инструкции if, присутствовать в объявлениях функ-  
ций def и так далее, и они не имеют никакого эффекта, пока интерпретатор не   
достигнет их в ходе выполнения программы. Другими словами, импортируе-  
мые модули и имена в них не будут доступны, пока не будут выполнены соот-  
ветствующие инструкции import или from. Кроме того, подобно инструкции def,   
import и from – это явные операции присваивания:

Использование модулей   
627  
 •  
Инструкция import присваивает объект модуля единственному имени.  
 •  
Инструкция from присваивает одно или более имен объектам с теми же име-  
нами в другом модуле.  
Все, что уже обсуждалось ранее, в равной степени применимо и к модулям. На-  
пример, имена, копируемые инструкцией from, становятся ссылками на разде-  
ляемые объекты – как и в случае с аргументами функций, повторное присваи-  
вание полученному имени не оказывает воздействия на модуль, откуда это имя   
было скопировано, но модификация изменяемого  объекта может оказывать   
воздействие на объект в модуле, откуда он был импортирован. Для иллюстра-  
ции рассмотрим следующий файл small.py:  
x = 1  
y = [1, 2]  
   
% python  
>>> from small import x, y # Скопировать два имени  
>>> x = 42 # Изменяется только локальная переменная x   
>>> y[0] = 42 # Изменяется непосредственно изменяемый объект  
Здесь x не является разделяемым изменяемым объектом, а вот y – является.   
Имена y в импортирующем и импортируемом модулях ссылаются на один и тот   
же объект списка, поэтому изменения, произведенные в одном модуле, будут   
видны в другом модуле:  
>>> import small # Получить имя модуля (инструкция from его не дает)  
>>> small.x # x в модуле small – это не моя переменная x  
1  
>>> small.y # Но изменяемый объект используется совместно  
[42, 2]  
Чтобы увидеть графическое изображение того, что делает со ссылками ин-  
струкция from, вернитесь к рис. 18.1 (передача аргументов функциям) и мыс-  
ленно замените слова «вызывающая программа» и «функция» на «импорти-  
руемый модуль» и «импортирующий модуль». Эффект тот же самый, за исклю-  
чением того, что здесь мы имеем дело с именами в модулях, а не с функциями.   
Операция присваивания везде в языке Python работает одинаково.  
Изменение значений имен в других файлах  
Вспомним, что в предыдущем примере присваивание переменной x в интерак-  
тивной оболочке изменяло ее значение только в этой области видимости и не   
оказывало влияния на переменную x в файле – между именем, скопированным   
инструкцией from, и именем в файле, откуда это имя было скопировано, нет ни-  
какой связи. Чтобы действительно изменить глобальное имя в другом файле,   
необходимо использовать инструкцию import:  
% python  
>>> from small import x, y # Скопировать два имени  
>>> x = 42 # Изменить только локальное имя x  
   
>>> import small # Получить имя модуля  
>>> small.x = 42 # Изменить x в другом модуле  
Это явление было описано в главе 17. Поскольку изменение переменных в дру-  
гих модулях, как в данном случае, часто является источником проблем (и след-

628   
Глава 22. Основы программирования модулей   
ствием неудачного проектирования), мы еще вернемся к этому приему позднее   
в этой части книги. Обратите внимание, что изменение элемента y[0] в преды-  
дущем примере – это нечто иное� изменяется объект, а не имя.  
Эквивалентность инструкций import и from  
Обратите внимание: в предыдущем примере после инструкции from нам по-  
требовалось выполнить инструкцию import, чтобы получить доступ к имени   
модуля small, – инструкция from копирует только имена из одного модуля   
в другой и ничего не присваивает самому имени модуля. Инструкция from,   
приведенная ниже:  
from module import name1, name2 # Копировать только эти два имени  
эквивалентна следующей последовательности, по крайней мере, концептуально:  
import module # Получить объект модуля  
name1 = module.name1 # Скопировать имена с помощью присваивания  
name2 = module.name2  
del module # Удалить имя модуля  
Как и все операции присваивания, инструкция from создает новые переменные   
в импортирующем модуле, которые ссылаются на объекты с теми же именами   
в импортируемом файле. При этом копируются только имена, а не сам модуль.   
При использовании формы from \* этой инструкции (from module import \*) эквива-  
лентная последовательность действий та же самая, только при этом копируют-  
ся все имена, определенные на верхнем уровне импортируемого модуля.  
Обратите внимание, что на первом шаге инструкция from выполняет обычную   
операцию import. Вследствие этого инструкция from всегда импортирует весь   
модуль целиком, если он еще не был импортирован, независимо от того, сколь-  
ко имен копируется из файла. Нет никакой возможности загрузить только   
часть модуля (например, только одну функцию), но так как модули – это байт-  
код, а не машинный код, влияние на производительность оказывается незна-  
чительным.  
Потенциальные проблемы инструкции from  
Инструкция from делает местоположение переменных менее явным и очевид-  
ным (имя name несет меньше информации, чем module.name), поэтому некоторые   
пользователи Python рекомендуют использовать инструкцию import вместо   
from. Однако я не уверен, что это такой уж однозначный совет: инструкция   
from используется достаточно часто и без каких-либо страшных последствий.   
На практике часто бывает удобно избавиться от необходимости набирать имя   
модуля всякий раз, когда требуется использовать один из его инструментов.   
Это особенно справедливо для крупных модулей, которые предоставляют   
большое число атрибутов, таких как модуль tkinter из стандартной библиоте-  
ки, например.  
Суть проблемы состоит в том, что инструкция from способна повреждать про-  
странства имен, по крайней мере, в принципе – если использовать ее для им-  
портирования переменных, когда существуют одноименные переменные в име-  
ющейся области видимости, то эти переменные просто будут перезаписаны.   
Эта проблема отсутствует при использовании инструкции import, потому что   
доступ к содержимому импортируемого модуля возможен только через его имя

Использование модулей   
629  
(имя module.attr не конфликтует с именем attr в текущей области видимости).   
Пока вы понимаете и контролируете все, что может происходить при исполь-  
зовании инструкции from, во всем этом нет большой проблемы, особенно если   
импортируемые имена указываются явно (например, from module import x, y, z).  
С другой стороны, инструкция from скрывает в себе более серьезные пробле-  
мы, когда используется в комбинации с функцией reload, так как импортиро-  
ванные имена могут ссылаться на предыдущие версии объектов. Кроме того,   
инструкция в форме from module import \* действительно может повреждать про-  
странства имен и затрудняет понимание имен, особенно когда она применяется   
более чем к одному файлу, – в этом случае нет никакого способа определить,   
какому модулю принадлежит то или иное имя, разве только выполнить поиск   
по файлам с исходными текстами. В действительности форма from \* вставляет   
одно пространство имен в другое, что сводит на нет преимущества, которые не-  
сет возможность разделения пространств имен. Мы будем рассматривать эти   
проблемы более подробно в разделе «Типичные ошибки при работе с модуля-  
ми» в конце этой части книги (глава 24).  
Пожалуй, лучший совет, который можно дать, – отдавать предпочтение ин-  
струкции import для простых модулей, явно перечислять необходимые пере-  
менные в инструкциях from и не использовать форму from \* для импорта более   
чем одного файла в модуле. При таком подходе можно предполагать, что все   
неопределенные имена располагаются в модуле, к которому обращались через   
инструкцию from \*. При работе с инструкцией from, конечно, следует проявлять   
осторожность, но, вооруженные знаниями, большинство программистов нахо-  
дят ее удобной для организации доступа к модулям.  
Когда необходимо использовать инструкцию import  
Единственное, когда необходимо вместо инструкции from использовать ин-  
струкцию import, – когда требуется использовать одно и то же имя, присутству-  
ющее в двух разных модулях. Например, когда два файла по-разному опреде-  
ляют одно и то же имя:  
# M.py  
   
def func():  
 ...выполнить что-то одно...  
   
# N.py  
   
def func():  
 ...выполнить что-то другое...  
и необходимо использовать обе версии имени в программе. В этом случае ин-  
струкцию from использовать нельзя, потому что в результате вы получите един-  
ственное имя в вашей области видимости:  
# O.py  
   
from M import func  
from N import func # Перезапишет имя, импортированное из модуля M  
func() # Будет вызвана N.func  
Зато можно использовать инструкцию import, потому что включение имени   
вмещающего модуля сделает имена уникальными:

630   
Глава 22. Основы программирования модулей   
# O.py  
   
import M, N # Получить модуль целиком, а не отдельные имена  
M.func() # Теперь можно вызывать обе функции  
N.func() # Наличие имени модуля делает их уникальными  
Этот случай достаточно необычен, поэтому вы вряд ли часто будете сталкивать-  
ся с ним на практике. Но если такая ситуация все-таки возникнет, инструкция   
import позволит вам избежать конфликта имен.  
Пространства имен модулей  
Модули будут, вероятно, более понятны, если представлять их, как простые   
пакеты имен, – то есть место, где определяются переменные, которые долж-  
ны быть доступны остальной системе. С технической точки зрения каждому   
модулю соответствует отдельный файл, и интерпретатор создает объект моду-  
ля, содержащий все имена, которым присвоены какие-либо значения в файле   
модуля. Проще говоря, модули – это всего лишь пространства имен (места, где   
создаются имена), и имена, находящиеся в модуле, называются его атрибута-  
ми. В этом разделе мы исследуем, как работает этот механизм.  
Файлы создают пространства имен  
Итак, как же файлы трансформируются в пространства имен? Суть в том, что   
каждое имя, которому присваивается некоторое значение на верхнем уровне   
файла модуля (то есть не вложенное в функции или в классы), превращается   
в атрибут этого модуля.  
Например, операция присваивания, такая как X = 1, на верхнем уровне модуля   
M.py превращает имя X в атрибут модуля M, обратиться к которому из-за преде-  
лов модуля можно как M.X. Кроме того, имя X становится глобальной перемен-  
ной для программного кода внутри M.py, но нам необходимо более формально   
объяснить понятия загрузки модуля и областей видимости, чтобы понять, по-  
чему:  
 •  
Инструкции модуля выполняются во время первой попытки импорта.   
Когда модуль импортируется в первый раз, интерпретатор Python создает   
пустой объект модуля и выполняет инструкции в модуле одну за другой, от   
начала файла до конца.  
 •  
Операции присваивания, выполняемые на верхнем уровне, создают атри-  
буты модуля. Во время импортирования инструкции присваивания, вы-  
полняемые на верхнем уровне файла и не вложенные в инструкции def или   
class (например, =, def), создают атрибуты объекта модуля – при присваива-  
нии имена сохраняются в пространстве имен модуля.  
 •  
Доступ к пространствам имен модулей можно получить через атрибут   
\_\_dict\_\_ или dir(M). Пространства имен модулей, создаваемые операцией   
импортирования, представляют собой словари – доступ к ним можно по-  
лучить через встроенный атрибут \_\_dict\_\_, ассоциированный с модулем, и с   
помощью функции dir. Функция dir – это примерный эквивалент отсорти-  
рованного списка ключей атрибута \_\_dict\_\_, но она включает унаследован-  
ные имена классов, может возвращать не полный список и часто изменяет-  
ся от версии к версии.

Пространства имен модулей   
631  
 •  
Модуль – это единая область видимости (локальная является глобаль-  
ной). Как мы видели в главе 17, имена на верхнем уровне модуля подчиня-  
ются тем же правилам обращения/присваивания, что и имена в функциях,   
только в этом случае локальная область видимости совпадает с глобальной   
(точнее, они следуют тому же правилу LEGB поиска в областях видимости,   
с которым мы познакомились в главе 17, только без уровней поиска L и E).   
Но в модулях область видимости модуля после загрузки модуля превраща-  
ется в атрибут-словарь объекта модуля. В отличие от функций (где локаль-  
ное пространство имен существует только во время выполнения функции),   
область видимости файла модуля превращается в область видимости атри-  
бутов объекта модуля и никуда не исчезает после выполнения операции им-  
портирования.  
Ниже эти понятия демонстрируются в программном коде. Предположим, мы   
создаем в текстовом редакторе следующий файл модуля с именем module2.py:  
Print(‘starting to load...’)  
import sys  
name = 42  
   
def func(): pass  
   
class klass: pass  
   
print(‘done loading.’)  
Когда модуль будет импортироваться в первый раз (или будет запущен как   
программа), интерпретатор выполнит инструкции модуля от начала до конца.   
В ходе операции импортирования одни инструкции создают имена в простран-  
стве имен модуля, а другие выполняют определенную работу. Например, две   
инструкции print в этом файле выполняются во время импортирования:  
>>> import module2  
starting to load...  
done loading.  
Но как только модуль будет загружен, его область видимости превратится   
в пространство имен атрибутов объекта модуля, который возвращает инструк-  
ция import. После этого можно обращаться к атрибутам в этом пространстве   
имен, дополняя их именем вмещающего модуля:  
>>> module2.sys  
<module ‘sys’ (built-in)>  
   
>>> module2.name  
42  
   
>>> module2.func  
<function func at 0x026D3BB8>>  
   
>>> module2.klass  
<class module2.klass>  
Здесь именам sys, name, func и klass были присвоены значения во время выпол-  
нения инструкций модуля, поэтому они стали атрибутами после завершения   
операции импортирования. О классах мы будем говорить в шестой части кни-  
ги, но обратите внимание на атрибут sys – инструкции import действительно

632   
Глава 22. Основы программирования модулей   
присваивают объекты модулей именам, а любая операция присваивания на   
верхнем уровне файла создает атрибут модуля.  
Внутри интерпретатора пространства имен хранятся в виде объектов слова-  
рей. Это самые обычные объекты словарей с обычными методами. Обратиться   
к словарю пространства имен модуля можно через атрибут \_\_dict\_\_ модуля (не   
забудьте обернуть вызов этого метода вызовом функции list – в Python 3.0 он   
возвращает объект представления!):  
>>> list(module2.\_\_dict\_\_.keys())  
[‘name’, ‘\_\_builtins\_\_’, ‘\_\_file\_\_’, ‘\_\_package\_\_’, ‘sys’, ‘klass’, ‘func’,  
‘\_\_name\_\_’, ‘\_\_doc\_\_’]  
Имена, которые были определены в файле модуля, становятся ключами вну-  
, которые были определены в файле модуля, становятся ключами вну-  
которые были определены в файле модуля, становятся ключами вну-  
 были определены в файле модуля, становятся ключами вну-  
были определены в файле модуля, становятся ключами вну-  
 определены в файле модуля, становятся ключами вну-  
определены в файле модуля, становятся ключами вну-  
треннего словаря, таким образом, большинство имен здесь отражают операции   
присваивания на верхнем уровне в файле. Однако интерпретатор Python до-  
Python до-  
 до-  
бавляет в пространство имен модуля еще несколько имен, например \_\_file\_\_   
содержит имя файла, из которого был загружен модуль, а \_\_name\_\_ – это имя,   
под которым модуль известен импортерам (без расширения .py и без пути к ка-  
талогу).  
Квалификация имен атрибутов  
После ознакомления с модулями мы должны поближе рассмотреть понятие   
квалификации имен. В языке Python для доступа к атрибутам любого объекта   
используется синтаксис квалификации имени object.attribute.  
Квалификация имени в действительности является выражением, возвращаю-  
щим значение, присвоенное имени атрибута, ассоциированного с объектом.   
Например, выражение module2.sys в предыдущем примере возвращает значе-  
ние атрибута sys в объекте module2. Точно так же, если имеется встроенный объ-  
ект списка L, выражение L.append вернет метод append, ассоциированный с этим   
списком.  
Итак, какую роль играет квалификация имен атрибутов с точки зрения пра-  
вил, рассмотренных нами в главе 17? В действительности – никакую: это совер-  
шенно независимые понятия. Когда вы обращаетесь к именам, квалифицируя   
их, вы явно указываете интерпретатору объект, атрибут которого требуется   
получить. Правило LEGB применяется только к кратким, неполным именам.   
Ниже приводятся принятые правила:  
Простые переменные  
Использование краткой формы имени, например X, означает, что будет про-  
изведен поиск этого имени в текущих областях видимости (следуя правилу   
LEGB).  
Квалифицированные имена  
Имя X.Y означает, что будет произведен поиск имени X в текущих областях   
видимости, а затем будет выполнен поиск атрибута Y в объекте X (не в обла-  
стях видимости).  
Квалифицированные пути  
Имя X.Y.Z означает, что будет произведен поиск имени Y в объекте X, а затем   
поиск имени Z в объекте X.Y.

Пространства имен модулей   
633  
Общий случай  
Квалификация имен применима ко всем объектам, имеющим атрибуты:   
модулям, классам, расширениям типов на языке C и так далее.  
В шестой части книги мы увидим, что квалификация имен для классов имеет   
немного большее значение (здесь также имеет место то, что называется насле-  
дованием), но в общем случае правила, описанные здесь, применяются ко всем   
именам в языке Python.  
Импортирование и области видимости  
Как мы уже знаем, невозможно получить доступ к именам, определенным   
в другом модуле, не импортировав его предварительно. То есть вы никогда ав-  
томатически не получите доступ к именам в другом файле, независимо от вида   
импортируемого модуля и вызовов функций в вашей программе. Смысл пере-  
менной всегда определяется местоположением операции присваивания в про-  
граммном коде, а для обращения к атрибутам всегда необходимо явно указы-  
вать объект.  
Например, рассмотрим два следующих простых модуля. Первый, moda.py,   
определяет переменную X, которая является глобальной только для программ-  
ного кода в этом файле, и функцию, изменяющую глобальную переменную X   
в этом файле:  
X = 88 # Переменная X: глобальная только для этого файла  
def f():  
 global X # Изменяет переменную X в этом файле  
 X = 99 # Имена в других модулях недоступны  
Второй модуль, modb.py, определяет свою собственную глобальную перемен-  
ную X, а также импортирует и вызывает функцию из первого модуля:  
X = 11 # Переменная X: глобальная только для этого файла   
   
import moda # Получает доступ к именам в модуле moda  
moda.f() # Изменяет переменную moda.X, но не X в этом файле  
print X, moda.X  
При запуске этого модуля функция moda.f изменит переменную X в модуле moda,   
а не в modb. Глобальной областью видимости для функции moda.f всегда явля-  
ется файл, вмещающий ее, независимо от того, из какого модуля она была вы-  
звана:  
% python modb.py  
11 99  
Другими словами, операция импортирования никогда не изменяет область ви-  
димости для программного кода в импортируемом файле – из импортируемого   
файла нельзя получить доступ к именам в импортирующем файле. Если быть   
более точным:  
 •  
Функциям никогда не будут доступны имена, определенные в других функ-  
циях, если только они физически не вложены друг в друга.  
 •  
Программному коду модуля никогда не будут доступны имена, определен-  
ные в других модулях, если только они явно не были импортированы.

634   
Глава 22. Основы программирования модулей   
Это поведение является частью понятия лексической  области  видимости –   
в языке Python области видимости, доступные части программного кода, пол-  
Python области видимости, доступные части программного кода, пол-  
 области видимости, доступные части программного кода, пол-  
ностью определяются физическим расположением этого программного кода   
в файле. Области видимости не подвержены влияниям вызовов функций или   
операции импортирования.1  
Вложенные пространства имен  
Операция импорта не дает возможности доступа к внешним областям видимо-  
сти, но она дает возможность обращаться к вложенным областям видимости.   
Используя квалифицированные пути к именам атрибутов, вполне возможно   
погрузиться в сколь угодно глубоко вложенные модули и получить доступ к их   
атрибутам. Например, рассмотрим следующие три файла. Файл mod3.py опре-  
деляет единственное глобальное имя и атрибут операцией присваивания:  
X = 3  
Файл mod2.py определяет свою переменную X, затем импортирует модуль mod3   
и использует квалификацию имени, чтобы получить доступ к атрибуту импор-  
тированного модуля:  
X = 2  
import mod3  
   
print(X, end=’ ‘) # Моя глобальная переменная X  
print mod3.X # Глобальная переменная X из модуля mod3  
Файл mod1.py также определяет свою собственную переменную X, затем импор-  
тирует модуль mod2 и получает значения атрибутов обоих модулей:  
X = 1  
import mod2  
   
print(X, end=’ ‘) # Моя глобальная переменная X  
print(mod2.X, end=’ ‘) # Переменная X из модуля mod2  
print(mod2.mod3.X # Переменная X из модуля mod3  
В действительности, когда mod1 импортирует mod2, он создает двухуровневое   
вложение пространств имен. Используя полный путь к имени mod2.mod3.X, он   
может погрузиться в модуль mod3, который вложен в импортированный модуль   
mod2. Суть в том, что модуль mod1 может обращаться к переменным X во всех   
трех файлах и, следовательно, имеет доступ ко всем трем глобальным областям   
видимости:  
% python mod1.py  
2 3  
1 2 3  
Однако обратное утверждение неверно: модуль mod3 не имеет доступа к именам   
в mod2, а модуль mod2 не имеет доступа к именам в mod1. Возможно, этот пример   
будет проще понять, если отвлечься от пространств имен и областей видимо-  
1   
Некоторые языки программирования подразумевают иной порядок действий и реа-  
лизуют динамические области видимости, когда области видимости в действитель-  
ности могут зависеть от вызовов функций во время выполнения программы. Это   
усложняет программный код, потому что смысл переменной может изменяться с те-  
чением времени.

Повторная загрузка модулей   
635  
сти и сосредоточиться на объектах, задействованных в примере. mod2 внутри   
модуля mod1 – это всего лишь имя, которое ссылается на объект с атрибутами,   
некоторые из которых могут ссылаться на другие объекты с атрибутами (ин-  
струкция import выполняет операцию присваивания). Для таких путей, как   
mod2.mod3.X, интерпретатор Python выполняет вычисления слева направо, из-  
влекая атрибуты из объектов.  
Обратите внимание: в mod1 можно вставить инструкцию import mod2 и затем ис-  
пользовать обращение mod2.mod3.X, но нельзя записать import mod2.mod3 – такой   
синтаксис используется для операции импортирования пакетов (каталогов),   
которая будет описана в следующей главе. При импортировании пакетов также   
создаются вложенные пространства имен, но в этом случае инструкция import   
воспринимает свой аргумент как дерево каталогов, а не как цепочку модулей.  
Повторная загрузка модулей  
Как мы уже видели, программный код модуля по умолчанию запускается все-  
го один раз за все время работы программы. Чтобы принудительно повторно за-  
грузить модуль и запустить программный код в нем, необходимо явно вызвать   
встроенную функцию reload. В этом разделе мы исследуем, как использовать   
возможность повторной загрузки модулей, чтобы сделать систему более дина-  
мичной. В двух словах:  
 •  
При вызове операции импортирования (с помощью инструкций import   
и from) программный код модуля загружается и выполняется, только когда   
модуль импортируется в первый раз за время работы программы.  
 •  
При последующих попытках импортировать модуль будет использоваться   
объект уже загруженного модуля. Повторная загрузка и запуск программ-  
ного кода в этом случае не происходит.  
 •  
Функция reload принудительно выполняет повторную загрузку уже загру-  
женного модуля и запускает его программный код. Инструкции присваива-  
ния, выполняемые при повторном запуске, будут изменять существующий   
объект модуля.  
Для чего вся эта суета вокруг повторной загрузки модулей? Функция reload по-  
зволяет изменять части программы, не останавливая всю программу. Благода-  
ря функции reload эффект от изменений в программном коде можно наблюдать   
сразу же после внесения этих изменений. Повторная загрузка модулей помо-  
жет не во всех ситуациях, но она позволит существенно сократить цикл разра-  
ботки. Например, представьте себе программу, предназначенную для работы   
с базами данных, которая должна при запуске соединиться с сервером, – так   
как изменения или настройки могут проверяться немедленно после повторной   
загрузки, вам достаточно соединиться с базой данных всего один раз за весь   
сеанс отладки. Таким же способом можно обновлять программный код долго   
работающих серверов, которые нельзя останавливать.  
Язык Python относится к языкам интерпретирующего типа (более или менее),   
поэтому в нем отсутствуют этапы компиляции/компоновки, необходимые, что-  
бы запустить программу, например, на языке C: модули загружаются динами-  
чески уже запущенной программой. Возможность повторной загрузки обеспе-  
чивает повышение производительности труда, позволяя вам изменять части   
работающей программы без ее остановки. Обратите внимание, что в настоящее   
время функция reload может обслуживать только модули, написанные на языке

636   
Глава 22. Основы программирования модулей   
Python, – скомпилированные модули расширений, написанные на таких язы-  
ках, как C, тоже могут динамически загружаться во время работы программы,   
но их нельзя загрузить повторно.  
Примечание,  касающееся  различий  между  версиями: В версии   
Python 2.6 функция reload доступна в виде встроенной функ-  
ции. В Python 3.0 она была перемещена в модуль imp стандарт-  
ной библиотеки – в версии 3.0 она доступна как imp.reload. Это   
означает, что прежде чем можно будет использовать эту функ-  
цию, необходимо загрузить этот инструмент, выполнив допол-  
нительную инструкцию import или from (только в 3.0). Читатели,   
использующие Python 2.6, могут игнорировать эти инструкции   
импортирования в примерах или оставить их – в версии 2.6 мо-  
дуль imp также содержит функцию reload, с целью упростить   
переход на версию 3.0. Операция повторной загрузки модуля вы-  
полняется одинаково, независимо от того, какая из функций ис-  
пользуется.  
Основы использования функции reload  
В отличие от инструкций import и from:  
 •  
reload – это не инструкция, а функция.  
 •  
Функции reload передается существующий объект модуля, а не имя.  
 •  
В Python 3.0 функции reload находится в модуле imp, который требуется им-  
портировать, чтобы получить доступ к функции.  
Функция reload ожидает получить объект, поэтому к моменту ее вызова мо-  
дуль уже должен быть успешно импортирован (если операция импорта оказа-  
лась неудачной из-за синтаксических или каких-либо других ошибок, вам мо-  
жет потребоваться повторить ее, прежде чем можно будет повторно загрузить   
модуль). Кроме того, синтаксис инструкции import и функции reload отлича-  
ется: аргумент должен передаваться функции reload в круглых скобках, а ин-  
струкции import – без них. Повторная загрузка модуля выполняется примерно   
следующим образом:  
import module # Первоначальное импортирование  
...используются атрибуты модуля...  
... # Теперь выполняются изменения в файле модуля  
...  
from imp import reload # Импортировать функцию reload (в 3.0)  
reload(module) # Загрузить обновленный модуль  
...используются атрибуты модуля...  
Это типичный случай, когда вы импортируете модуль, затем изменяете исход-  
ный программный код в текстовом редакторе, а потом повторно загружаете   
его. Когда вы вызываете функцию reload, интерпретатор повторно читает файл   
с исходными текстами и выполняет инструкции, находящиеся на верхнем   
уровне. Пожалуй, самое важное, что следует знать о функции reload, – это то,   
что она изменяет непосредственно сам объект модуля – она не удаляет и не соз-  
дает его повторно. Вследствие этого все ссылки на объект модуля, имеющиеся   
в программе, автоматически будут учитывать изменения, произошедшие в ре-

Повторная загрузка модулей   
637  
зультате повторной загрузки. А теперь подробнее о том, как происходит по-  
вторная загрузка:   
 •  
Функция reload запускает новый программный код в файле модуля в теку-  
щем пространстве имен модуля. При повторном выполнении программный   
код перезаписывает существующее пространство имен вместо того, чтобы   
удалять его и создавать вновь.  
 •  
Инструкции присваивания на верхнем уровне файла замещают имена но-  
выми значениями. Например, повторный запуск инструкции def приводит   
к замещению предыдущей версии функции в пространстве имен модуля,   
выполняя повторную операцию присваивания имени функции.  
 •  
Повторная загрузка оказывает воздействие на всех клиентов, использо-  
вавших инструкцию import для получения доступа к модулю. Клиенты,   
использовавшие инструкцию import, получают доступ к атрибутам модуля,   
указывая полные их имена, поэтому после повторной загрузки они будут   
получать новые значения атрибутов.  
 •  
Повторная загрузка будет воздействовать лишь на тех клиентов, которые   
еще только будут использовать инструкцию from в будущем. Клиенты, ко-  
торые использовали инструкцию from для получения доступа к атрибутам   
в прошлом, не заметят изменений, произошедших в результате повторной   
загрузки, – они по-прежнему будут ссылаться на старые объекты, получен-  
ные до выполнения перезагрузки.  
Пример использования reload  
Ниже приводится более конкретный пример использования функции reload.   
В следующем примере мы изменяем и повторно загружаем файл модуля без   
остановки интерактивного сеанса работы с интерпретатором Python. Повтор-  
ная загрузка может использоваться в различных других случаях (смотрите   
врезку «Придется держать в уме: повторная загрузка модулей» ниже), но мы   
рассмотрим лишь самый простой пример. Во-первых, в текстовом редакторе   
создайте файл модуля с именем changer.py и добавьте в него следующее содер-  
жимое:  
message = “First version”  
def printer():  
 print(message)  
Этот модуль создает и экспортирует два имени – одно связано со строкой, а дру-  
гое является функцией. Теперь запустите интерпретатор Python, импортируй-  
те модуль и вызовите функцию, которую он экспортирует. Функция выведет   
значение глобальной переменной message:  
% python  
>>> import changer  
>>> changer.printer()  
First version  
Не закрывая интерактивную оболочку интерпретатора, отредактируйте файл   
модуля в другом окне:  
...измените файл changer.py, не останавливая интерактивный сеанс...  
% vi changer.py

638   
Глава 22. Основы программирования модулей   
Измените глобальную переменную message, а также тело функции printer:  
message = “After editing”  
def printer():  
 print(‘reloaded:’, message)  
Затем вернитесь в окно интерактивной оболочки и перезагрузите модуль, что-  
бы выполнить обновленный программный код. Обратите внимание: в следую-  
щем листинге видно, что операция импортирования модуля не дает желаемого   
результата – на экран выводится первоначальный текст сообщения, несмотря   
на то, что файл был изменен. Чтобы задействовать новую версию, необходимо   
вызвать функцию reload:  
...вернитесь обратно в интерактивную оболочку...  
   
>>> import changer  
>>> changer.printer() # Никакого эффекта: используется прежняя версия модуля  
First version  
>>> from imp import reload  
>>> reload(changer) # Принудительная загрузка/выполнение нового кода  
<module ‘changer’ from ‘changer.py’>  
>>> changer.printer() # Теперь будет запущена новая версия  
reloaded: After editing  
Обратите внимание, что функция reload в действительности возвращает объ-  
ект – обычно ее результат игнорируется, но поскольку интерактивная оболочка   
автоматически выводит результат выражения, интерпретатор вывел результат   
в виде строки <module ‘name’...>.  
Придется держать в уме: повторная загрузка модулей  
Помимо возможности перезагружать (и соответственно, перезапускать)   
модули в интерактивной оболочке операция повторной загрузки может   
также использоваться в крупных системах, особенно когда стоимость   
перезапуска всего приложения слишком высока. Например, первыми   
кандидатами на использование возможности динамической перезагруз-  
ки модулей являются системы, которые на запуске соединяются с серве-  
рами сети.  
Эта возможность также может использоваться в приложениях с графи-  
ческим интерфейсом (чтобы изменять действие обработчиков событий   
в графических элементах управления, не закрывая окна графического   
интерфейса) и при использовании Python в качестве встроенного языка   
в программах, написанных на C или C++ (вмещающая программа мо-  
жет вызывать повторную загрузку программного кода на языке Python   
без остановки всего приложения). За более подробным описанием по-  
вторной загрузки обработчиков событий в графическом интерфейсе   
и встроенном программном коде на языке Python обращайтесь к книге   
«Программирование на Python».  
Как правило, повторная загрузка позволяет программам реализовать   
высокодинамичные интерфейсы. Например, Python часто использует-  
ся как язык для настройки больших систем  – пользователи могут на-

В заключение   
639  
страивать программные продукты, изменяя программный код на языке   
Python без необходимости перекомпилировать весь продукт (и даже не   
имея исходных текстов этого продукта). В таких условиях программ-  
ный код на языке Python уже сам по себе добавляет динамичности.  
В более общем случае возможность повторной загрузки позволяет про-  
граммам обеспечивать высокодинамичные интерфейсы. Например,   
Python часто используется как язык для создания сценариев настройки   
в крупных системах – пользователи могут настраивать программные   
продукты под свои нужды за счет создания небольших встраиваемых   
сценариев на языке Python, не пересобирая весь продукт целиком (или   
вообще не имея исходных текстов программ). Возможность внедрения   
сценариев на языке Python в таких системах сама по себе привносит ди-  
намические особенности.  
Тем не менее, чтобы обеспечить еще более высокую динамичность, такие   
системы могут автоматически выполнять повторную загрузку настроеч-  
ного кода на языке Python с заданной периодичностью. В этом случае   
изменения, внесенные пользователями, автоматически вступают в силу   
прямо во время работы системы – нет никакой необходимости останав-  
ливать и перезапускать ее всякий раз, когда изменяется программный   
код на языке Python. Не все системы реализуют такой подход, но для   
тех из них, которые обеспечивают такую возможность, повторная пере-  
загрузка модулей является простым и удобным средством выполнения   
настроек.  
В заключение  
В этой главе были рассмотрены основные инструменты, используемые при   
программировании модулей, – инструкции import и from и функция reload. Мы   
узнали, что инструкция from просто выполняет один дополнительный шаг, на   
котором она копирует имена из файла после того, как он будет импортирован,   
и что функция reload принудительно выполняет операцию импортирования   
файла без остановки и перезапуска интерпретатора Python. Мы также рассмо-  
трели понятия пространства имен, увидели, что происходит при вложенных   
операциях импортирования, узнали, как файлы становятся пространствами   
имен модулей, и познакомились с некоторыми потенциальными ловушками   
инструкции from.  
Мы уже достаточно знаем, чтобы начать работать с файлами модулей в наших   
программах, и тем не менее, в следующей главе приводятся расширенные све-  
дения о модели импортирования – об импортировании пакетов – о способе,   
с помощью которого инструкции import можно указать относительный путь   
к каталогу, где находится требуемый модуль. Как мы увидим, возможность   
импортирования пакетов обеспечивает механизм, удобный для крупных си-  
стем и позволяющий избежать конфликтов между одинаковыми именами мо-  
дулей. Но прежде чем двинуться дальше, постарайтесь ответить на контроль-  
ные вопросы по представленным здесь идеям.

640   
Глава 22. Основы программирования модулей   
Закрепление пройденного  
Контрольные вопросы  
1. Как создать модуль?  
2. Как взаимосвязаны инструкции from и import?  
3. Какое отношение к операции импортирования имеет функция reload?  
4. Когда вместо инструкции from следует использовать инструкцию import?  
5. Назовите три потенциальных ловушки инструкции from.  
6. Какова скорость полета ласточки без груза?  
Ответы  
1. Чтобы создать модуль, достаточно просто создать текстовый файл с ин-  
струкциями на языке Python� любой файл с исходным программным кодом   
автоматически становится модулем – нет никаких синтаксических кон-  
струкций для его объявления. Можно также создать модуль, написав про-  
граммный код на другом языке программирования, таком как C или Java,   
но такие модули находятся вне рассмотрения этой книги.  
2. Инструкция from импортирует модуль целиком, как и инструкция import,   
но кроме этого она еще копирует одно или более имен из импортируемого   
модуля в ту область видимости, где находится инструкция from. Это позво-  
ляет использовать импортированные имена напрямую (name), без дополне-  
ния их именем модуля (module.name).  
3. По умолчанию модуль импортируется один раз за все время выполнения   
программы. Функция reload принудительно выполняет повторное импор-  
тирование. Она часто используется, чтобы загрузить новую версию исход-  
ного программного кода модуля в процессе разработки и в случаях динами-  
ческой настройки.  
4. Инструкция import обязательно должна использоваться вместо инструкции   
from, только когда необходимо обеспечить доступ к одному и тому же имени   
в двух разных модулях, – поскольку вы будете вынуждены указывать име-  
на вмещающих модулей, эти два имени будут уникальны.  
5. Инструкция from может делать непонятным смысл переменной (в каком   
модуле она определена), вызывать проблемы при использовании функ-  
ции reload (имена могут ссылаться на прежние версии объектов) и может   
повреждать пространства имен (может приводить к перезаписи значений   
имен, используемых в вашей области видимости). Самой худшей, во многих   
отношениях, является форма from \* – она может приводить к серьезным по-  
вреждениям пространств имен и скрывать смысл переменных – эту форму   
инструкции следует использовать с большой осторожностью.  
6. А какая ласточка имеется в виду? Африканская или европейская?

Глава 23.  
   
Пакеты модулей  
До сих пор, импортируя модули, мы загружали файлы. Это типичный способ   
использования модулей и, скорее всего, этот прием будет вами использоваться   
наиболее часто в начале вашей карьеры программиста на языке Python. Одна-  
Python. Одна-  
. Одна-  
ко возможности импортирования модулей немного богаче, чем я предлагал вам   
считать до настоящего момента.   
Помимо возможности импортировать имя модуля существует возможность им-  
портировать имена каталогов. Каталог на языке Python называется пакетом,   
поэтому такая операция импортирования называется импортированием паке-  
тов. В действительности, операция импортирования пакета превращает имя   
каталога в еще одну разновидность пространства имен, в котором атрибутам   
соответствуют подкаталоги и файлы модулей, находящиеся в этих каталогах.  
Это немного усложненная особенность, но иерархическая структура, которую   
она создает, оказывается удобной для организации файлов в крупных систе-  
мах и в большинстве случаев упрощает настройку пути поиска модулей. Как   
мы увидим дальше, операция импортирования пакетов иногда оказывается   
просто необходимой, чтобы избежать неоднозначности при наличии несколь-  
ких файлов программ с одинаковыми именами, установленных на одном ком-  
пьютере.  
В этой главе мы также рассмотрим недавно появившийся в языке Python ме-  
ханизм импортирования относительно пакета и его синтаксис, так как оно   
имеет отношение только к программному коду в пакетах. Как мы увидим да-  
лее, этот механизм изменяет путь поиска и расширяет инструкцию from, позво-  
ляя с ее помощью выполнять импортирование имен из пакетов.  
Основы операции импортирования пакетов  
Так как же выполняется импортирование пакетов? В инструкциях import, там,   
где вы указывали имя простого файла, можно указать список имен в пути к ка-  
талогу, разделяя их символами точки:  
import dir1.dir2.mod

642   
Глава 23. Пакеты модулей   
То же самое относится и к инструкции from:  
from dir1.dir2.mod import x  
Предполагается, что такой «точечный» путь в этих инструкциях соответству-  
ет пути через иерархию каталогов на вашей машине, ведущему к файлу mod.  
py (или к файлу с похожим именем – расширение в имени файла может быть   
другим). Таким образом, предыдущие инструкции указывают, что на вашей   
машине имеется каталог dir1, в котором существует подкаталог dir2, в котором   
находится файл модуля mod.py (или с похожим именем).  
Кроме того, эти инструкции предполагают, что каталог dir1 находится внутри   
некоторого контейнерного каталога dir0, который находится в пути поиска   
модулей. Другими словами, обе инструкции импорта предполагают наличие   
структуры каталогов, которая выглядит примерно так, как показано ниже   
(здесь в качестве разделителей имен каталогов используется символ обратного   
слеша, принятый в операционной системе DOS):  
dir0\dir1\dir2\mod.py # Или mod.pyc, mod.so и так далее  
Контейнерный каталог dir0 должен быть добавлен в путь поиска модулей (если   
это не домашний каталог главного файла программы), как если бы имя dir1   
было именем модуля. Инструкция import в вашем сценарии определяет пути,   
ведущие непосредственно к модулям, начиная от этого каталога.  
В любом случае самый левый компонент пути в операции импортирования   
пакета вычисляется относительно некоторого каталога, включенного в путь   
поиска модулей, – в список sys.path, с которым мы познакомились в главе 21.   
То есть инструкции import в вашем сценарии должны содержать полный путь   
к импортируемым модулям относительно этого каталога.  
Пакеты и настройка пути поиска  
Если вы используете эту возможность, имейте в виду, что пути к каталогам   
в инструкции import могут содержать только имена переменных, разделенные   
точками. Здесь нельзя использовать синтаксис путей к каталогам, специфич-  
ный для текущей платформы. Например, C:\dir1.My Documents.dir2 или ../dir1 –   
это недопустимый синтаксис. Напротив, в настройках путей поиска модулей   
используется платформозависимый синтаксис – для именования необходимых   
каталогов-контейнеров.  
Так, в предыдущем примере dir0 – это имя каталога, которое требуется доба-  
вить в путь поиска модулей и которое может иметь произвольную длину и путь,   
с учетом специфики используемой платформы, ведущий к каталогу dir1. Вме-  
сто того, чтобы использовать ошибочный синтаксис, как показано ниже:  
import C:\mycode\dir1\dir2\mod # Ошибка: недопустимый синтаксис  
добавьте путь C:\mycode в переменную окружения PYTHONPATH или в файл .pth   
(предполагается, что это не домашний каталог программы, поскольку в этом   
случае этот шаг не является необходимым) и используйте такую инструкцию:  
import dir1.dir2.mod  
В сущности, записи в списке путей поиска модулей содержат платформозави-  
симые пути к каталогам, которые ведут к самым левым именам в цепочках,

Основы операции импортирования пакетов   
643  
представленных в инструкциях import, а сами инструкции import содержат   
окончание пути к каталогам платформонезависимым способом.1  
Файлы \_\_init\_\_.py пакетов   
Если вы решили использовать импортирование пакетов, существует еще одно   
условие, которое необходимо будет соблюдать: каждый каталог в пути, ука-  
занном в инструкции импортирования пакета, должен содержать файл с име-  
нем \_\_init\_\_.py, в противном случае операция импорта пакета будет терпеть   
неудачу. То есть в примере выше каталоги dir1 и dir2 должны содержать файл   
с именем \_\_init\_\_.py� каталог-контейнер dir0 может не содержать такой файл,   
потому что сам он не указан в инструкции импортирования пакета. Точнее го-  
воря, для такой структуры каталогов:  
dir0\dir1\dir2\mod.py  
и инструкции импортирования, имеющей следующий вид:  
import dir1.dir2.mod  
применяются следующие правила:  
 •  
dir1 и dir2 должны содержать файл \_\_init\_\_.py.  
 •  
dir0, каталог-контейнер, может не содержать файл \_\_init\_\_.py – этот файл   
будет проигнорирован, если он присутствует.  
 •  
dir0, но не dir0\dir1, должен присутствовать в пути поиска модулей (то есть   
он должен быть домашним каталогом или присутствовать в переменной   
окружения PYTHONPATH и так далее).  
Таким образом, структура каталогов в этом примере должна иметь следующий   
вид (здесь отступы указывают на вложенность каталогов):  
dir0\ # Каталог-контейнер в пути поиска модулей  
 dir1\  
 \_\_init\_\_.py  
 dir2\  
 \_\_init\_\_.py  
 mod.py  
Файлы \_\_init\_\_.py могут содержать программный код на языке Python, как   
любые другие файлы модулей. Отчасти они являются объявлениями для ин-  
терпретатора и могут вообще ничего не содержать. Эти файлы, будучи объявле-  
ниями, предотвращают неумышленное сокрытие в каталогах с совпадающими   
именами истинно требуемых модулей, если они отображаются позже в списке   
путей поиска модулей. Без этого защитного механизма интерпретатор мог бы   
1   
Символ точки как разделитель имен каталогов был выбран не только для обеспече-  
ния независимости от используемой платформы, но и потому, что пути в инструкци-  
ях import в действительности становятся вложенными объектами пути. Этот синтак-  
сис также подразумевает, что вы будете получать невразумительные сообщения об   
ошибках, если забудете опустить расширение .py. Например, инструкция import mod.  
py подразумевает, что выполняется импорт пути к каталогу, – она загрузит mod.py,   
затем попытается загрузить mod\py.py, что в конечном счете приведет к появлению   
сбивающего с толку сообщения об ошибке «No module named py» (Модуль с именем   
py не найден).

644   
Глава 23. Пакеты модулей   
выбирать каталоги, которые не имеют никакого отношения к вашему про-  
граммному коду, только лишь потому, что в пути поиска они появляются ранее.  
В общем случае файл \_\_init\_\_.py предназначен для выполнения действий по   
инициализации пакета, создания пространства имен для каталога и реализа-  
ции поведения инструкций from \* (то есть from ... import \*), когда они использу-  
ются для импортирования каталогов:  
Инициализация пакета  
Когда интерпретатор Python импортирует каталог в первый раз, он автома-  
Python импортирует каталог в первый раз, он автома-  
 импортирует каталог в первый раз, он автома-  
тически запускает программный код файла \_\_init\_\_.py этого каталога. По   
этой причине обычно в эти файлы помещается программный код, выпол-  
няющий действия по инициализации, необходимые для файлов в пакете.   
Например, этот файл инициализации в пакете может использоваться для   
создания файлов с данными, открытия соединения с базой данных и так   
далее. Обычно файлы \_\_init\_\_.py не предназначены для непосредственного   
выполнения – они запускаются автоматически, когда выполняется первое   
обращение к пакету.  
Инициализация пространства имен модуля  
При импортировании пакетов пути к каталогам в вашем сценарии после за-  
вершения операции импортирования превращаются в настоящие иерархии   
вложенных объектов. Например, в предыдущем примере после завершения   
операции импортирования можно будет использовать выражение dir1.dir2,   
которое возвращает объект модуля, чье пространство имен содержит все   
имена, определяемые файлом \_\_init\_\_.py из каталога dir2. Такие файлы   
создают пространства имен для объектов модулей, соответствующих ката-  
логам, в которых отсутствуют настоящие файлы модулей.  
Поведение инструкции from \*  
В качестве дополнительной особенности, в файлах \_\_init\_\_.py можно ис-  
пользовать списки \_\_all\_\_, чтобы определить, что будет импортироваться   
из каталога инструкцией from \*. Список \_\_all\_\_ в файлах \_\_init\_\_.py пред-  
ставляет собой список имен субмодулей, которые должны импортировать-  
ся, когда в инструкции from \* указывается имя пакета (каталога). Если   
список \_\_all\_\_ отсутствует, инструкция from \* не будет автоматически за-  
гружать субмодули, вложенные в каталог, – она загрузит только имена,   
определяемые инструкциями присваивания в файле \_\_init\_\_.py, включая   
любые субмодули, явно импортируемые программным кодом в этом фай-  
ле. Например, инструкция from submodule import X в файле \_\_init\_\_.py создаст   
имя X в пространстве имен каталога. (Мы поближе познакомимся со спи-  
ском \_\_all\_\_ в главе 24.)  
Эти файлы можно оставить пустыми, если вам не требуется выполнять специ-  
альных действий. Однако для успешного выполнения операции импортирова-  
ния каталогов они должны существовать обязательно.  
Не путайте файлы \_\_init\_\_.py пакетов с методами-конструк то-  
рами \_\_init\_\_ классов, с которыми мы встретимся в следующей   
части книги. Первые из них – это файлы с программным кодом,   
который выполняется, когда операция импортирования произ-  
водит обход каталогов пакета, тогда как вторые вызываются при   
создании экземпляров классов. Оба они служат для инициали-  
зации, но во всем остальном – это совершенно разные вещи.

Пример импортирования пакета   
645  
Пример импортирования пакета  
Рассмотрим практический пример программного кода, который демонстриру-  
ет, как используются файлы инициализации и пути к каталогам. Следующие   
три файла располагаются в каталоге dir1 и в подкаталоге dir2 – комментарии   
описывают пути к этим файлам:  
# Файл: dir1\\_\_init\_\_.py  
Print(‘dir1 init’)  
x = 1  
   
# Файл: dir1\dir2\\_\_init\_\_.py  
Print(‘dir2 init’)  
y = 2  
   
# Файл: dir1\dir2\mod.py  
Print(‘in mod.py’)  
z = 3  
В данном случае каталог dir1 может быть подкаталогом нашего рабочего ката-  
лога (то есть домашнего каталога программы) или подкаталогом одного из ка-  
талогов, перечисленных в пути поиска модулей (технически, входящего в спи-  
сок sys.path). В любом из этих случаев для каталога, вмещающего подкаталог   
dir1, не требуется наличие файла \_\_init\_\_.py.  
Инструкции import выполняют файлы инициализации в каждом каталоге,   
которые присутствуют в пути к модулю, – инструкции print, присутствующие   
в этих файлах, позволят отследить их выполнение. Кроме того, как и файлы   
модулей, уже импортированные каталоги могут передаваться функции reload   
для принудительного повторного исполнения этого единственного элемента.   
Как показано ниже, для повторной загрузки каталогов и файлов функция re-  
load также может принимать цепочку имен, разделенных точками:  
% python  
>>> import dir1.dir2.mod # Сначала запускаются файлы инициализации  
dir1 init  
dir2 init  
in mod.py  
>>>  
>>> import dir1.dir2.mod # Повторное импортирование не выполняется  
>>>  
>>> from imp import reload # Требуется в версии 3.0  
>>> reload(dir1)  
dir1 init  
<module ‘dir1’ from ‘dir1\\_\_init\_\_.pyc’>  
>>>  
>>> reload(dir1.dir2)  
dir2 init  
<module ‘dir1.dir2’ from ‘dir1\dir2\\_\_init\_\_.pyc’>  
После операции импортирования путь, указанный в инструкции import, стано-  
вится цепочкой вложенных объектов. Здесь mod – это объект, вложенный в объ-  
ект dir2, который в свою очередь вложен в объект dir1:  
>>> dir1  
<module ‘dir1’ from ‘dir1\\_\_init\_\_.pyc’>  
>>> dir1.dir2

646   
Глава 23. Пакеты модулей   
<module ‘dir1.dir2’ from ‘dir1\dir2\\_\_init\_\_.pyc’>  
>>> dir1.dir2.mod  
<module ‘dir1.dir2.mod’ from ‘dir1\dir2\mod.pyc’>  
Каждый каталог в пути фактически становится переменной, которой присваи-  
вается объект модуля, пространство имен которого инициализируется всеми   
инструкциями присваивания в файле \_\_init\_\_.py, находящемся в этом ката-  
логе. Имя dir1.x ссылается на переменную x, которой присваивается значение   
в файле dir1\\_\_init\_\_.py, точно так же, как имя mod.z ссылается на переменную   
z, которой присваивается значение в файле mod.py:  
>>> dir1.x  
1  
>>> dir1.dir2.y  
2  
>>> dir1.dir2.mod.z  
3  
Инструкции from и import для пакетов  
Использование инструкции import могут оказаться несколько неудобным для   
импортирования пакетов, потому что в этом случае далее в программе вам   
придется часто вводить полные пути для обращения к именам. В примере из   
преды дущего раздела, например, приходилось каждый раз вводить полный   
путь от dir1, когда необходимо было обратиться к переменой z. Если попытать-  
ся непосредственно обратиться к dir2 или mod, будет получено сообщение об   
ошибке:  
>>> dir2.mod  
NameError: name ‘dir2’ is not defined  
>>> mod.z  
NameError: name ‘mod’ is not defined  
Поэтому для импортирования пакетов часто более удобно использовать ин-  
струкцию from, чтобы избежать необходимости ввода полного имени при каж-  
дом обращении к нему. Еще более важно следующее: если вы когда-нибудь   
произведете реструктуризацию дерева каталогов, то в случае использования   
инструкции from достаточно будет обновить путь только в самой этой инструк-  
ции, тогда как в случае использования инструкции import придется обновлять   
все обращения к именам в изменившемся пакете. Расширение import as, обсуж-  
даемое в следующей главе, поможет вам определить сокращенные синонимы   
для полных путей:  
% python  
>>> from dir1.dir2 import mod # Описание пути находится только в этом месте  
dir1 init  
dir2 init  
in mod.py  
>>> mod.z # Указывать полный путь не требуется  
3  
>>> from dir1.dir2.mod import z  
>>> z  
3  
>>> import dir1.dir2.mod as mod # Использование короткого синонима  
>>> mod.z  
3

Когда используется операция импортирования пакетов?   
647  
Когда используется операция   
импортирования пакетов?   
Если вы только начинаете осваивать язык Python, то прежде чем переходить   
к использованию пакетов, вам сначала необходимо освоить работу с простыми   
модулями. Пакеты действительно являются полезным инструментом, особен-  
но в крупных программах: они делают операцию импортирования более ин-  
формативной, выступают в роли организационного инструмента, упрощают   
поиск файлов модулей и способны разрешать возникающие неоднозначности.  
Прежде всего, так как операция импортирования пакетов содержит некото-  
рые сведения о структуре каталогов, где находятся файлы программы, они,   
в первую очередь, упрощают поиск файлов и служат организационным ин-  
струментом. Не имея информации о путях к пакетам, вам часто пришлось бы   
обращаться к содержимому пути поиска модулей, чтобы отыскать требуемые   
файлы. Кроме того, если вы организовали размещение своих файлов в дереве   
каталогов по функциональным признакам, то операция импортирования па-  
кетов делает более очевидной роль, которую играют пакеты, что обеспечивает   
более высокую удобочитаемость программного кода. Например, обычная опе-  
рация импорта файла в каталоге, находящемся где-то в пути поиска модулей,   
выглядит так:  
import utilities  
предлагая намного меньше информации, чем операция импорта, включающая   
путь к модулю:  
import database.client.utilities  
Операция импортирования пакетов может также упростить задание перемен-  
ной окружения PYTHONPATH и файлов .pth, хранящих настройки пути поиска   
модулей. Фактически если вы используете импортирование пакетов для всех   
имеющихся каталогов, где хранится ваш программный код, и импорт произво-  
дится относительно общего корневого каталога, вам достаточно будет добавить   
единственную запись в путь поиска модулей: общий корневой каталог. Нако-  
нец, операция импортирования пакетов способна разрешать неоднозначности   
за счет явного и точного указания импортируемых файлов. В следующем раз-  
деле эта роль исследуется более подробно.  
История о трех программах  
Единственный случай, когда операция импортирования действительно необхо-  
дима, – это разрешение неоднозначностей, которые могут возникать, когда на   
одной машине установлено множество программ, содержащих файлы с одина-  
ковыми именами. В определенной степени это проблема установки программ,   
но она может стать источником беспокойств в обычной практике. Давайте рас-  
смотрим гипотетическую ситуацию, чтобы проиллюстрировать эту проблему.  
Предположим, что программист разработал программу на языке Python, кото-  
Python, кото-  
, кото-  
рая содержит файл именем utilities.py, хранящий вспомогательный программ-  
ный код, и файл main.py, используемый для запуска программы. Все файлы   
программы вызывают инструкцию import utilities для загрузки и использо-  
вания общего программного кода. Программа распространяется в виде единого

648   
Глава 23. Пакеты модулей   
архива в формате .tar или .zip, содержащего все файлы программы, и при уста-  
новке все файлы распаковываются в единственный каталог с именем system1:  
system1\  
 utilities.py # Общие вспомогательные функции, классы  
 main.py # Этот файл запускает программу  
 other.py # Импортирует и использует модуль utilities   
Теперь предположим, что другой программист разработал другую программу,   
в которой также имеются файлы utilities.py и main.py, и также используется   
инструкция import utilities во всех файлах программы для загрузки общего   
программного кода. Во время установки этой второй программы на том же са-  
мом компьютере, где уже была установлена первая программа, ее файлы были   
распакованы в новый каталог с именем system2, чтобы не перезаписать одно-  
именные файлы первой программы:  
system2\  
 utilities.py # Общие вспомогательные функции  
 main.py # Этот файл запускает программу  
 other.py # Импортирует модуль utilities   
Пока что никаких явных проблем не наблюдается: обе программы прекрасно   
сосуществуют и работают на одной и той же машине. Фактически для этих   
программ вам даже не нужно настраивать путь поиска модулей, потому что ин-  
терпретатор всегда начинает поиск модулей с домашнего каталога программы   
(то есть с каталога, в котором находится главный файл программы), операции   
импортирования в любой из этих программ автоматически будут находить все   
необходимые файлы в домашнем каталоге программы. Например, если запу-  
скается файл system1\main.py, все операции импортирования сначала будут   
просматривать каталог system1. Точно так же при запуске файла system2\main.  
py в первую очередь будет просматриваться каталог system2. Не забывайте, что   
настраивать путь поиска модулей необходимо только при необходимости им-  
портировать модули из разных каталогов.  
А теперь предположим, что после установки этих двух программ вы решили   
использовать вспомогательные функции из обоих файлов utilities.py в своей   
собственной программе. В конце концов, это обычный вспомогательный про-  
граммный код, а для программного кода на языке Python вполне естественно,   
когда он используется многократно. В такой ситуации вам необходима воз-  
можность из своего программного кода, хранящегося в третьем каталоге, за-  
грузить один из двух файлов:  
import utilities  
utilities.func(‘spam’)  
Теперь проблема начинает вырисовываться. Чтобы вообще выполнить эту ра-  
боту, вам придется включить в путь поиска модулей каталоги, содержащие   
файлы utilities.py. Но какой каталог поместить первым – system1 или system2?  
Проблема заключается в линейной природе пути поиска. Он всегда просматри-  
вается слева направо. Независимо от того, как долго вы будете ломать голову   
над этой проблемой, вы всегда будете получать файл utilities.py из каталога,   
который находится в пути поиска раньше (левее). Как следствие, вы никогда   
не сможете импортировать одноименный файл из другого каталога. Вы можете   
попытаться изменять sys.path в своей программе перед каждой операцией им-

Когда используется операция импортирования пакетов?   
649  
портирования, но это сложная работа и при ее выполнении легко ошибиться.   
По умолчанию проблема оказывается для вас неразрешимой.  
Эту проблему можно решить с помощью пакетов. Вместо того, чтобы устанав-  
ливать программы как плоские списки файлов в независимые каталоги, мож-  
но установить их в подкаталоги с общим корнем. Например, можно было бы   
организовать установку всего программного кода из этого примера в виде сле-  
дующей иерархии:  
root\  
 system1\  
 \_\_init\_\_.py  
 utilities.py  
 main.py  
 other.py  
 system2\  
 \_\_init\_\_.py  
 utilities.py  
 main.py  
 other.py  
 system3\ # Здесь или в другом месте  
 \_\_init\_\_.py # располагается ваш новый программный код  
 myfile.py  
Теперь достаточно просто добавить общий корневой каталог в путь поиска мо-  
дулей. Если выполнять импортирование относительно этого общего корня,   
можно будет с помощью операции импортирования пакетов импортировать   
любой файл из любой программы – использование имени вмещающего катало-  
га делает путь (а, значит, и ссылку на модуль) уникальным. Фактически мож-  
но даже импортировать обе утилиты сразу в одном и том же модуле, при усло-  
вии, что вы будете использовать инструкцию import и при каждом обращении   
к именам будете указывать полный путь к вспомогательным модулям:  
import system1.utilities  
import system2.utilities  
system1.utilities.function(‘spam’)  
system2.utilities.function(‘eggs’)  
В данном случае имя вмещающего каталога обеспечивает уникальность ссы-  
лок на модули.  
Обратите внимание, что вместо инструкции from необходимо использовать ин-  
струкцию import, только если вам необходимо получить доступ к двум или более   
одноименным атрибутам. Если бы имена вызываемых здесь функций различа-  
лись, можно было бы использовать инструкцию from, чтобы избежать необходи-  
мости всякий раз вводить полные пути к пакетам, как уже описывалось выше.  
Кроме того, следует заметить, что в приведенной выше иерархии установки   
файлы \_\_init\_\_.py были добавлены в каталоги установки программ system1   
и system2, но не в корневой каталог root. Этот файл требуется помещать только   
в каталоги, перечисленные в инструкциях import, – как вы наверняка помните,   
они автоматически выполняются интерпретатором при первой попытке про-  
граммы импортировать каталог пакета.  
Технически, каталог system3 не обязательно должен находиться в каталоге   
root – здесь должны находиться только каталоги, откуда производится импорт

650   
Глава 23. Пакеты модулей   
пакетов. Однако, поскольку ваши собственные модули могут когда-нибудь   
быть использованы в других программах, было бы желательно поместить их   
в общий корневой каталог, чтобы избежать подобных проблем в будущем.  
Наконец, обратите внимание, что операции импорта в обеих оригинальных   
программах продолжают действовать без изменений. Поскольку в этих про-  
грамма поиск модулей производится в первую очередь в их домашних ката-  
логах, добавление общего корневого каталога в путь поиска модулей никак не   
отражается на программном коде в system1 и system2 – они по-прежнему могут   
использовать инструкции import utilities и получать в ответ свои собственные   
файлы. Кроме того, если вы предусмотрительно будете устанавливать все про-  
граммы на языке Python в общий корневой каталог, как в данном примере, на-  
Python в общий корневой каталог, как в данном примере, на-  
 в общий корневой каталог, как в данном примере, на-  
стройка пути поиска станет элементарной: вам достаточно будет один раз до-  
бавить в него общий корневой каталог.  
Импортирование относительно пакета  
При описании операции импортирования пакетов мы пока рассматривали воз-  
можность импортирования файлов пакетов из-за пределов этих пакетов. При   
импортировании файлов пакета внутри самого пакета можно использовать тот   
же синтаксис, как и при импортировании из-за пределов пакета, но можно так-  
же использовать специальные правила поиска модулей внутри пакета, позво-  
ляющие упростить инструкции import. То есть вместо того, чтобы указывать   
полный путь к модулю пакета, можно использовать форму относительного   
пути внутри пакета.  
На сегодняшний день принцип действия операции импортирования относи-  
тельно пакета зависит от версии Python: в Python 2.6 операция импортирова-  
Python: в Python 2.6 операция импортирова-  
: в Python 2.6 операция импортирова-  
Python 2.6 операция импортирова-  
 2.6 операция импортирова-  
ния неявно выполняет поиск в каталогах пакета, тогда как в версии 3.0 необ-  
ходимо явно использовать синтаксис относительного импортирования. Это из-  
менение в версии 3.0 помогает повысить удобочитаемость программного кода,   
делая операцию импортирования модулей из того же пакета более очевидной.   
Если вы приступаете к использованию языка Python, начиная с версии 3.0, ве-  
Python, начиная с версии 3.0, ве-  
, начиная с версии 3.0, ве-  
роятно, вам следует сосредоточить свое внимание в этом разделе на новом син-  
таксисе операции импортирования. Если же вы используете пакеты на языке   
Python предыдущих версий, возможно, вам также будет интересно, чем отли-  
 предыдущих версий, возможно, вам также будет интересно, чем отли-  
чается механизм импортирования в версии 3.0 от более ранних версий.  
Изменения в Python 3.0  
Принцип действия операции импортирования внутри пакетов немного изме-  
нился в Python 3.0. Изменения коснулись лишь импортирования файлов паке-  
Python 3.0. Изменения коснулись лишь импортирования файлов паке-  
 3.0. Изменения коснулись лишь импортирования файлов паке-  
та из файлов, находящихся в каталогах этого же пакета, о котором мы говорим   
в этой главе, – операция импортирования других файлов действует, как и пре-  
жде. В Python 3.0 в операцию импортирования внутри пакетов было внесено   
два изменения:  
 •  
Изменилась семантика пути поиска модулей так, что теперь операция им-  
портирования модуля по умолчанию пропускает собственный каталог па-  
кета. Она проверяет только компоненты пути поиска. Эта операция называ-  
ется импортированием по «абсолютному» пути.

Импортирование относительно пакета   
651  
 •  
Расширен синтаксис инструкции from так, что теперь имеется возможность   
явно указать, что поиск импортируемых модулей должен производиться   
только в каталоге пакета. Эта операция называется импортированием по   
«относительному» пути.  
Эти изменения были полностью реализованы в Python 3.0. Новый синтаксис   
операции импортирования относительно пакета доступен также в Python 2.6,   
но по умолчанию изменения в семантике пути поиска отключены и их требует-  
ся активировать. В настоящее время эти изменения предполагается добавить   
в версию 2.71 – такое поэтапное изменение обусловлено тем, что изменения се-  
мантики пути поиска нарушают обратную совместимость с более ранними вер-  
сиями Python.  
Суть этих изменений в версии 3.0 (и в 2.6, если они используются) состоит   
в том, что вы должны использовать специальный синтаксис инструкции from   
для импортирования модулей, находящихся в том же пакете, что и импорти-  
рующий модуль, если вы не указываете полный путь к модулю, начиная от   
корневого каталога пакета. Если не использовать этот синтаксис, интерпрета-  
тор не сможет отыскать требуемый модуль в пакете.  
Основы импортирования по относительному пути  
В обеих версиях, Python 3.0 и 2.6, инструкции from теперь могут использовать   
точки («.»), чтобы указать, что поиск модулей в первую очередь должен про-  
изводиться в том же самом пакете (эта особенность известна как импортиро-  
вание относительно пакета), а не где-то в другом месте, в пути поиска (эта   
особенность называется импортирование по абсолютному пути). То есть:  
 •  
В обеих версиях Python, 3.0 и 2.6, в инструкции from в начале пути мож-  
но использовать точки, чтобы указать, что импорт должен производиться   
относительно вмещающего пакета, – при таком способе импортирования   
поиск модулей будет производиться только внутри пакета, а модули с теми   
же именами, находящиеся где-то в пути поиска (sys.path), будут недоступ-  
ны. Благодаря этому модули внутри пакета получают преимущество перед   
модулями за его пределами.  
 •  
В Python 2.6 обычная операция импортирования в программном коде па-  
кета (без точек) в настоящее время по умолчанию выполняется в порядке   
«сначала поиск относительно пакета, потом – абсолютный поиск». То есть   
поиск сначала производится в каталоге пакета и только потом в пути по-  
иска. Однако в Python 3.0 по умолчанию выполняется импортирование по   
абсолютному пути – при отсутствии точек операции импортирования про-  
пускают вмещающий пакет и пытаются отыскать импортируемые модули   
в пути поиска sys.path.   
Например, в обеих версиях Python, 3.0 и 2.6, инструкция вида:  
from . import spam # Импортирование относительно текущего пакета  
1   
Это не опечатка� действительно будет выпущена версия 2.7 и, возможно, 2.8 и даже   
более новые версии в ветке 2.X, параллельно с новыми версиями в ветке 3.X. Как   
уже говорилось в предисловии, обе ветки, Python 2 и Python 3, будут поддерживать-  
ся параллельно еще на протяжении нескольких лет, чтобы обеспечить поддержку   
большому количеству пользователей Python 2 и существующих программ.

652   
Глава 23. Пакеты модулей   
предписывает интерпретатору импортировать модуль с именем spam, располо-  
женный в том же пакете, что и файл, где находится эта инструкция. Аналогич-  
но, следующая инструкция:  
from .spam import name  
означает: «из модуля с именем spam, расположенного в том же пакете, что   
и файл, где находится эта инструкция, импортировать переменную name».  
Поведение инструкции без начальной точки зависит от используемой версии   
Python. В версии 2.6 такая инструкция импортирования по умолчанию также   
будет использовать порядок поиска «сначала относительно пакета, а затем – аб-  
солютный поиск» (то есть сначала поиск выполняется в каталоге пакета), если   
только в импортирующий файл не будет включена следующая инструкция:  
from \_\_future\_\_ import absolute\_import # Обязательно до версии 2.7?  
Если эта инструкция присутствует, она включает использование абсолютного   
пути поиска, которое по умолчанию используется в Python 3.0.   
В Python 3.0 все операции импортирования без дополнительных точек никогда   
не пытаются отыскать модуль внутри пакета и производят поиск по абсолютно-  
му пути, хранящемуся в списке sys.path. Например, когда задействован меха-  
низм импортирования в версии 3.0, следующая инструкция всегда будет нахо-  
дить не модуль string в текущем пакете, а одноименный модуль в стандартной   
библиотеке:  
import string # Пропустит поиск модуля в пакете  
Без инструкции from \_\_future\_\_ в Python 2.6 всегда будет импортироваться мо-  
дуль string из пакета. Чтобы получить то же поведение в версии 3.0 и 2.6, ког-  
да по умолчанию выполняется импорт по абсолютному пути, для выполнения   
операции импортирования относительно пакета можно использовать следую-  
щую форму инструкции:  
from . import string # Поиск выполняется только в пределах пакета  
На сегодняшний день этот прием работает в обеих версиях Python, 2.6 и 3.0.   
Единственное отличие модели импортирования в версии 3.0 состоит в том, что   
она является обязательной, когда требуется по простому имени загрузить мо-  
дуль, находящийся в том же каталоге пакета, что и файл, откуда производится   
импортирование.  
Обратите внимание: ведущий символ точки может использоваться только в ин-  
струкции from – в инструкции import он недопустим. В Python 3.0 инструкция   
import modname всегда выполняет импортирование по абсолютному пути, пропу-  
ская поиск в каталоге пакета. В Python 2.6 она по-прежнему выполняет импорт   
по относительному пути (то есть сначала она просматривает каталог пакета),   
но в Python 2.7 она будет выполнять импорт по абсолютному пути. Инструк-  
ции from без ведущей точки ведут себя точно так же, как инструкции import, –   
в версии 3.0 они выполняют импортирование по абсолютному пути (пропуская   
каталог пакета), а в версии 2.6 они выполняют поиск «сначала относительно   
пакета, а затем – абсолютный поиск» (поиск в каталоге пакета выполняется   
в первую очередь).  
Возможны также и другие варианты точечной нотации для ссылки на модули   
в пакете. Допустим, что имеется каталог mypkg пакета, тогда следующие аль-

Импортирование относительно пакета   
653  
тернативные варианты импортирования внутри этого пакета будут работать   
так, как описывается:  
from .string import name1, name2 # Импорт имен из mypkg.string  
from . import string # Импорт mypkg.string  
from .. import string # Импорт string из родительского каталога  
Чтобы лучше понять эти последние формы инструкций, необходимо разобрать-  
ся с обоснованием этого изменения.  
Зачем необходим импорт относительно пакета?   
Эта возможность предназначена, чтобы дать сценариям возможность ликвиди-  
ровать возникающие неоднозначности, которые могут возникать, когда в раз-  
ных местах в пути поиска присутствует несколько одноименных модулей. Рас-  
смотрим следующий каталог пакета:  
mypkg\  
 \_\_init\_\_.py  
 main.py  
 string.py  
Это каталог пакета с именем mypkg, содержащий модули mypkg.main и mypkg.  
string. Теперь предположим, что модуль main пытается импортировать модуль   
с именем string. В Python 2.6 и в более ранних версиях интерпретатор будет   
сначала искать модуль в каталоге mypkg, выполняя импорт относительно па-  
кета. Он найдет и импортирует файл string.py, находящийся в этом каталоге,   
и присвоит его имени string в пространстве имен модуля mypkg.main.  
Однако может так получиться, что этой инструкцией предполагалось импор-  
тировать модуль string из стандартной библиотеки языка Python. К сожале-  
нию, в этих версиях Python нет достаточно простого способа проигнорировать   
модуль mypkg.string и импортировать модуль string из стандартной библиотеки,   
расположенной в пути поиска модулей. Кроме того, мы не сможем решить эту   
проблему с помощью инструкции импортирования пакетов, потому что мы не   
можем зависеть от структуры каталогов пакета, описанных выше, стандарт-  
ной библиотеки, присутствующей на любом компьютере.  
Другими словами, инструкции импортирования в пакетах могут быть неодно-  
значными – внутри пакета может быть непонятно, какой модуль пытается   
импортировать инструкция import spam, – внутри пакета или за его предела-  
ми. Если говорить более точно, локальный модуль или пакет могут сделать не-  
возможным импорт другого модуля, присутствующего в пути поиска sys.path,   
преднамеренно или нет.  
На практике пользователи Python могут избегать использовать имена модулей   
стандартной библиотеки для своих модулей (если вам требуется стандартный   
модуль string, не называйте свой модуль этим именем!). Но это не поможет,   
если пакет делает недоступным стандартный модуль случайно. Кроме того,   
с течением времени в стандартную библиотеку Python могут добавляться но-  
вые модули – с теми же именами, которые присвоены вашим уже существую-  
щим модулям. Программный код, использующий особенности импорта отно-  
сительно пакетов, сложнее понять, потому что бывает трудно выяснить, какой   
модуль импортируется. Гораздо лучше, если решение явно описывается в про-  
граммном коде.

654   
Глава 23. Пакеты модулей   
Решение проблемы с импортированием   
относительно пакета в 3.0  
С целью разрешить эту дилемму поведение операции импортирования внутри   
пакетов в Python 3.0 (и в виде дополнительной возможности в 2.6) было измене-  
Python 3.0 (и в виде дополнительной возможности в 2.6) было измене-  
 3.0 (и в виде дополнительной возможности в 2.6) было измене-  
но так, что теперь она выполняет импорт только по абсолютному пути. Соглас-  
но этой модели следующая инструкция import, находящаяся в нашем файле   
mypkg/main.py, всегда будет находить модуль string за пределами пакета, за   
счет использования схемы поиска модулей по абсолютному пути sys.path:  
import string # Импортирует модуль string за пределами пакета  
Инструкция from, в которой не используется синтаксис с ведущей точкой, так-  
же выполняет импорт по абсолютному пути:  
from string import name # Импортирует имя name из модуля string   
 # за пределами пакета  
Однако, если вы действительно хотите импортировать модуль из своего пакета,   
не указывая полный путь, начиная от корневого каталога пакета, можно вос-  
пользоваться синтаксисом инструкции from:  
from . import string # Импортирует mypkg.string (относительно пакета)  
Данная форма инструкции пытается импортировать модуль string только из   
текущего пакета и является относительным эквивалентом абсолютной формы   
инструкции import из предыдущего примера – когда используется специаль-  
ный синтаксис относительного импортирования, поиск модулей выполняется   
только в каталоге пакета.  
Кроме того, мы можем копировать имена из модуля, используя синтаксис им-  
портирования относительно пакета:  
from .string import name1, name2 # Импортирует имена из mypkg.string  
Эта инструкция также ссылается на модуль string в текущем пакете. Если по-  
местить эту инструкцию в модуль mypkg.main, например, она будет импортиро-  
вать имена name1 и name2 из модуля mypkg.string.   
По сути, символ точки «.» в инструкции относительного импорта представляет   
каталог в пакете, содержащий файл, где выполняется операция импортирова-  
ния. Дополнительная начальная точка предписывает выполнить относитель-  
ный импорт, начиная с родительского каталога текущего пакета. Например,   
инструкция:  
from .. import spam # Импортирует модуль одного уровня с пакетом mypkg  
загрузит модуль, находящийся на том же уровне в иерехии каталогов, что и па-  
кет mypkg, – то есть модуль spam, находящийся в каталоге, родительском по от-  
ношению к пакету mypkg. В общем случае действия программного кода в модуле   
A.B.C будут следующими:  
from . import D # Импортирует A.B.D (. означает A.B)  
from .. import E # Импортирует A.E (.. означает A)  
   
from .D import X # Импортирует A.B.D.X (. означает A.B)  
from ..E import X # Импортирует A.E.X (.. означает A)

Импортирование относительно пакета   
655  
Импорт в пакетах по относительному и абсолютному пути  
Как вариант, в файле может явно указываться имя его пакета в инструкции   
импортирования по абсолютному пути. Например, следующая инструкция   
найдет пакет mypkg по абсолютному пути в sys.path:  
from mypkg import string # Импортирует mypkg.string (по абсолютному пути)  
Однако результат этой инструкции зависит от настроек и от порядка следова-  
ния каталогов в пути поиска модулей, тогда как форма импортирования с точ-  
кой, относительно текущего пакета, такой зависимости не имеет. Фактически   
чтобы иметь возможность использовать данную форму импортирования, тре-  
буется включить путь к каталогу пакета mypkg в путь поиска модулей. Вообще   
говоря, инструкция поиска по абсолютному пути должна просмотреть все ка-  
талоги, находящиеся левее каталога пакета в sys.path, когда имя пакета указы-  
вается явно, как в следующей инструкции:  
from system.section.mypkg import string # sys.path содержит только system  
В крупных пакетах или в пакетах с глубокой вложенностью такое импортиро-  
вание будет выполняться дольше, чем в случае использования точки:  
from . import string # Синтаксис импортирования относительно текущего пакета  
При использовании этой последней формы инструкции поиск автоматически   
будет выполняться только в объемлющем пакете, независимо от настройки   
пути поиска.  
Правила импортирования по относительному пути  
Операция импортирования при первой встрече может показаться немного за-  
мысловатой, но разобраться в ней вам помогут следующие ключевые моменты:  
 •  
Операция импортирования по относительному пути применяется исклю-  
чительно для импортирования внутри пакета. Имейте в виду, что изме-  
нение семантики пути поиска модулей действует только в инструкциях   
импортирования внутри модуля, помещенного в пакет. Обычная операция   
импортирования в файлах за пределами пакетов действует точно так, как   
было описано выше, – она в первую очередь выполняет поиск в каталоге,   
содержащем главный файл сценария.  
 •  
Импортирование по относительному пути возможно только с помощью ин-  
струкции from. Запомните также, что новый синтаксис может применяться   
только в инструкциях from и считается недопустимым в инструкциях im-  
port. Требование импортирования по относительному пути определяется   
по тому, что имя модуля в инструкции from начинается с одной или более   
точек. Если в инструкции указано имя, содержащее точки, но не начинаю-  
щееся с точек, она интерпретируется как инструкция импортирования па-  
кета, а не как инструкция импортирования по относительному пути.  
 •  
Неоднозначность терминологии. Честно признаться, терминология, ис-  
Честно признаться, терминология, ис-  
пользуемая для описания этой особенности, создает больше путаницы, чем   
должна бы. В действительности, все операции импортирования выполня-  
ются относительно чего-нибудь. За пределами пакетов импортирование вы-  
полняется относительно каталогов, перечисленных в пути поиска модулей   
sys.path. Как мы узнали в главе 21, этот путь включает в себя домашний

656   
Глава 23. Пакеты модулей   
каталог программы, значение переменной окружения PYTHONPATH, каталоги   
из файлов .pth и каталоги стандартной библиотеки. При работе в интерак-  
тивной оболочке домашним каталогом программы считается текущий ра-  
бочий каталог.  
Операции импорта внутри пакета в 2.6 расширяют это поведение и выпол-  
няют поиск модулей сначала внутри самого пакета. Единственное, что из-  
менилось в модели поиска в версии 3.0, – обычная инструкция импорти-  
рования по «абсолютному» пути пропускает каталог пакета, а инструкции   
импорта со «специальным» синтаксисом импортирования относительно   
текущего пакета выполняют поиск только в пределах пакета. Когда мы го-  
ворим об импортировании по «абсолютному» пути в версии 3.0, мы в дей-  
ствительности подразумеваем, что импортирование будет выполняться от-  
носительно каталогов, перечисленных в sys.path, но не внутри самого паке-  
та. И наоборот, когда мы говорим об импортировании «относительно» теку-  
щего пакета, мы подразумеваем, что импортирование будет выполняться   
только внутри каталога пакета. Конечно, пути к каталогам в sys.path могут   
быть не только абсолютными, но и относительными. (Этим утверждени-  
ем я, возможно, запутал вас еще больше, но не волнуйтесь, это была всего   
лишь разминка!)  
Другими словами, реализация «импортирования относительно текущего   
пакета» в 3.0 в действительности сводится лишь к тому, что из реализации   
импортирования для пакетов версии 2.6 был убран просмотр каталогов   
в пути поиска, и в инструкцию from был добавлен специальный синтаксис,   
явно указывающий, что операция импортирования выполняется отно-  
сительно текущего пакета. Если ранее вам приходилось писать инструк-  
ции импортирования в версии 2.6, не зависящие от неявного поведения   
инструкции импортирования в пакетах (например, за счет повсеместного   
использования полных путей, начиная от корневого каталога пакета), это   
изменение может показаться вам спорным. Если вы поступали иначе, вам   
придется внести изменения в свои пакеты и использовать новый синтаксис   
в инструкциях from, импортирующих модули из локального пакета.  
Правила поиска модулей  
Полное представление о природе импортирования пакетов и импортирования   
относительно текущего пакета в Python 3.0 дают следующие правила:  
 •  
Для простых имен пакетов (например, A) поиск выполняется во всех катало-  
гах, перечисленных в списке sys.path, слева направо. Этот список конструи-  
руется из системных значений по умолчанию и из настроек пользователя.  
 •  
Пакеты – это обычные каталоги с модулями на языке Python, содержащие   
специальный файл \_\_init\_\_.py, который позволяет указывать в инструк-  
циях импортирования цепочки каталогов вида A.B.C. Чтобы получить воз-  
можность импортировать, например, A.B.C, каталог A должен находиться   
в одном из каталогов, перечисленных в пути поиска модулей sys.path, B дол-  
жен быть подкаталогом пакета в каталоге A, а C должен быть модулем или   
другим компонентом в каталоге B, доступным для импортирования.  
 •  
Внутри файлов пакета обычные инструкции import выполняют поиск моду-  
лей в sys.path в соответствии с теми же правилами, что и инструкции им-  
портирования в любых других модулях. Однако при импортировании с ис-

Импортирование относительно пакета   
657  
пользованием инструкций from и начальных точек в именах поиск выпол-  
няется относительно текущего пакета – то есть поиск производится только   
в каталоге текущего пакета, а обычный поиск в sys.path не выполняется.   
Инструкция from . import A, например, ограничится поиском модуля в ката-  
логе, содержащем файл, где находится эта инструкция.  
Относительный импорт в примерах  
Но достаточно теории: рассмотрим несколько примеров, демонстрирующих   
концепции, на которых основана операция импортирования относительно па-  
кета.  
Импортирование за пределами пакетов  
Прежде всего, как уже упоминалось выше, данная особенность не оказывает   
влияния на операцию импортирования за пределами пакета. То есть следую-  
щий пример загрузит модуль string из стандартной библиотеки, как и предпо-  
лагалось:  
C:\test> c:\Python30\python  
>>> import string  
>>> string  
<module ‘string’ from ‘c:\Python30\lib\string.py’>  
Но если в текущий рабочий каталог добавить модуль с тем же именем, будет   
загружен он, вместо библиотечного модуля, потому что текущий рабочий ката-  
лог стоит на первом месте в пути поиска:  
# test\string.py  
print(‘string’ \* 8)  
   
C:\test> c:\Python30\python  
>>> import string  
stringstringstringstringstringstringstringstring  
>>> string  
<module ‘string’ from ‘string.py’>  
Другими словами, обычная операция импортирования выполняется относи-  
тельно «домашнего» каталога (где находится главный сценарий, или относи-  
тельно текущего рабочего каталога). В действительности синтаксис импорти-  
рования относительно текущего пакета считается недопустимым для исполь-  
зования в файлах, не входящих в состав пакета:  
>>> from . import string  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
ValueError: Attempted relative import in non-package  
В этом и во всех остальных примерах, что приводятся в этом разделе, инструк-  
ции, которые вводятся в интерактивной оболочке, ведут себя точно так же, как   
они вели бы себя, находясь на верхнем уровне сценария. Это обусловлено тем,   
что первый элемент списка sys.path соответствует либо текущему рабочему ка-  
талогу интерактивной оболочки, либо каталогу, содержащему главный файл   
программы. Единственное отличие состоит в том, что во втором случае первый   
элемент списка sys.path содержит не пустую строку, а абсолютный путь к ка-  
талогу:

658   
Глава 23. Пакеты модулей   
# test\main.py  
import string  
print(string)  
   
C:\test> C:\python30\python main.py # Тот же результат получается в 2.6  
stringstringstringstringstringstringstringstring  
<module ‘string’ from ‘C:\test\string.py’>  
Импортирование внутри пакетов  
Теперь избавимся от локального модуля string, находящегося в текущем ра-  
бочем каталоге, и создадим каталог пакета с двумя модулями, включая обяза-  
тельный, но пустой файл test\pkg\\_\_init\_\_.py (который я опустил здесь):  
C:\test> del string\*  
C:\test> mkdir pkg  
   
# test\pkg\spam.py  
import eggs # <== Работает в 2.6, но не в 3.0!  
print(eggs.X)  
   
# test\pkg\eggs.py  
X = 99999  
import string  
print(string)  
Первый файл в этом пакете пытается импортировать второй с помощью обыч-  
ной инструкции import. Поскольку в версии 2.6 эта инструкция выполняет им-  
портирование относительно текущего пакета, а в версии 3.0 – импортирование   
по абсолютному пути, в будущем она будет терпеть неудачу. То есть в версии 2.6   
она сначала выполнит поиск в каталоге пакета, а в версии 3.0 – нет. Вы должны   
помнить об этой несовместимости в версии 3.0:  
C:\test> c:\Python26\python  
>>> import pkg.spam  
<module ‘string’ from ‘c:\Python26\lib\string.pyc’>  
99999  
   
C:\test> c:\Python30\python  
>>> import pkg.spam  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
 File “pkg\spam.py”, line 1, in <module>  
 import eggs  
ImportError: No module named eggs  
Чтобы эта операция работала в обеих версиях, 2.6 и 3.0, в первом файле не-  
обходимо использовать специальный синтаксис операции импортирования от-  
носительно текущего пакета, чтобы в версии 3.0 обеспечить поиск в каталоге   
пакета:  
# test\pkg\spam.py  
from . import eggs # <== Используйте операцию импорта относительно print(eggs.X)   
# текущего пакета в 2.6 или 3.0  
   
# test\pkg\eggs.py  
X = 99999  
import string  
print(string)

Импортирование относительно пакета   
659  
C:\test> c:\Python26\python  
>>> import pkg.spam  
<module ‘string’ from ‘c:\Python26\lib\string.pyc’>  
99999  
   
C:\test> c:\Python30\python  
>>> import pkg.spam  
<module ‘string’ from ‘c:\Python30\lib\string.py’>  
99999  
Импорт по-прежнему выполняется   
относительно текущего рабочего каталога  
Обратите внимание, что модули пакета по-прежнему могут импортировать   
модули из стандартной библиотеки, такие как string. В действительности ин-  
струкции импортирования в этих модулях по-прежнему выполняют операцию   
относительного импортирования, но уже относительно элементов пути поиска,   
даже если сами эти элементы содержат относительные пути. Если теперь снова   
добавить модуль string в текущий рабочий каталог, операции импортирования   
будут обнаруживать его в пакете, а не в стандартной библиотеке. В версии 3.0   
имеется возможность выполнить импортирование по абсолютному пути, про-  
пустив каталог пакета, но невозможно пропустить домашний каталог про-  
граммы, которая импортирует пакет:  
# test\string.py  
print(‘string’ \* 8)  
   
# test\pkg\spam.py  
from . import eggs  
print(eggs.X)  
   
# test\pkg\eggs.py  
X = 99999  
import string # <== Импортирует модуль string из текущего рабочего   
print(string) # каталога, а не из стандартной библиотеки!  
   
C:\test> c:\Python30\python # Тот же результат будет получен в 2.6  
>>> import pkg.spam  
stringstringstringstringstringstringstringstring  
<module ‘string’ from ‘string.py’>  
99999  
Выбор модулей операциями импортирования   
по относительному и абсолютному пути  
Чтобы показать, как это относится к операциям импортирования модулей из   
стандартной библиотеки, изменим пакет еще раз. Удалите локальный модуль   
string и создайте новый внутри пакета:  
C:\test> del string\*  
   
# test\pkg\spam.py  
import string # <== Относительная в 2.6, абсолютная в 3.0  
print(string)  
   
# test\pkg\string.py  
print(‘Ni’ \* 8)

660   
Глава 23. Пакеты модулей   
Теперь выбор, какая версия модуля будет импортирована, зависит от того, ка-  
кая версия Python используется. Как и прежде, в Python 3.0 операция импор-  
Python используется. Как и прежде, в Python 3.0 операция импор-  
 используется. Как и прежде, в Python 3.0 операция импор-  
Python 3.0 операция импор-  
 3.0 операция импор-  
тирования в первом файле будет интерпретироваться, как импортирование по   
абсолютному пути, и она пропустит каталог пакета, а в Python 2.6 – нет:  
C:\test> c:\Python30\python  
>>> import pkg.spam  
<module ‘string’ from ‘c:\Python30\lib\string.py’>  
   
C:\test> c:\Python26\python  
>>> import pkg.spam  
NiNiNiNiNiNiNiNi  
<module ‘pkg.string’ from ‘pkg\string.py’>  
Использование синтаксиса импортирования относительно текущего пакета   
вынудит интерпретатор версии 3.0 выполнить поиск в пакете, как это делается   
в версии 2.6. Используя синтаксис импортирования по абсолютному или от-  
носительному пути в версии 3.0, вы можете явно указать, как следует посту-  
пить – пропустить или просмотреть каталог пакета. Фактически так действует   
модель импортирования в версии 3.0:  
# test\pkg\spam.py  
from . import string # <== Относительная в обеих версиях, 2.6 и 3.0  
print(string)  
   
# test\pkg\string.py  
print(‘Ni’ \* 8)  
   
C:\test> c:\Python30\python  
>>> import pkg.spam  
NiNiNiNiNiNiNiNi  
<module ‘pkg.string’ from ‘pkg\string.py’>  
   
C:\test> c:\Python26\python  
>>> import pkg.spam  
NiNiNiNiNiNiNiNi  
<module ‘pkg.string’ from ‘pkg\string.py’>  
Важно отметить, что синтаксис импортирования относительно текущего паке-  
та в действительности является не просто объявлением предпочтений, а обя-  
зывающей декларацией. Если мы удалим файл string.py и снова запустим этот   
пример, инструкция импортирования относительного текущего пакета в фай-  
ле spam.py будет терпеть неудачу в обеих версиях, 3.0 и 2.6, и не будет обнару-  
живать одноименный модуль в стандартной библиотеке (или в любом другом   
месте):  
# test\pkg\spam.py  
from . import string # <== Будет терпеть неудачу в случае отсутствия   
 # string.py в каталоге пакета!  
   
C:\test> C:\python30\python  
>>> import pkg.spam  
...текст сообщения опущен...  
ImportError: cannot import name string  
Модули, которые упоминаются в инструкциях импорта относительно текуще-  
го пакета, должны существовать в каталоге пакета.

Импортирование относительно пакета   
661  
Импорт по-прежнему выполняется   
относительно текущего рабочего каталога (еще раз)   
Инструкции импортирования по абсолютному пути позволяют пропускать   
модули пакета, однако они по-прежнему зависят от других элементов списка   
sys.path. В последнем нашем примере попробуем определить два собственных   
модуля string. Один модуль с этим именем находится в текущем рабочем ката-  
логе, другой – в пакете и еще один – в стандартной библиотеке:  
# test\string.py  
print(‘string’ \* 8)  
   
# test\pkg\spam.py  
from . import string # <== Относительная в обеих версиях, 2.6 и 3.0  
print(string)  
   
# test\pkg\string.py  
print(‘Ni’ \* 8)  
Когда мы импортируем модуль string с применением синтаксиса импортирова-  
ния относительно пакета, мы получаем версию модуля из пакета, как и следо-  
вало ожидать:  
C:\test> c:\Python30\python # Тот же результат получается в 2.6  
>>> import pkg.spam  
NiNiNiNiNiNiNiNi  
<module ‘pkg.string’ from ‘pkg\string.py’>  
Однако, когда используется синтаксис импортирования по абсолютному пути,   
версия модуля зависит от версии интерпретатора. Python 2.6 интерпретирует   
эту инструкцию, как операцию импортирования относительно пакета, но Py-  
Py-  
thon 3.0 рассматривает ее, как «абсолютную», то есть просто пропускает ката-  
 3.0 рассматривает ее, как «абсолютную», то есть просто пропускает ката-  
лог пакета и импортирует модуль из текущего рабочего каталога (не из стан-  
дартной библиотеки):  
# test\string.py  
print(‘string’ \* 8)  
   
# test\pkg\spam.py  
import string # <== Относительная в 2.6, “абсолютная” в 3.0:   
print(string) # текущий рабочий каталог!  
   
# test\pkg\string.py  
print(‘Ni’ \* 8)  
   
C:\test> c:\Python30\python  
>>> import pkg.spam  
stringstringstringstringstringstringstringstring  
<module ‘string’ from ‘string.py’>  
   
C:\test> c:\Python26\python  
>>> import pkg.spam  
NiNiNiNiNiNiNiNi  
<module ‘pkg.string’ from ‘pkg\string.pyc’>  
Как видите, хотя внутри пакета можно явно запросить модуль из этого же па-  
кета, тем не менее за его пределами инструкция импортирования все равно   
остается относительной по отношению к остальным обычным модулям в пути

662   
Глава 23. Пакеты модулей   
поиска. В данном случае для файла программы, использующей пакет, модуль   
в стандартной библиотеке оказывается недоступным. Все, что в действитель-  
ности было достигнуто изменениями в Python 3.0, – это лишь возможность   
выбирать, откуда импортировать модуль, – из пакета или из-за его пределов   
(то есть выполнять импортирование относительно текущего пакета или по аб-  
солютному пути). Поскольку разрешающая способность операции импортиро-  
вания зависит от окружения, которое невозможно предсказать заранее, опе-  
рация импортирования по абсолютному пути в версии 3.0 не гарантирует, что   
модуль будет найден в стандартной библиотеке.  
Поэкспериментируйте с этими примерами самостоятельно, чтобы глубже   
вникнуть в их смысл. На практике иногда имеется возможность структури-  
ровать инструкции импортирования, пути поиска и подбирать имена модулей   
так, что все будет работать именно так, как задумывалось при разработке. Но   
имейте в виду, что операции импортирования в крупных системах могут в зна-  
чительной степени зависеть от окружения, а определение правил импортиро-  
вания является неотъемлемой частью архитектуры успешной библиотеки.  
Теперь, когда вы познакомились с операциями импортирования   
относительно пакетов, запомните также, что их использование   
не всегда представляет собой лучший выбор. Импортирование   
пакетов по абсолютному пути, относительно каталогов в sys.  
path, иногда оказывается предпочтительнее, чем неявные опера-  
ции импортирования относительно пакета в Python 2 и явные   
операции импортирования относительно пакета в обеих версиях   
Python, 2 и 3.  
Синтаксис импортирования относительно пакетов и новые пра-  
вила поиска в операции импортирования по абсолютному пути   
в Python 3.0 требуют, как минимум, чтобы импортирование от-  
Python 3.0 требуют, как минимум, чтобы импортирование от-  
 3.0 требуют, как минимум, чтобы импортирование от-  
носительно текущего пакета выполнялось явно, что облегчает   
понимание и сопровождение. Однако файлы, в которых исполь-  
зуется синтаксис инструкций импортирования с точками, не-  
явно оказываются привязанными к каталогу пакета и не могут   
использоваться где-либо в другом месте без внесения изменений   
в программный код.  
Естественно, степень влияния этого обстоятельства на ваши мо-  
дули может отличаться от пакета к пакету – инструкции импор-  
тирования по абсолютному пути также могут потребовать вне-  
сения изменений в случае реорганизации структуры каталогов.  
Придется держать в уме: пакеты модулей  
Теперь, когда пакеты стали стандартной частью Python, часто мож-  
но встретить крупные расширения сторонних разработчиков, рас-  
пространяемые не как плоский список модулей, а как набор катало-  
гов с пакетами. Например, пакет расширений win32all для Python   
в операционной системе �indows был одним из первых, кто перешел   
на сторону победителя.

В заключение   
663  
Многие вспомогательные модули этого пакета располагаются в паке-  
тах, импортируемых посредством указания пути. Например, чтобы   
загрузить набор инструментальных средств для работы с технологи-  
ей COM на стороне клиента, можно использовать такую инструкцию:  
from win32com.client import constants, Dispatch  
Эта инструкция извлекает имена из модуля client в пакете win32com   
(подкаталог, куда был установлен пакет).  
Импортирование пакетов повсеместно используется в программном   
коде, работающем под управлением Jython, – реализации языка   
Python на Java, потому что библиотеки самого языка Java тоже ор-  
ганизованы в виде иерархии каталогов. В последних версиях Python   
инструменты для работы с электронной почтой и XML в стандартной   
библиотеке также были организованы в подкаталоги пакетов, а в   
Python 3.0 еще большее число родственных модулей было перемеще-  
но в пакеты (включая инструменты создания графического интер-  
фейса tkinter, инструменты организации сетевых взаимодействий по   
протоколу HTTP и многие другие). Следующие инструкции обеспе-  
чивают доступ к различным инструментам стандартной библиотеки   
в Python 3.0:  
from email.message import Message  
from tkinter.filedialog import askopenfilename  
from http.server import CGIHTTPRequestHandler  
Независимо от того, создаете вы каталоги пакетов или нет, в конеч-  
ном итоге у вас наверняка будет возможность импортировать их.  
В заключение  
В этой главе была представлена модель импортирования пакетов – необяза-  
тельный, но удобный способ явно указать путь к каталогам с модулями. В ин-  
струкциях импорта указывается путь относительно каталога, находящегося   
в пути поиска модулей, но вместо того, чтобы полагаться на результаты поиска,   
выполняемого интерпретатором, ваши сценарии могут явно указывать остаток   
пути к модулю.  
Как мы видели, пакеты не только делают операцию импортирования более   
осмысленной в крупных программах, но еще и упрощают настройку пути по-  
иска (если все каталоги, откуда производится импорт, вложены в один общий   
корневой каталог), а также позволяют разрешать возникающие неоднознач-  
ности в тех случаях, когда существует более одного модуля с одним и тем же   
именем (наличие имен каталогов в операциях импортирования позволяет обе-  
спечить уникальность имен модулей).  
Поскольку это имеет отношение к программированию пакетов, здесь мы так-  
же исследовали новейшую модель импортирования относительно пакетов –   
способ импортирования в файлах пакета, позволяющий с помощью начальных   
точек в инструкции from явно указывать, что модуль должен импортироваться

664   
Глава 23. Пакеты модулей   
из того же пакета, – вместо того, чтобы полагаться на устаревшие неявные пра-  
вила поиска модулей в пакетах.  
В следующей главе мы исследуем несколько более сложных тем, имеющих от-  
ношение к модулям, таких как синтаксис относительного импорта и перемен-  
ная режима использования \_\_name\_\_. Как обычно, мы заканчиваем эту главу   
серией контрольных вопросов, чтобы проверить, насколько хорошо вы усвои-  
ли сведения, полученные здесь.  
Закрепление пройденного  
Контрольные вопросы  
1. Для чего служит файл \_\_init\_\_.py в каталогах пакетов модулей?  
2. Как избежать необходимости снова и снова вводить полное имя пакета при   
каждом обращении к содержимому пакетов?  
3. В каких каталогах необходимо создавать файл \_\_init\_.py?  
4. В каких случаях вместо инструкции from приходится использовать ин-  
струкцию import?  
5. Чем отличаются инструкции from mypkg import spam и from . import spam?  
Ответы  
1. Файлы \_\_init\_\_.py служат для объявления и инициализации пакета, – ин-  
терпретатор автоматически запускает программный код в этих файлах,   
когда каталог импортируется программой впервые. Переменные, которым   
выполняется присваивание в этих файлах, становятся атрибутами объекта   
модуля для соответствующего каталога. Присутствие этих файлов в ката-  
логах пакетов обязательно – вы не сможете импортировать пакеты при от-  
сутствии этих файлов в каталогах.  
2. Используйте инструкцию from, чтобы скопировать имена из пакета или   
воспользуйтесь расширением as инструкции import, чтобы переименовать   
полный путь в короткий синоним. В обоих случаях полный путь будет при-  
сутствовать только в одном месте – в инструкции from или import.  
3. Каждый каталог, перечисленный в инструкции import или from, должен со-  
держать файл \_\_init\_\_.py. Другие каталоги, включая каталог, содержащий   
самый первый компонент пути к пакету, не требуют наличия этого файла.  
4. Инструкция import должна использоваться вместо инструкции from, только   
если вам необходимо обеспечить доступ к одному и тому же имени более чем   
в одном каталоге. Благодаря инструкции import употребление полного пути   
обеспечивает уникальность ссылок, тогда как инструкция from допускает   
наличие только одной версии любого имени.  
5. Инструкция from mypkg import spam выполняет импорт по абсолютному   
пути – при поиске mypkg она пропускает каталог пакета и пользуется спи-  
ском каталогов в sys.path. Инструкция from . import spam, напротив, выпол-  
няет импорт относительно текущего пакета – поиск модуля spam выпол-  
няется относительно пакета, внутри которого находится эта инструкция.

Глава 24.  
   
Дополнительные возможности   
модулей  
Эта глава завершает пятую часть книги, представляя коллекцию более слож-  
ных тем, имеющих отношение к модулям, – сокрытие данных, модуль \_\_fu-  
fu-  
ture\_\_, использование переменной \_\_name\_\_, способы изменения списка sys.  
path, инструменты интроспекции, запуск модуля по имени в виде строки, тран-  
зитивная перезагрузка модулей и так далее – и, кроме того, содержит перечень   
наиболее типичных ошибок и упражнения, завершающие эту часть книги.   
Попутно мы создадим несколько более крупных и полезных инструментов,   
чем те, что мы видели до сих пор, объединяющих в себе функции и модули.   
Как и функции, модули наиболее эффективны, когда они имеют хорошо про-  
думанные интерфейсы, поэтому в данной главе будет представлен краткий   
обзор концепций проектирования модулей, часть из которых мы исследовали   
в предыдущих главах.   
Несмотря на слово «дополнительные» в названии этой главы, некоторые из   
обсуждаемых здесь возможностей (такие, как приемы работы с переменной   
\_\_name\_\_) используются достаточно широко, поэтому обязательно ознакомь-  
тесь с ними, прежде чем переходить к классам в следующей части книги.  
Сокрытие данных в модулях  
Как мы уже видели, модули в языке Python экспортируют все имена, которым   
были присвоены значения на верхнем уровне файлов. В языке нет никаких   
объявлений, которые позволили бы сделать одни имена видимыми, а другие –   
невидимыми за пределами модуля. Фактически нет никакого способа предот-  
вратить возможность изменения имен в модуле извне, если у кого-то появится   
такое желание.  
Сокрытие данных модуля в языке Python регулируется соглашениями, а не   
синтаксическими конструкциями. Если задаться целью повредить модуль,   
изменяя имена в нем, вам ничто не сможет помешать, но, к счастью, я еще   
не встречал программистов, кто стремился бы это сделать. Некоторые пури-

666   
Глава 24. Дополнительные возможности модулей   
сты возражают против такого либерального отношения к сокрытию данных,   
утверждая в связи с этим, что в языке Python отсутствует возможность инкап-  
Python отсутствует возможность инкап-  
 отсутствует возможность инкап-  
суляции. Однако инкапсуляция в языке Python имеется, просто она, скорее,   
относится к организации пакетов, чем к возможности накладывать ограни-  
чения.  
Минимизация повреждений, причиняемых   
инструкцией from \*: \_X и \_\_all\_\_  
Как особый случай, существует возможность начинать имена переменных   
с одного символа подчеркивания (например, \_X), чтобы предотвратить их пере-  
записывание, когда клиент выполняет импортирование модуля инструкцией   
from \*. Этот прием на самом деле предназначен только для минимизации за-  
грязнения пространства имен – так как инструкция from \* копирует все имена,   
импортирующий модуль может получить больше, чем предполагал (включая   
имена, которые перезапишут имена импортирующего модуля). Символы под-  
черкивания не являются объявлением «частных» данных: вы по-прежнему   
можете видеть эти имена и изменять их с помощью других форм импортирова-  
ния, таких как инструкция import.  
Альтернативный способ достижения эффекта сокрытия данных, напоминаю-  
щий соглашение об именовании \_X, заключается в присвоении на верхнем уров-  
не модуля переменной \_\_all\_\_ списка строк с именами переменных. Например:  
\_\_all\_\_ = [“Error”, “encode”, “decode”] # Экспортируются только эти имена  
При использовании этого приема инструкция from \* будет копировать только   
имена, перечисленные в списке \_\_all\_\_. В действительности это соглашение,   
обратное соглашению \_X: переменная \_\_all\_\_ идентифицирует имена, доступ-  
ные для копирования, тогда как соглашение \_X идентифицирует имена, недо-  
ступные для копирования. Интерпретатор Python сначала отыскивает список   
\_\_all\_\_ в модуле, и если он отсутствует, инструкция from \* копирует все имена,   
которые не начинаются с единственного символа подчеркивания.  
Подобно соглашению \_X, список \_\_all\_\_ имеет смысл только для инструкции   
from \* и не является объявлением частных данных. Программисты могут ис-  
пользовать при реализации модулей любой из этих приемов, которые хорошо   
работают с инструкцией from \*. (Смотрите также обсуждение списков \_\_all\_\_   
в файлах пакетов \_\_init\_\_.py в главе 23 – там эти списки объявляют субмоду-  
ли, которые могут быть загружены инструкцией from \*.)  
Включение будущих возможностей языка  
В языке периодически появляются изменения, которые могут повлиять на ра-  
ботоспособность существующего программного кода. Сначала они появляются   
в виде расширений, которые по умолчанию отключены. Чтобы включить та-  
кие расширения, используется инструкция импорта специального вида:  
from \_\_future\_\_ import имя\_функциональной\_особенности  
Эта инструкция вообще должна появляться в самом начале файла модуля (воз-  
можно, сразу же вслед за строкой документирования), потому что она включа-  
ет специальный режим компиляции программного кода для каждого отдельно

Смешанные режимы использования: \_\_name\_\_ и \_\_main\_\_   
667  
взятого модуля. Возможно также выполнить эту инструкцию в интерактив-  
ной оболочке, что позволит поэкспериментировать с грядущими изменениями   
в языке – включенная особенность будет после этого доступна в течение всего   
интерактивного сеанса.  
Например, в предыдущих изданиях этой книги мы использовали эту форму   
инструкции для демонстрации функций-генераторов, в которых использует-  
ся ключевое слово, еще недоступное по умолчанию в то время (в качестве имя\_  
функциональной\_особенности указывалось имя generators). Мы уже использовали   
эту инструкцию для включения операции истинного деления чисел в главе 5,   
функции print в главе 11 и импорта в пакетах по абсолютному пути в главе 23.  
Все эти изменения могут отрицательно сказаться на работоспособности суще-  
ствующего программного кода для Python 2.6, и поэтому они так постепенно   
вводятся в язык – сначала в виде дополнительных возможностей, включаемых   
с помощью этой специальной формы инструкции импорта.  
Смешанные режимы использования:   
\_\_name\_\_ и \_\_main\_\_  
Ниже демонстрируется специальный прием, позволяющий импортировать   
файлы как модули и запускать их как самостоятельные программы. Каждый   
модуль обладает встроенным атрибутом \_\_name\_\_, который устанавливается   
интерпретатором следующим образом:  
 •  
Если файл запускается как главный файл программы, атрибуту \_\_name\_\_ на   
запуске присваивается значение “\_\_main\_\_”.  
 •  
Если файл импортируется, атрибуту \_\_name\_\_ присваивается имя модуля,   
под которым он будет известен клиенту.  
Благодаря этому модуль может проверить собственный атрибут \_\_name\_\_   
и определить, был ли он запущен как самостоятельная программа или импор-  
тирован другим модулем. Например, предположим, что мы создаем файл моду-  
ля с именем runme.py, который экспортирует единственную функцию с именем   
tester:  
def tester():  
 print(“It’s Christmas in Heaven...”)  
   
if \_\_name\_\_ == ‘\_\_main\_\_’: # Только когда запускается,  
 tester() # а не импортируется  
Этот модуль определяет функцию для клиентов и может импортироваться как   
обычный модуль:  
% python  
>>> import runme  
>>> runme.tester()  
It’s Christmas in Heaven...  
Но в самом конце модуля имеется программный код, который вызывает функ-  
цию, когда этот файл запускается как самостоятельная программа:  
% python runme.py  
It’s Christmas in Heaven...

668   
Глава 24. Дополнительные возможности модулей   
Таким образом, переменная \_\_name\_\_ может играть роль флага, определяющего   
режим использования, позволяя программному коду выполнять разные дей-  
ствия, когда он используется как импортируемая библиотека или как само-  
стоятельный сценарий. Вы будете встречать этот прием практически во всех   
действующих программах на языке Python, с которыми вам предстоит стол-  
Python, с которыми вам предстоит стол-  
, с которыми вам предстоит стол-  
кнуться.  
Пожалуй, чаще всего проверка атрибута \_\_name\_\_ выполняется в программном   
коде для самопроверки модуля. Проще говоря, вы можете добавить в конец мо-  
дуля программный код, который будет выполнять проверку экспортируемых   
элементов внутри самого модуля, заключив этот код в условную инструкцию,   
проверяющую атрибут \_\_name\_\_. При таком подходе вы можете использовать   
файл в других модулях, импортируя его, и тестировать логику работы, запу-  
ская его из командной строки или каким-либо другим способом. На практике   
программный код самопроверки в конце файла, заключенный в условную ин-  
струкцию, проверяющую атрибут \_\_name\_\_, является, пожалуй, самым распро-  
страненным и удобным способом модульного тестирования в языке Python. (В   
главе 35 обсуждаются другие часто используемые способы тестирования про-  
граммного кода на языке Python – как будет показано, в стандартной библио-  
теке существуют модули unittest и doctest, которые реализуют более совершен-  
ные средства тестирования.)  
Прием, основанный на проверке атрибута \_\_name\_\_, также часто используется   
при создании файлов, которые могут использоваться и как утилиты команд-  
ной строки, и как библиотеки инструментов. Например, предположим, что   
вы пишете на языке Python сценарий поиска файлов. Код принесет больше   
пользы, если реализовать его в виде функций и добавить проверку атрибута   
\_\_name\_\_ для организации вызова этих функций, когда файл запускается как   
самостоятельная программа. При таком подходе сценарий может повторно ис-  
пользоваться в составе других программ.  
Тестирование модулей с помощью \_\_name\_\_  
Мы уже видели в этой книге один хороший пример, когда проверка атрибута   
\_\_name\_\_ могла бы быть полезной. В разделе, рассказывающем об аргументах,   
в главе 18, мы написали сценарий, который находит минимальное значение   
среди множества предложенных аргументов:  
def minmax(test, \*args):  
 res = args[0]  
 for arg in args[1:]:  
 if test(arg, res):  
 res = arg  
 return res  
   
 def lessthan(x, y): return x < y  
 def grtrthan(x, y): return x > y  
   
 print(minmax(lessthan, 4, 2, 1, 5, 6, 3)) # Код самопроверки  
 print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))  
В самом конце этого сценария присутствует программный код самопроверки,   
благодаря которому мы можем проверить правильность работы модуля без   
необходимости вводить его в интерактивной оболочке всякий раз, когда нам   
потребуется проверить модуль. Однако при такой реализации имеется одна

Смешанные режимы использования: \_\_name\_\_ и \_\_main\_\_   
669  
проблема – результаты самопроверки будут выводиться на экран всякий раз,   
когда этот файл будет импортироваться для использования другим файлом, но   
тогда это становится невежливым по отношению к пользователю! Чтобы испра-  
вить положение, можно обернуть проверочные вызовы функции в условную   
инструкцию, проверяющую атрибут \_\_name\_\_ так, чтобы они выполнялись,   
только когда файл запускается как самостоятельная программа, а не во время   
импорта:  
print ‘I am:’, \_\_name\_\_  
   
def minmax(test, \*args):  
 res = args[0]  
 for arg in args[1:]:  
 if test(arg, res):  
 res = arg  
 return res  
   
def lessthan(x, y): return x < y  
def grtrthan(x, y): return x > y  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 print(minmax(lessthan, 4, 2, 1, 5, 6, 3)) # Код самопроверки  
 print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))  
Тут в самом начале добавлена инструкция вывода значения атрибута \_\_name\_\_,   
чтобы проверить его визуально. Интерпретатор Python создает эту переменную   
и присваивает ей значение во время загрузки файла. Когда файл запускает-  
ся как самостоятельная программа, этому имени присваивается значение ‘\_\_  
main\_\_’, поэтому в данном случае происходит автоматическое выполнение кода   
самопроверки:  
% python min.py  
I am: \_\_main\_\_  
1  
6  
Однако, когда файл импортируется, значение атрибута \_\_name\_\_ уже не равно   
‘\_\_main\_\_’, поэтому необходимо явно вызвать функцию, чтобы запустить ее:  
>>> import min  
I am: min  
>>> min.minmax(min.lessthan, ‚s‘, ‚p‘, ‚a‘, ‚m‘)  
‚a‘  
Неважно, будет ли использоваться этот прием для нужд тестирования, глав-  
ный результат – что наш программный код может использоваться и как би-  
блиотека инструментов, и как самостоятельная программа.  
Обработка аргументов командной строки   
с помощью \_\_name\_\_  
Теперь представим более практичный пример, демонстрирующий еще один   
распространенный способ использования переменной \_\_name\_\_. Следующий   
модуль, formats.py, определяет вспомогательные функции форматирования   
строк. Кроме того, он проверяет имя модуля, чтобы узнать, был ли он запущен   
как самостоятельная программа. Если это так, он проверяет и использует ар-

670   
Глава 24. Дополнительные возможности модулей   
гументы командной строки для запуска встроенного или конкретного теста.   
Список sys.argv в языке Python содержит аргументы командной строки – спи-  
сок строк со словами, введенными в командной строке, где первый элемент   
списка всегда содержит имя файла сценария:  
“””  
Различные специализированные функции форматирования строк.  
Модуль можно протестировать с помощью встроенных тестов или посредством передачи   
аргументов командной строки.  
“””  
   
def commas(N):  
 “””  
 Форматирует целое положительное число N, добавляя запятые,  
 разделяющие группы разрядов: xxx,yyy,zzz  
 “””  
 digits = str(N)  
 assert(digits.isdigit())  
 result = ‘’  
 while digits:  
 digits, last3 = digits[:-3], digits[-3:]  
 result = (last3 + ‘,’ + result) if result else last3  
 return result  
   
def money(N, width=0):  
 “””  
 Форматирует число N, добавляя запятые, оставляя 2 десятичных знака   
 в дробной части, добавляя в начало символ $ и знак числа, и,   
 при необходимости, – отступ: $ -xxx,yyy.zz  
 “””  
 sign = ‘-’ if N < 0 else ‘’  
 N = abs(N)  
 whole = commas(int(N))  
 fract = (‘%.2f’ % N)[-2:]  
 format = ‘%s%s.%s’ % (sign, whole, fract)  
 return ‘$%\*s’ % (width, format)  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 def selftest():  
 tests = 0, 1 # ошибка при значениях: -1, 1.23  
 tests += 12, 123, 1234, 12345, 123456, 1234567  
 tests += 2 \*\* 32, 2 \*\* 100  
 for test in tests:  
 print(commas(test))  
   
 print(‘’)  
 tests = 0, 1, -1, 1.23, 1., 1.2, 3.14159  
 tests += 12.34, 12.344, 12.345, 12.346  
 tests += 2 \*\* 32, (2 \*\* 32 + .2345)  
 tests += 1.2345, 1.2, 0.2345  
 tests += -1.2345, -1.2, -0.2345  
 tests += -(2 \*\* 32), -(2\*\*32 + .2345)  
 tests += (2 \*\* 100), -(2 \*\* 100)  
 for test in tests:  
 print(‘%s [%s]’ % (money(test, 17), test))  
   
 import sys  
 if len(sys.argv) == 1:

Смешанные режимы использования: \_\_name\_\_ и \_\_main\_\_   
671  
 selftest()  
 else:  
 print(money(float(sys.argv[1]), int(sys.argv[2])))  
Этот модуль работает одинаково в версиях Python 2.6 и 3.0. Когда он запуска-  
Python 2.6 и 3.0. Когда он запуска-  
 2.6 и 3.0. Когда он запуска-  
ется как самостоятельная программа, он тестирует себя, как было описано   
выше, но при этом для управления тестированием он использует аргументы   
командной строки. Чтобы увидеть, что будет выведено в результате тестирова-  
ния, запустите у себя этот файл как самостоятельную программу. Чтобы про-  
тестировать определенные строки, передайте их в командной строке вместе со   
значением минимальной ширины поля вывода:  
C:\misc> python formats.py 999999999 0  
$999,999,999.00  
   
C:\misc> python formats.py -999999999 0  
$-999,999,999.00  
   
C:\misc> python formats.py 123456789012345 0  
$123,456,789,012,345.00  
   
C:\misc> python formats.py -123456789012345 25  
$ -123,456,789,012,345.00  
   
C:\misc> python formats.py 123.456 0  
$123.46  
   
C:\misc> python formats.py -123.454 0  
$-123.45  
   
C:\misc> python formats.py  
...чтобы увидеть результаты встроенных тестов – запустите сценарий сами...  
Этот модуль может запускаться как самостоятельная программа и импорти-  
роваться другими модулями, поэтому мы можем импортировать его функции   
как обычные библиотечные компоненты:  
>>> from formats import money, commas  
>>> money(123.456)  
‘$123.46’  
>>> money(-9999999.99, 15)  
‘$ -9,999,999.99’  
>>> X = 99999999999999999999  
>>> ‘%s (%s)’ % (commas(X), X)  
‘99,999,999,999,999,999,999 (99999999999999999999)’  
В этом файле также присутствуют строки документирования, с которыми мы   
познакомились в главе 15, поэтому мы можем использовать функцию help для   
исследования инструментов, входящих в него:  
>>> import formats  
>>> help(formats)  
Help on module formats:  
NAME  
 formats  
   
FILE  
 c:\misc\formats.py

672   
Глава 24. Дополнительные возможности модулей   
DESCRIPTION  
Различные специализированные функции форматирования строк.  
Модуль можно протестировать с помощью встроенных тестов или посредством передачи   
аргументов командной строки.  
   
FUNCTIONS  
 commas(N)  
 Форматирует целое положительное число N, добавляя запятые,  
 разделяющие группы разрядов: xxx,yyy,zzz  
   
 money(N, width=0)  
 Форматирует число N, добавляя запятые, оставляя 2 десятичных знака   
 в дробной части, добавляя в начало символ $ и знак числа, и,   
 при необходимости, отступ: $ -xxx,yyy.zz  
Вы можете использовать аргументы командной строки похожими способами,   
чтобы обеспечить передачу в свои сценарии входных данных, которые также   
могут быть встроены в программный код как функции или классы и доступ-  
ны импортирующим модулям. Если вам потребуются более широкие возмож-  
ности обработки аргументов командной строки, обратите внимание на модули   
getopt и optparse – они входят в состав стандартной библиотеки Python и опи-  
Python и опи-  
 и опи-  
сываются в руководствах. В некоторых случаях можно также использовать   
встроенную функцию input, представленную в главе 3, которой мы пользова-  
лись в главе 10, чтобы реализовать возможность запрашивать данные у поль-  
зователя, вместо того чтобы заставлять его вводить эти данные в командной   
строке.  
Загляните также в главу 7, где обсуждается новый синтаксис   
спецификаторов формата {,d}, который будет доступен в версии   
Python 3.1 и выше, – он позволяет вставлять запятые между   
группами разрядов, подобно функциям в этом модуле. Кроме   
того, модуль, представленный здесь, позволяет также формати-  
ровать денежные суммы и может служить альтернативным спо-  
собом добавления запятых для тех, кто пользуется версиями   
Python ниже 3.1.  
Изменение пути поиска модулей  
В главе 21 мы узнали, что путь поиска модулей – это список каталогов, и что   
этот список можно дополнить с помощью переменной окружения PYTHONPATH   
и файлов .pth. Но я пока еще не показывал, как сами программы на языке   
Python могут изменять путь поиска, изменяя встроенный список с именем sys.  
path (атрибут path встроенного модуля sys). Список sys.path инициализируется   
во время запуска программы, однако и после этого допускается удалять, до-  
бавлять и изменять компоненты списка по своему усмотрению:  
>>> import sys  
>>> sys.path  
[‘’, ‘C:\\users’, ‘C:\\Windows\\system32\\python30.zip’, ...далее опущено...]  
   
>>> sys.path.append(‘C:\\sourcedir’) # Дополнение пути поиска модулей  
>>> import string # Новый каталог будет участвовать  
 # в поиске

Расширение as для инструкций import и from   
673  
Как только будут внесены изменения, они будут воздействовать на все последу-  
ющие инструкции импорта, выполняемые в программе, так как все инструк-  
ции во всех файлах программы используют один и тот же общий список sys.  
path. Этот список может изменяться произвольным образом:  
>>> sys.path = [r’d:\temp’] # Изменяет путь поиска модулей  
>>> sys.path.append(‘c:\\lp4e\\examples’) # Только для этой программы  
>>> sys.path  
[‘d:\\temp’, ‘c:\\lp4e\\examples’]  
   
>>> import string  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in ?  
ImportError: No module named string  
Таким образом, этот прием может использоваться для динамической настрой-  
ки пути поиска внутри программ на языке Python. Однако будьте вниматель-  
ны: если убрать из пути критически важный каталог, можно потерять доступ   
к критически важным утилитам. Так в предыдущем примере был потерян   
доступ к модулю string, потому что из пути был удален исходный каталог би-  
блиотеки.  
Кроме того, не забывайте, что такие изменения списка sys.path действуют   
только в рамках интерактивного сеанса или внутри программы (технически –   
в рамках процесса), где были выполнены эти изменения, – они не сохраняются   
после завершения работы интерпретатора. Настройки в переменной окруже-  
ния PYTHONPATH и в файлах .pth располагаются в самой операционной системе,   
а не в работающей программе, и потому они имеют более глобальный характер:   
они воспринимаются всеми программами, которые запускаются на вашей ма-  
шине, и продолжают существовать по завершении программы.  
Расширение as для инструкций import и from  
Обе инструкции, import и from, были расширены так, чтобы позволить давать   
модулям в вашем сценарии другие имена. Следующая инструкция import:  
import longmodulename as name  
эквивалентна инструкциям:  
import longmodulename  
name = longmodulename  
del longmodulename # Не сохранять оригинальное имя  
После выполнения такой инструкции import для ссылки на модуль можно (и   
фактически необходимо) использовать имя, указанное после ключевого слова   
as. Точно такое же расширение имеется и у инструкции from, где оно позволяет   
изменять имена, импортируемые из файла:  
from module import longname as name  
Это расширение обычно используется с целью создать короткие синонимы для   
длинных имен и избежать конфликтов с именами, уже используемыми в сце-  
нарии, которые в противном случае были бы просто перезаписаны инструкци-  
ей импортирования:

674   
Глава 24. Дополнительные возможности модулей   
import reallylongmodulename as name # Использовать короткий псевдоним  
name.func()  
   
from module1 import utility as util1 # Допускается только одно имя “utility”  
from module2 import utility as util2  
util1(); util2()  
Кроме того, это расширение может пригодиться с целью создания коротких   
и простых имен для длинных путей, состоящих из цепочки каталогов, при им-  
портировании пакетов, которое описывалось в главе 23:  
import dir1.dir2.mod as mod # Полный путь достаточно указать всего один раз  
mod.func()  
Модули – это объекты: метапрограммы  
Поскольку модули экспортируют большую часть своих свойств в виде встро-  
енных атрибутов, это позволяет легко создавать программы, которые управ-  
ляют другими программами. Такие менеджеры программ мы обычно назы-  
ваем метапрограммами, потому что они работают поверх других программ.   
Этот прием также называется интроспекцией, потому что программы могут   
просматривать внутреннее устройство объектов и действовать исходя из этого.   
Интроспекция – это дополнительная особенность, которая может быть полезна   
при создании инструментальных средств программирования.  
Например, чтобы получить значение атрибута с именем name в модуле с именем   
M, мы можем использовать полное имя атрибута или обратиться к нему с помо-  
щью словаря атрибутов модуля (экспортируется в виде встроенного атрибута   
\_\_dict\_\_, с которым мы уже встречались в главе 22). Кроме того, интерпрета-  
тор экспортирует список всех загруженных модулей в виде словаря sys.modules   
(то есть в виде атрибута modules модуля sys) и предоставляет встроенную функ-  
цию getattr, которая позволяет получать доступ к атрибутам по строкам с их   
именами (напоминает выражение object.attr, только attr – это строка времени   
выполнения). Благодаря этому все следующие выражения представляют один   
и тот же атрибут и объект:  
M.name # Полное имя объекта  
M.\_\_dict\_\_[‘name’] # Доступ с использованием словаря пространства имен  
sys.modules[‘M’].name # Доступ через таблицу загруженных модулей  
getattr(M, ‘name’) # Доступ с помощью встроенной функции  
Обеспечивая доступ к внутреннему устройству модулей, интерпретатор по-  
могает создавать программы, управляющие другими программами.1 Напри-  
мер, ниже приводится модуль с именем mydir.py, в котором использованы эти   
1   
Как мы видели в главе 17, функция может получить доступ к вмещающему модулю   
с помощью таблицы sys.modules, что позволяет имитировать действие инструкции   
global. Например, эффект действия инструкций global X; X=0 внутри функции можно   
реализовать (хотя для этого придется ввести с клавиатуры значительно больше сим-  
волов!) так: import sys; glob=sys.modules[\_\_name\_\_]; glob.X=0. Не забывайте, что каж-  
дый модуль имеет атрибут \_\_name\_\_� внутри функции, принадлежащей модулю, он   
выглядит как глобальное имя. Этот прием обеспечивает еще один способ изменения   
одноименных локальных и глобальных переменных внутри функции.

Модули – это объекты: метапрограммы   
675  
идеи для реализации измененной версии встроенной функции dir. Этот модуль   
определяет и экспортирует функцию с именем listing, которая принимает объ-  
ект модуля в качестве аргумента и выводит отформатированный листинг про-  
странства имен модуля:  
“””  
mydir.py: выводит содержимое пространства имен других модулей  
“””  
   
seplen = 60  
sepchr = ‘-’  
   
def listing(module, verbose=True):  
 sepline = sepchr \* seplen  
 if verbose:  
 print(sepline)  
 print(‘name:’, module.\_\_name\_\_, ‘file:’, module.\_\_file\_\_)  
 print(sepline)  
   
 count = 0  
 for attr in module.\_\_dict\_\_: # Сканировать пространство имен  
 print(“%02d) %s” % (count, attr), end=’ ‘)  
 if attr.startswith(‘\_\_’):  
 print(“<built-in name>”) # Пропустить \_\_file\_\_ и др.  
 else:  
 print(getattr(module, attr)) # То же, что и .\_\_dict\_\_[attr]  
 count = count+1  
   
 if verbose:  
 print(sepline)  
 print(module.\_\_name\_\_, ‘has %d names’ % count)  
 print(sepline)  
   
 if \_\_name\_\_ == “\_\_main\_\_”:  
 import mydir  
 listing(mydir) # Код самопроверки: вывести свое   
 # пространство имен  
Обратите внимание на строку документирования в начале модуля – как и в   
предыдущем примере formats.py, мы можем получить доступ к ней с помощью   
универсальных инструментов. Строки документирования создаются с целью   
предоставить функциональное описание, которое можно получить через атри-  
бут \_\_doc\_\_ или с помощью функции help (подробности приводятся в главе 15):  
>>> import mydir  
>>> help(mydir)  
Help on module mydir:  
   
NAME  
 mydir - mydir.py: выводит содержимое пространства имен других модулей  
   
FILE  
 c:�sers\veramark\mark\mydir.py  
   
FUNCTIONS  
 listing(module, verbose=True)

676   
Глава 24. Дополнительные возможности модулей   
DATA  
 sepchr = ‘-’  
 seplen = 60  
В модуле, в самом конце, реализована логика самопроверки, которая заставля-  
ет модуль импортировать самого себя и вывести содержимое своего простран-  
ства имен. Ниже показан результат работы этого модуля под управлением   
Python 3.0 (чтобы использовать его в Python 2.6, включите возможность ис-  
пользования функции print как в версии 3.0, импортировав модуль \_\_future\_\_,   
как описывается в главе 11, потому что ключевое слово end допустимо только   
в версии 3.0):  
C:�sers\veramark\Mark> c:\Python30\python mydir.py  
------------------------------------------------------------  
name: mydir file: C:�sers\veramark\Mark\mydir.py  
------------------------------------------------------------  
00) seplen 60  
01) \_\_builtins\_\_ <built-in name>  
02) \_\_file\_\_ <built-in name>  
03) \_\_package\_\_ <built-in name>  
04) listing <function listing at 0x026D3B70>  
05) \_\_name\_\_ <built-in name>  
06) sepchr -  
07) \_\_doc\_\_ <built-in name>  
------------------------------------------------------------  
mydir has 8 names  
------------------------------------------------------------  
Чтобы использовать его как инструмент интроспекции других модулей, просто   
передайте функции listing объект требуемого модуля. Ниже приводится спи-  
сок атрибутов модуля tkinter из стандартной библиотеки (он же Tkinter в Py-  
Py-  
thon 2.6):  
>>> import mydir  
>>> import tkinter  
>>> mydir.listing(tkinter)  
------------------------------------------------------------  
name: tkinter file: c:\PYTHON30\lib\tkinter\\_\_init\_\_.py  
------------------------------------------------------------  
00) getdouble <class ‘float’>  
01) MULTIPLE multiple  
02) mainloop <function mainloop at 0x02913B70>  
03) Canvas <class ‘tkinter.Canvas’>  
04) AtSelLast <function AtSelLast at 0x028FA7C8>  
...many more name omitted...  
151) StringVar <class ‘tkinter.StringVar’>  
152) ARC arc  
153) At <function At at 0x028FA738>  
154) NSEW nsew  
155) SCROLL scroll  
------------------------------------------------------------  
tkinter has 156 names  
------------------------------------------------------------  
С функцией getattr и родственными ей мы встретимся еще раз позднее. Самое   
важное здесь, что mydir – это программа, которая позволяет исследовать другие

Импортирование модулей по имени в виде строки   
677  
программы. Так как интерпретатор не скрывает внутреннее устройство моду-  
лей, вы можете реализовать обработку любых объектов единообразно.1  
Импортирование модулей   
по имени в виде строки  
Имя модуля в инструкции import или from является именем переменной. Тем   
не менее иногда ваша программа будет получать имя модуля, который следует   
импортировать, в виде строки во время выполнения (например, в случае, когда   
пользователь выбирает имя модуля внутри графического интерфейса). К сожа-  
лению, невозможно напрямую использовать инструкции импорта для загруз-  
ки модуля, имя которого задано в виде строки, – в этих инструкциях интерпре-  
татор ожидает получить имя переменной, а не строку. Например:  
>>> import “string”  
 File “<stdin>”, line 1  
 import “string”  
 ^  
SyntaxError: invalid syntax  
Точно так же невозможно импортировать модуль, если просто присвоить стро-  
ку переменной:  
x = “string”  
import x  
Здесь интерпретатор попытается импортировать файл x.py, а не модуль string –   
имя в инструкции import превращается в имя переменной, которой присваива-  
ется объект загружаемого модуля, и идентифицирует внешний файл букваль-  
но.  
Чтобы решить эту проблему, необходим специальный инструмент, выполняю-  
щий динамически загрузку модулей, имена которых создаются в виде строк   
во время выполнения. Обычно для этого конструируется строка программного   
кода, содержащая инструкцию import, которая затем передается встроенной   
функции exec для исполнения (в Python 2.6 exec – это инструкция, но она мо-  
жет использоваться точно так же, как показано здесь, – круглые скобки просто   
игнорируются интерпретатором):  
>>> modname = “string”  
>>> exec(“import “ + modname) # Выполняется как строка программного кода  
>>> string # Модуль был импортирован в пространство имен  
<module ‘string’ from ‘c:\Python30\lib\string.py’>  
Функция exec (и родственная ей функция eval, используемая для вычисления   
значений выражений) скомпилирует строку в код и передаст его интерпрета-  
1   
Инструменты, такие как mydir.listing, могут быть предварительно загружены в про-  
странство имен интерактивной оболочки импортированием их в файле, указанном   
в переменной окружения PYTHONSTARTUP. Так как программный код этого файла выпол-  
няется в интерактивном пространстве имен (модуль \_\_main\_\_), такой способ импорти-  
рования часто используемых инструментов позволит вам сэкономить время на вводе   
инструкций вручную. Дополнительная информация приводится в приложении A.

678   
Глава 24. Дополнительные возможности модулей   
тору для исполнения. В языке Python компилятор байт-кода доступен непо-  
Python компилятор байт-кода доступен непо-  
 компилятор байт-кода доступен непо-  
средственно во время выполнения, поэтому можно писать программы, которые   
конструируют и выполняют другие программы, как в этом случае. По умол-  
чанию функция exec выполняет программный код в текущей области видимо-  
сти, но существует возможность передавать ей необязательные словари про-  
странств имен.  
Единственный настоящий недостаток функции exec состоит в том, что она   
должна компилировать инструкцию import всякий раз, когда она запускает-  
ся, – если импортировать приходится достаточно часто, программный код мо-  
жет работать немного быстрее при использовании встроенной функции \_\_im-  
im-  
port\_\_, которая выполняет загрузку модуля, получая его имя в виде строки.   
Результат получается тот же самый, но функция \_\_import\_\_ возвращает объект   
модуля, поэтому его надо присвоить переменной, чтобы сохранить:  
>>> modname = “string”  
>>> string = \_\_import\_\_(modname)  
>>> string  
<module ‘string’ from ‘c:\Python30\lib\string.py’>  
Транзитивная перезагрузка модулей   
В главе 22 мы изучали возможность повторной загрузки модулей как способ   
ввести в действие изменения в программном коде без остановки и перезапуска   
программы. Когда выполняется повторная загрузка модуля, интерпретатор   
перезагружает только данный конкретный файл модуля – он не выполняет по-  
вторную загрузку модулей, которые были импортированы перезагружаемым   
модулем.   
Например, если выполняется перезагрузка некоторого модуля A, и A импорти-  
рует модули B и C, перезагружен будет только модуль A, но не B и C. Инструк-  
ции внутри модуля A, которые импортируют модули B и C, будут перезапуще-  
ны в процессе перезагрузки, но они просто вернут объекты уже загруженных   
модулей B и C (предполагается, что к этому моменту они уже были импорти-  
рованы). Чтобы было более понятно, ниже приводится пример содержимого   
файла A.py:  
import B # Эти модули не будут перезагружены вместе с A   
import C # Просто будут импортированы уже загруженные модули  
   
% python  
>>> ...  
>>> from imp import reload  
>>> reload(A)  
Не следует полагаться на транзитивную перезагрузку модулей – лучше не-  
сколько раз вызывайте функцию reload для непосредственного обновления   
субкомпонентов. В крупных системах это может потребовать выполнить зна-  
чительный объем работы при тестировании в интерактивной оболочке. При   
желании можно предусмотреть в программе автоматическую перезагрузку ее   
компонентов, добавив вызовы reload в родительские модули, каким здесь явля-  
ется A, но это усложнит реализацию модуля.  
Еще лучше было бы написать универсальный инструмент для транзитив-  
ной перезагрузки модулей, сканируя содержимое атрибутов \_\_dict\_\_ модулей

Транзитивная перезагрузка модулей   
679  
и проверяя атрибут type в каждом элементе, чтобы отыскать вложенные мо-  
дули. Такие вспомогательные функции могут рекурсивно вызывать сами себя   
и обходить произвольные цепочки импортирования. Атрибут \_\_dict\_\_ модулей   
был представлен выше, в разделе «Модули – это объекты: метапрограммы»,   
а функция type была представлена в главе 9� от нас требуется лишь объединить   
эти два инструмента.  
Например, модуль reloadall.py, в листинге ниже, содержит функцию reload\_all,   
автоматически выполняющую перезагрузку модуля, каждого импортируемого   
им модуля и так далее, до самого конца каждой цепочки импортирования. Она   
использует словарь, с помощью которого отыскивает уже загруженные моду-  
ли, рекурсию – для обхода цепочек импорта и модуль types из стандартной би-  
блиотеки, в котором просто предопределены значения атрибута type для всех   
встроенных типов. Словарь visited применяется с целью избежать зациклива-  
ния в случае появления рекурсивных или избыточных инструкций импорти-  
рования. Этот прием основан на том факте, что объекты модулей могут играть   
роль ключей словаря (как мы узнали в главе 5, множества также могут пред-  
ложить подобную возможность, если использовать метод visited.add(module)):  
“””  
reloadall.py: транзитивная перезагрузка вложенных модулей  
“””  
   
import types  
from imp import reload # требуется в версии 3.0  
   
def status(module):  
 print(‘reloading’ + module.\_\_name\_\_)  
   
def transitive\_reload(module, visited):  
 if not module in visited: # Пропустить повторные посещения  
 status(module) # Перезагрузить модуль  
 reload(module) # И посетить дочерние модули  
 visited[module] = None  
 for attrobj in module.\_\_dict\_\_.values(): # Для всех атрибутов  
 if type(attrobj) == types.ModuleType: # Рекурсия, если модуль  
 transitive\_reload(attrobj, visited)  
   
def reload\_all(\*args):  
 visited = {}  
 for arg in args:  
 if type(arg) == types.ModuleType:  
 transitive\_reload(arg, visited)  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 import reloadall # Тест: перезагрузить самого себя  
 reload\_all(reloadall) # Должна перезагрузить этот модуль  
 # и модуль types  
Чтобы воспользоваться этой утилитой, импортируйте функцию reload\_all   
и передайте ей имя уже загруженного модуля (как если бы это была встроен-  
ная функция reload). Когда файл запускается как самостоятельная програм-  
ма, его код самопроверки выполнит проверку самого модуля – он должен им-  
портировать самого себя, потому что его собственное имя не будет определено   
в файле без инструкции импортирования (этот программный код работает под   
управлением обеих версий Python, 3.0 и 2.6, и выводит одни и те же данные,

680   
Глава 24. Дополнительные возможности модулей   
благодаря тому, что мы использовали операцию + конкатенации вместо запя-  
той в инструкции print):  
C:\misc> c:\Python30\python reloadall.py  
reloading reloadall  
reloading types  
Ниже приводится результат применения этого модуля, под управлением Py-  
Py-  
thon 3.0, к некоторым модулям из стандартной библиотеки. Обратите внима-  
 3.0, к некоторым модулям из стандартной библиотеки. Обратите внима-  
ние, что модуль os импортируется модулем tkinter, при этом модуль tkinter им-  
портирует модуль sys раньше, чем модуль os (если вы захотите выполнить этот   
тест под управлением Python 2.6, замените имя tkinter на Tkinter):  
>>> from reloadall import reload\_all  
>>> import os, tkinter  
   
>>> reload\_all(os)  
reloading os  
reloading copyreg  
reloading ntpath  
reloading genericpath  
reloading stat  
reloading sys  
reloading errno  
   
>>> reload\_all(tkinter)  
reloading tkinter  
reloading \_tkinter  
reloading tkinter.\_fix  
reloading sys  
reloading ctypes  
reloading os  
reloading copyreg  
reloading ntpath  
reloading genericpath  
reloading stat  
reloading errno  
reloading ctypes.\_endian  
reloading tkinter.constants  
А ниже приводится листинг сеанса, в котором демонстрируются различия   
между обычной и транзитивной операциями перезагрузки, – изменения, вы-  
полненные в двух вложенных файлах, не вступят в силу после перезагрузки,   
если не использовать транзитивную утилиту:  
import b # a.py  
X = 1  
   
import c # b.py  
Y = 2  
   
Z = 3 # c.py  
   
C:\misc> C:\Python30\python  
>>> import a  
>>> a.X, a.b.Y, a.b.c.Z  
(1, 2, 3)

Концепции проектирования модулей   
681  
# Здесь были изменены значения, присваиваемые переменным во всех трех файлах  
   
>>> from imp import reload  
>>> reload(a) # Обычная функция reload перезагружает только   
<module ‘a’ from ‘a.py’> # указанный файл  
>>> a.X, a.b.Y, a.b.c.Z  
(111, 2, 3)  
   
>>> from reloadall import reload\_all  
>>> reload\_all(a)  
reloading a  
reloading b  
reloading c  
>>> a.X, a.b.Y, a.b.c.Z # Вложенные модули также были перезагружены  
(111, 222, 333)  
Я рекомендую поэкспериментировать с этим примером самостоятельно, чтобы   
глубже вникнуть в смысл происходящего, – это еще один инструмент, доступ-  
ный для импортирования, который вы можете добавить в свою собственную   
библиотеку.  
Концепции проектирования модулей  
Как и в случае с функциями, при проектировании модулей используются свои   
правила: вам необходимо подумать о том, какие функции и в какие модули   
будут входить, предусмотреть механизмы взаимодействия между модулями   
и так далее. Все это станет более понятным, когда вы начнете создавать круп-  
ные программы на языке Python, а пока ознакомьтесь с несколькими основны-  
ми положениями:  
 •  
В языке Python вы всегда находитесь в модуле. Нет никакого способа на-  
писать программный код, который не находился бы в каком-нибудь моду-  
ле. Фактически даже программный код, который вводится в интерактив-  
ной оболочке, на самом деле относится к встроенному модулю с именем   
\_\_main\_\_ – единственная уникальная особенность интерактивной оболочки   
состоит в том, что программный код после выполнения сразу же удаляется,   
а результаты выражений выводятся автоматически.  
 •  
Минимизируйте взаимозависимость модулей: глобальные переменные.   
Как и функции, модули работают лучше, когда они написаны как самосто-  
ятельные закрытые компоненты. Следуйте правилу: модули должны быть   
максимально независимы от глобальных имен в других модулях.  
 •  
Максимизируйте согласованность внутри модуля: общая цель. Умень-  
шить взаимозависимость модулей можно за счет увеличения согласован-  
ности отдельного модуля – если все компоненты модуля используются для   
достижения общей цели, маловероятно, что такой модуль будет зависеть от   
внешних имен.  
 •  
Модули должны редко изменять переменные в других модулях. Мы де-  
монстрировали справедливость этого правила на примере программного   
кода в главе 17. Но будет совсем не лишним повторить его: использование   
глобальных переменных из других модулей (в конце концов, это один из   
способов, каким клиенты импортируют службы) – совершенно нормальное

682   
Глава 24. Дополнительные возможности модулей   
явление , тогда как внесение изменений в глобальные переменные в других   
модулях служит признаком проблем с проектированием. Конечно, из этого   
правила есть исключения, но вы должны стремиться обмениваться данны-  
ми через такие механизмы, как аргументы и возвращаемые значения функ-  
ций, не прибегая к прямому изменению переменных в других модулях.   
В противном случае глобальные значения могут попасть в зависимость от   
порядка выполнения инструкций присваивания в других файлах, и такие   
модули будет сложнее понять и приспособить к повторному использованию   
в других программах.  
Для иллюстрации на рис. 24.1 приводится окружение, в котором действуют   
модули. Модули содержат переменные, функции, классы и другие модули   
(если импортируют их). В функциях имеются свои собственные локальные   
переменные. С классами – еще одной разновидностью объектов, которые на-  
ходятся в модулях, – вы познакомитесь в главе 25.  
импортирование  
импортирование  
Переменные  
Функции  
Классы/Типы  
Переменные  
Функции  
Классы/Типы  
Другие модули  
(на языке Python или C)  
Другие модули  
(на языке Python или C)  
Модули  
Переменные  
Функции  
Классы  
Методы  
Переменные  
Переменные  
Переменные  
Рис. 24.1. Среда выполнения модуля. Модули импортируются и сами могут   
импортировать и использовать другие модули, которые могут быть написа-  
ны на языке Python или на других языках программирования, таких как C.   
Модули содержат переменные, функции и классы, с помощью которых    
решают возложенные на них задачи. Их функции и классы также могут    
содержать свои собственные переменные и другие программные элементы.    
Но надо помнить, что на самом верхнем уровне программы – это всего лишь   
наборы модулей  
Типичные проблемы при работе с модулями  
В этом разделе мы рассмотрим обычный набор пограничных ситуаций, которые   
делают жизнь интересной для тех, кто только начинает осваивать язык Python.   
Некоторые из них настолько неочевидны, что трудно привести к ним примеры,   
но в большинстве своем они иллюстрируют важные сведения о языке.

Типичные проблемы при работе с модулями   
683  
Порядок следования инструкций   
на верхнем уровне имеет значение  
Когда модуль впервые импортируется (или загружается повторно), интерпре-  
татор выполняет инструкции в нем одну за другой, сверху вниз. Из этого сле-  
дует несколько замечаний, касающихся опережающих ссылок на переменные,   
которые следует подчеркнуть особо:  
 •  
Инструкции программного кода на верхнем уровне в файле модуля (не вло-  
женные в функцию) выполняются, как только интерпретатор достигает их   
в процессе импортирования. По этой причине он не может ссылаться на   
имена, присваивание которым производится ниже.  
 •  
Программный код внутри функций не выполняется, пока функция не будет   
вызвана, – разрешение имен внутри функций не производится до момента   
их вызова, поэтому они обычно могут ссылаться на имена, расположенные   
в любой части файла.  
Вообще опережающие ссылки доставляют беспокойство только в программ-  
ном коде верхнего уровня, который выполняется немедленно� функции могут   
ссылаться на любые имена. Ниже приводится пример, демонстрирующий опе-  
режающие ссылки:  
func1() # Ошибка: имя “func1” еще не существует  
   
def func1():  
 print(func2()) # OK: поиск имени “func2” будет выполнен позднее  
   
func1() # Ошибка: имя “func2” еще не существует  
   
def func2():  
 return “Hello”  
   
func1() # OK: “func1” и “func2” определены  
Когда этот файл будет импортироваться (или запускаться как самостоятельная   
программа), интерпретатор Python будет выполнять его инструкции сверху   
вниз. Первый вызов func1 потерпит неудачу, потому что инструкция def для   
имени func1 еще не была выполнена. Вызов func2 внутри func1 будет работать   
без ошибок при условии, что к моменту вызова func1 инструкция def func2 уже   
будет выполнена (этого еще не произошло к моменту второго вызова func1 на   
верхнем уровне). Последний вызов func1 в конце файла будет выполнен успеш-  
но, потому что оба имени, func1 и func2, уже определены.  
Смешивание инструкций def с программным кодом верхнего уровня не только   
осложняет его чтение, но еще и ставит его работоспособность в зависимость от   
порядка следования инструкций. Если вам необходимо объединять в модуле   
программный код, выполняемый непосредственно, с инструкциями def, возь-  
мите за правило помещать инструкции def в начало файла, а программный код   
верхнего уровня – в конец файла. При таком подходе ваши функции гаранти-  
рованно будут определены к моменту выполнения программного кода, кото-  
рый их использует.  
Инструкция from создает копии, а не ссылки  
Несмотря на то что инструкция from широко применяется, она часто стано-  
вится источником самых разных проблем. Инструкция from при выполнении

684   
Глава 24. Дополнительные возможности модулей   
присваивания именам в области видимости импортирующего модуля не соз-  
дает синонимы, а копирует имена. Результат будет тем же самым, что и для   
любых других инструкций присваивания в языке Python, но есть одно тонкое   
отличие, особенно когда программный код, использующий объекты совместно,   
находится в разных файлах. Например, предположим, что у нас имеется сле-  
дующий модуль (nested1.py):  
# nested1.py  
X = 99  
def printer(): print(X)  
Если импортировать эти два имени с помощью инструкции from в другом моду-  
ле (nested2.py), будут получены копии этих имен, а не ссылки на них. Измене-  
ние имени в импортирующем модуле приведет к изменениям только локальной   
версии этого имени, а имя в модуле nested1.py будет иметь прежнее значение:  
# nested2.py  
from nested1 import X, printer # Копировать имена  
X = 88 # Изменит только локальную версию “X”!  
printer() # X в nested1 по-прежнему будет равно 99  
   
% python nested2.py  
99  
Однако если выполнить импорт всего модуля с помощью инструкции import   
и затем изменить значение с использованием полного имени, это приведет к из-  
менению имени в файле nested1.py. Квалифицированное имя направляет ин-  
терпретатор к имени в указанном объекте модуля, а не к имени в импортирую-  
щем модуле nested3.py:  
# nested3.py  
import nested1 # Импортировать модуль целиком  
nested1.X = 88 # OK: изменяется имя X в nested1  
nested1.printer()  
   
% python nested3.py  
88  
Инструкция from \* может затушевывать   
смысл переменных  
Я упоминал об этом ранее, но оставил подробности до этого момента. Посколь-  
ку в инструкции from module import \* не указываются необходимые имена пере-  
менных, она может непреднамеренно перезаписать имена, уже используемые   
в области видимости импортирующего модуля. Хуже того, это может ослож-  
нить поиск модуля, откуда исходит переменная, вызвавшая проблемы. Это   
особенно верно, когда форма инструкции from \* используется более чем в одном   
импортированном файле.  
Например, если инструкция from \* применяется к трем модулям, то у вас не   
будет иного способа узнать, какая в действительности вызывается функция,   
кроме как выполнить поиск в трех разных файлах модулей (каждый из кото-  
рых может находиться в отдельном каталоге):  
>>> from module1 import \* # Плохо: может незаметно перезаписать мои имена  
>>> from module2 import \* # Еще хуже: нет никакого способа понять,

Типичные проблемы при работе с модулями   
685  
>>> from module3 import \* # что мы получили!  
>>> ...  
   
>>> func() # Ну???  
Решение опять же заключается в том, чтобы так не делать: старайтесь явно   
перечислять требуемые атрибуты в инструкции from и ограничивайте приме-  
нение формы from \* одним модулем на файл. В этом случае любые неопределен-  
ные имена, согласно дедуктивному методу, должны находиться в модуле, кото-  
рый импортируется единственной инструкцией from \*. Этой проблемы вообще   
можно избежать, если всегда использовать инструкцию import вместо from, но   
это слишком сложно – как и многое другое в программировании, инструкция   
from – очень удобный инструмент при разумном использовании. Даже этот при-  
мер не может расцениваться, как неправильный, – этот способ вполне может   
применяться в программах, для большего удобства собирающих имена в одном   
месте, при условии, что это место хорошо известно.  
Функция reload может не оказывать влияния,   
если импорт осуществлялся инструкцией from  
Это еще одна ловушка, связанная с инструкцией from: как уже говорилось ра-  
нее, инструкция from копирует (присваивает) имена при выполнении, поэтому   
нет никакой обратной связи с модулем, откуда были скопированы имена. Име-  
на, скопированные инструкцией from, просто становятся ссылками на объек-  
ты, на которые ссылались по тем же именам в импортируемом модуле, когда   
была выполнена инструкция from.  
Вследствие этого повторная загрузка импортируемого модуля может не оказы-  
вать воздействия на клиентов, которые импортировали его имена с помощью   
инструкции from. То есть имена в модулях-клиентах будут по-прежнему ссы-  
латься на оригинальные объекты, полученные инструкцией from, даже если   
имена в оригинальном модуле будут переопределены:  
from module import X # X может не измениться в результате перезагрузки!  
. . .  
from imp import reload  
reload(module) # Изменится модуль, но не мои имена  
X # По-прежнему ссылается на старый объект  
Чтобы сделать повторную загрузку более эффективной, используйте инструк-  
цию import и полные имена переменных вместо инструкции from. Поскольку   
полные имена всегда ведут обратно в импортированный модуль, после повтор-  
ной загрузки они автоматически будут связаны с новыми именами в модуле.  
import module # Получить объект модуля, а не имена  
. . .  
from imp import reload  
reload(module) # Изменит непосредственно сам объект модуля  
module.X # Текущее значение X: отражает результат перезагрузки  
reload, from и тестирование в интерактивной оболочке  
На практике предыдущая проблема имеет более глубокие корни, чем может   
показаться. В главе 3 уже говорилось, что из-за возникающих сложностей луч-  
ше не использовать операции импорта и повторной загрузки для запуска про-

686   
Глава 24. Дополнительные возможности модулей   
грамм. Дело еще больше осложняется, когда в игру вступает инструкция from.   
Начинающие осваивать язык Python часто сталкиваются с проблемой, описы-  
ваемой здесь. Представим, что после открытия модуля в окне редактирования   
текста вы запускаете интерактивный сеанс, чтобы загрузить и протестировать   
модуль с помощью инструкции from:  
from module import function  
function(1, 2, 3)  
Отыскав ошибку, вы переходите обратно в окно редактирования, исправляете   
ее и пытаетесь повторно загрузить модуль следующим способом:  
from imp import reload  
reload(module)  
Но вы не получите ожидаемого эффекта – инструкция from создала имя func-  
tion, но не создала имя module. Чтобы сослаться на модуль в функции reload, его   
сначала необходимо импортировать инструкцией import:  
from imp import reload  
import module  
reload(module)  
function(1, 2, 3)  
Однако и в этом случае вы не получите ожидаемого эффекта – функция reload   
обновит объект модуля, но, как говорилось в предыдущем разделе, имена, та-  
кие как function, скопированные из модуля ранее, по-прежнему продолжают   
ссылаться на старые объекты (в данном случае – на первоначальную версию   
функции). Чтобы действительно получить доступ к новой версии функции, по-  
сле перезагрузки модуля ее необходимо вызывать как module.function или по-  
вторно запустить инструкцию from:  
from imp import reload  
import module  
reload(module)  
from module import function # Или оставить этот прием и использовать   
function(1, 2, 3) # module.function()  
Теперь наконец-то нам удалось запустить новую версию функции.  
Как видите, прием, основанный на использовании функции reload в паре   
с инструкцией from, имеет врожденные проблемы: вы не только должны не   
забывать перезагружать модуль после импорта, но и не забывать повторно за-  
пускать инструкции from после перезагрузки модуля. Это достаточно сложно,   
чтобы время от времени сбивать с толку даже опытного программиста. (При   
этом в Python 3.0 описываемая ситуация только усугубилась, потому что нуж-  
Python 3.0 описываемая ситуация только усугубилась, потому что нуж-  
 3.0 описываемая ситуация только усугубилась, потому что нуж-  
но не забыть импортировать саму функцию reload!)  
Вы не должны ожидать, что функция reload и инструкция from будут безуко-  
ризненно работать в паре. Лучше всего вообще не объединять их – используйте   
функцию reload в паре с инструкцией import или запускайте свои программы   
другими способами, как предлагалось в главе 3, например, выбирая пункт Run   
(Запустить) → Run Module (Запустить модуль) в меню среды разработки IDLE щелч-  
ком мыши на ярлыке файла, из командной строки системы или с помощью   
встроенной функции exec.

Типичные проблемы при работе с модулями   
687  
Рекурсивный импорт с инструкцией from   
может не работать  
Напоследок я оставил самую странную (и, к счастью, малоизвестную) пробле-  
му. В ходе импортирования инструкции в файле выполняются от начала и до   
конца, поэтому необходимо быть внимательнее, когда используются модули,   
импортирующие друг друга (эта ситуация называется рекурсивным  импор-  
том). Поскольку не все инструкции в модуле могут быть выполнены к моменту   
запуска процедуры импортирования другого модуля, некоторые из его имен   
могут оказаться еще не определенными.  
Если вы используете инструкцию import, чтобы получить модуль целиком, это   
может иметь, а может не иметь большого значения – имена модуля не будут   
доступны, пока позже не будут использованы полные имена для получения их   
значений. Но если для получения определенных имен используется инструк-  
ция from, имейте в виду, у вас будет доступ только к тем именам, которые уже   
были определены в этом модуле.  
Например, рассмотрим следующие модули, recur1 и recur2. Модуль recur1 соз-  
дает имя X и затем импортирует recur2 до того, как присвоит значение имени Y.   
В этом месте модуль recur2 может импортировать модуль recur1 целиком с по-  
мощью инструкции import (он уже существует во внутренней таблице модулей   
интерпретатора), но если используется инструкция from, ей будет доступно   
только имя X, а имя Y, которому будет присвоено значение только после ин-  
струкции import в recur1, еще не существует, поэтому возникнет ошибка:  
# recur1.py  
X = 1  
import recur2 # Запустить recur2, если он еще не был импортирован  
Y = 2  
   
# recur2.py  
from recur1 import X # OK: “X” уже имеет значение  
from recur1 import Y # Ошибка: “Y” еще не существует  
   
C:\misc> C:\Python30\python  
>>> import recur1  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
 File “recur1.py”, line 2, in ?  
 import recur2  
 File “recur2.py”, line 2, in ?  
 from recur1 import Y # Ошибка: “Y” еще не существует  
ImportError: cannot import name Y  
При рекурсивном импорте модуля recur1 и модуля recur2 интерпретатор не бу-  
дет повторно выполнять инструкции модуля recur1 (в противном случае это   
могло бы привести к бесконечному циклу), но пространство имен модуля recur1   
еще не заполнено до конца к моменту, когда он импортируется модулем recur2.  
Решение? Не используйте инструкцию from в операции рекурсивного импорта   
(в самом деле!). Интерпретатор не зациклится, если вы все-таки сделаете это,   
но ваша программа попадет в зависимость от порядка следования инструкций   
в модулях.

688   
Глава 24. Дополнительные возможности модулей   
Существует два способа решения этой проблемы:  
 •  
Обычно можно ликвидировать рекурсивный импорт, подобный приведен-  
ному, правильно проектируя модули: увеличить согласованность внутри   
модуля и уменьшить взаимозависимость между модулями – это самое пер-  
вое, что стоит попробовать сделать.  
 •  
Если от циклов не удается избавиться полностью, попробуйте отсрочить   
обращение к именам модуля, используя инструкции import и полные име-  
на (вместо инструкции from), или поместите инструкции from либо внутрь   
функций (чтобы они не вызывались в программном коде верхнего уровня),   
либо ближе к концу файла, чтобы отложить их выполнение.  
В заключение  
В этой главе был рассмотрен ряд дополнительных концепций, связанных с мо-  
дулями. Мы изучили приемы сокрытия данных, включение новых особен-  
ностей языка из модуля \_\_future\_\_, возможности использования переменной   
\_\_name\_\_, транзитивную перезагрузку модулей, импортирование модулей по   
имени в виде строки и многое другое. Мы также исследовали проблемы проек-  
тирования модулей и познакомились с типичными ошибками при работе с мо-  
дулями, что позволит вам избежать их в своем программном коде.  
Со следующей главы мы приступим к изучению объектно-ориентированного   
инструмента языка Python – класса. Большая часть сведений, рассмотрен-  
ных в последних нескольких главах, применима и здесь – классы располага-  
ются в модулях и также являются пространствами имен, но они добавляют   
еще один элемент в поиск атрибутов, который называется поиск в цепочке на-  
следования. Поскольку это последняя глава в этой части книги, прежде чем   
углубиться в объектно-ориентированное программирование, обязательно вы-  
полните упражнения к этой части книги. Но перед этим попробуйте ответить   
на контрольные вопросы главы, чтобы освежить в памяти темы, рассматривав-  
шиеся здесь.  
Закрепление пройденного  
Контрольные вопросы  
1. Что важно знать о переменных в программном коде верхнего уровня моду-  
ля, имена которых начинаются с одного символа подчеркивания?  
2. Что означает, когда переменная \_\_name\_\_ модуля имеет значение “\_\_main\_\_”?  
3. Если пользователь вводит имя модуля в ответ на запрос программы, как   
импортировать этот модуль?  
4. Чем отличается изменение списка sys.path от изменения значения перемен-  
ной окружения PYTHONPATH?  
5. Импорт будущих изменений в языке возможен с помощью модуля \_\_fu-  
fu-  
ture\_\_, а возможен ли импорт из прошлого?

Закрепление пройденного   
689  
Ответы  
1. Переменные в программном коде верхнего уровня модуля, чьи имена начи-  
наются с одного символа подчеркивания, не копируются при импортирова-  
нии с помощью инструкции from \*. Однако они доступны при использовании   
инструкции import и обычной формы инструкции from.  
2. Если переменная \_\_name\_\_ модуля содержит строку “\_\_main\_\_”, это означает,   
что файл выполняется как самостоятельный сценарий, а не был импорти-  
рован как модуль другим файлом в программе. То есть файл используется   
как программа, а не как библиотека.  
3. Как правило, ввод пользователя поступает в сценарий в виде строки – что-  
бы импортировать модуль по имени, заданному в виде строки, можно со-  
брать и выполнить инструкцию import с помощью функции exec или пере-  
дать строку с именем функции \_\_import\_\_.  
4. Изменения в sys.path воздействуют только на работающую программу и но-  
сят временный характер – изменения будут утеряны сразу же после завер-  
шения программы. Содержимое переменной окружения PYTHONPATH хранит-  
ся в операционной системе – оно оказывает воздействие на все программы,   
выполняемые на этом компьютере, а изменения сохраняются после завер-  
шения программ.  
5. Нет, мы не можем импортировать из прошлого. Мы можем установить (или   
упорно использовать) более старую версию языка, но, как правило, самая   
лучшая версия – это последняя версия Python.1  
Упражнения к пятой части  
Решения приводятся в разделе «Часть V, Модули» приложения A.  
1. Основы импортирования. Напишите программу, которая подсчитывает ко-  
личество строк и символов в файле (в духе утилиты wc в операционной систе-  
мы UNIX). В своем текстовом редакторе создайте модуль с именем mymod.  
py, который экспортирует три имени:  
 •  
Функцию countLines(name), которая читает входной файл и подсчитывает   
число строк в нем (подсказка: большую часть работы можно выполнить   
с помощью метода file.readlines, а оставшуюся часть – с помощью функ-  
ции len).  
 •  
Функцию countChars(name), которая читает входной файл и подсчитыва-  
ет число символов в нем (подсказка: метод file.read возвращает единую   
строку).  
 •  
Функцию test(name), которая вызывает две предыдущие функции с за-  
данным именем файла. Вообще говоря, имя файла можно жестко   
определить в программном коде, принимать ввод от пользователя или   
принимать имя как параметр командной строки через список sys.argv –   
но пока исходите из предположения, что оно передается как аргумент   
функции.  
Все три функции в модуле mymod должны принимать имя файла в виде стро-  
ки. Если размер любой из функций превысит две-три строки, это значит, что   
вы делаете лишнюю работу, – используйте подсказки, которые я вам дал!  
1   
Последняя стабильная версия. – Примеч. перев.

690   
Глава 24. Дополнительные возможности модулей   
Затем проверьте свой модуль в интерактивной оболочке, используя ин-  
струкцию import и полные имена экспортируемых функций. Следует ли до-  
бавить в переменную PYTHONPATH каталог, где находится ваш файл mymod.  
py? Попробуйте проверить модуль на самом себе: например, test(“mymod.py”).   
Обратите внимание, что функция test открывает файл дважды, – если вы   
достаточно честолюбивы, попробуйте оптимизировать программный код,   
передавая двум функциям счета объект открытого файла (подсказка: метод   
file.seek(0) выполняет переустановку указателя в начало файла).  
2. from/from \*. Проверьте модуль mymod из упражнения 1 в интерактивной обо-  
лочке, используя для загрузки экспортируемых имен инструкцию from –   
сначала по имени, а потом с помощью формы from \*.  
3. \_\_main\_\_. Добавьте в модуль mymod строку, в которой автоматически про-  
изводился бы вызов функции test, только когда модуль выполняется как   
самостоятельный сценарий, а не во время импортирования. Добавляемая   
вами строка, вероятно, должна содержать проверку значения атрибута   
\_\_name\_\_ на равенство строке “\_\_main\_\_”, как было показано в этой главе.   
Попробуйте запустить модуль из системной командной строки, затем им-  
портируйте модуль и проверьте работу функций в интерактивном режиме.   
Будут ли работать функции в обоих режимах?  
4. Вложенное  импортирование. Напишите второй модуль myclient.py, кото-  
рый импортирует модуль mymod и проверяет работу его функций, затем за-  
пустите myclient из системной командной строки. Будут ли доступны функ-  
ции из mymod на верхнем уровне myclient, если импортировать их с помощью   
инструкции from? А если они будут импортированы с помощью инструкции   
import? Попробуйте реализовать оба варианта в myclient и проверить в инте-  
рактивном режиме, импортируя модуль myclient и проверяя содержимое его   
атрибута \_\_dict\_\_.  
5. Импорт  пакетов. Импортируйте ваш файл из пакета. Создайте каталог   
с именем mypkg, вложенный в каталог, находящийся в пути поиска моду-  
лей. Переместите в него файл mymod.py, созданный в упражнении 1 или 3,   
и попробуйте импортировать его как пакет, инструкцией import mypkg.mymod.  
Вам потребуется добавить файл \_\_init\_\_.py в каталог, куда был помещен   
ваш модуль. Это упражнение должно работать на всех основных платфор-  
мах Python (это одна из причин, почему в языке Python в качестве разде-  
лителя компонентов пути используется символ «.»). Каталог пакета может   
быть простым подкаталогом в вашем рабочем каталоге – в этом случае он   
будет обнаружен интерпретатором при поиске в домашнем каталоге и вам   
не потребуется настраивать путь поиска. Добавьте какой-нибудь программ-  
ный код в \_\_init\_\_.py и посмотрите, будет ли он выполняться при каждой   
операции импортирования.  
6. Повторная загрузка. Поэкспериментируйте с возможностью повторной за-  
грузки модуля: выполните тесты, которые приводятся в примере changer.py   
в главе 22, многократно изменяя текст сообщения и/или поведение модуля,   
без остановки интерактивного сеанса работы с интерпретатором Python.   
В зависимости от операционной системы файл модуля можно было бы ре-  
дактировать в другом окне или, приостановив интерпретатор, редактиро-  
вать модуль в том же окне (в UNIX комбинация клавиш Ctrl-Z обычно при-  
водит к приостановке текущего процесса, а команда fg – возобновляет его   
работу).

Закрепление пройденного   
691  
7. Циклический импорт.1 В разделе, где описываются проблемы, связанные   
с рекурсивным импортом, импортирование модуля recur1 вызывает появ-  
ление ошибки. Но если перезапустить интерактивный сеанс работы с ин-  
терпретатором и предварительно импортировать модуль recur2, ошибка не   
возникает – проверьте этот факт сами. Как вы думаете, почему импортиро-  
вание recur2 проходит без ошибок, а импортирование recur1 с ошибками?   
(Подсказка: интерпретатор Python сохраняет новые модули во встроенной   
таблице (словаре) sys.modules перед их запуском, независимо от того, «завер-  
шен» модуль или нет.) Теперь попробуйте запустить recur1 как самостоя-  
тельный сценарий: python recur1.py. Получите ли вы ту же самую ошибку,   
которая возникает при импортировании recur1 в интерактивной оболочке?   
Почему? (Подсказка: когда модули запускаются как самостоятельные про-  
граммы, они не импортируются, поэтому здесь возникает тот же эффект,   
как и при импортировании recur2 в интерактивной оболочке, – recur2 яв-  
ляется первым импортируемым модулем.) Что произойдет, если запустить   
recur2 как самостоятельный сценарий? Почему?  
1   
Обратите внимание, что циклическое импортирование чрезвычайно редко встречает-  
ся на практике. Однако если вы в состоянии понять, в чем заключается проблема при   
использовании циклического импорта, это будет означать, что вы достаточно много   
знаете о семантике языка Python.

Часть VI.  
Классы и ООП

Глава 25.  
   
ООП: общая картина  
До сих пор в этой книге мы использовали термин «объект» в общем смысле.   
В действительности весь программный код, написанный нами до сих пор,   
был основан  на  объектах  (object-based), – везде в своих сценариях мы пере-  
давали объекты, использовали их в выражениях, вызывали их методы и так   
далее. Однако, чтобы наш программный код стал по-настоящему объектно-  
ориентированным (ОО), наши объекты должны участвовать в иерархии насле-  
дования.   
Начиная с этой главы, мы приступаем к исследованию классов в языке Python –   
программных компонентов, используемых для реализации новых типов объ-  
ектов, поддерживающих наследование. Классы – это основные инструменты   
объектно-ориентированного программирования (ООП) в языке Python, поэто-  
му в этой части книги мы попутно рассмотрим основы ООП. ООП предлагает   
другой, часто более эффективный подход к программированию, при котором   
мы разлагаем программный код на составляющие, чтобы уменьшить его избы-  
точность, и пишем новые программы, адаптируя имеющийся программный   
код, а не изменяя его.  
Классы в языке Python создаются с помощью новой для нас инструкции: ин-  
струкции class. Как вы увидите, объекты, определяемые с помощью классов,   
очень напоминают встроенные типы, которые мы изучали ранее в этой книге.   
В действительности классы лишь применяют и дополняют понятия, которые   
мы уже рассмотрели. Грубо говоря – они представляют собой пакеты функций,   
которые в основном используют и обрабатывают объекты встроенных типов.   
Основное назначение классов состоит в том, чтобы создавать и манипулиро-  
вать новыми объектами, а кроме того, они поддерживают механизм наследо-  
вания – совсем иной способ адаптации программного кода и повторного его ис-  
пользования, чем мы рассматривали ранее.  
Одно предварительное замечание: ООП в языке Python является необязатель-  
ным к применению и на первых порах вам не обязательно использовать клас-  
сы. Многие задачи можно реализовать с помощью более простых конструкций,   
таких как функции, или даже просто программируя на верхнем уровне в сце-  
нарии. Поскольку использование классов требует некоторого предварительно-  
го планирования, они представляют больший интерес для тех, кто работает

696   
Глава 25. ООП: общая картина   
в стратегическом режиме (участвует в долгосрочной разработке программ-  
ных продуктов), чем для тех, кто работает в тактическом режиме (где на раз-  
работку дается очень короткое время).  
Однако, как вы увидите в этой части книги, классы являются одним из самых   
мощных инструментов в языке Python. При грамотном использовании клас-  
сы способны радикально сократить время, затрачиваемое на разработку. Они   
также присутствуют в таких популярных инструментах, как tkinter GUI API,   
поэтому для большинства программистов, использующих язык Python, зна-  
ние основ работы с классами будет как минимум полезно.  
Зачем нужны классы?  
Помните, как я говорил вам, что «программы выполняют некоторые действия   
над чем-то»? Выражаясь простым языком, классы – это всего лишь способ   
определить новое «что-то», они являются отражением реальных объектов   
в мире программ. Например, предположим, что мы решили реализовать гипо-  
тетическую машину по изготовлению пиццы, которую мы использовали в ка-  
честве примера в главе 16. Если реализовать ее на основе классов, мы могли   
бы смоделировать более близкое к реальности строение машины и взаимосвязь   
между ее механизмами. Полезными здесь оказываются два аспекта ООП:  
Наследование  
Машина по изготовлению пиццы – это разновидность машин, поэтому она   
обладает обычными свойствами, характерными для машин. В терминах   
ООП это называется «наследованием» свойств более общей категории ма-  
шин. Эти общие свойства необходимо реализовать всего один раз, после   
чего мы сможем использовать их для моделирования любых типов машин,   
которые нам потребуется создать в будущем.  
Композиция  
Машины по изготовлению пиццы состоят из множества компонентов, ко-  
торые все вместе работают как единое целое. Например, нашей машине   
необходимы манипуляторы, чтобы раскатывать тесто, двигатели, чтобы   
перемещаться к духовке, и так далее. На языке ООП наша машина – это   
пример композиции� она содержит другие объекты, которые активизиру-  
ются для выполнения определенных действий. Каждый компонент может   
быть оформлен как класс, который определяет свое поведение и принципы   
взаимодействия.  
Общие идеи ООП, такие как наследование и композиция, применимы к любым   
приложениям, которые могут быть разложены на ряд объектов. Например,   
в типичных программах с графическим интерфейсом сам интерфейс создает-  
ся как набор визуальных элементов управления – кнопок, меток и так далее,   
которые рисуются на экране в тот момент, когда выполняется рисование их   
контейнеров (композиция). Кроме того, мы можем создать свои собственные   
визуальные элементы – кнопки с уникальными шрифтами, метки с новыми   
цветовыми схемами, которые будут представлять собой специализированные   
версии более общих интерфейсных элементов (наследование).  
Если говорить более точно, с точки зрения программирования классы – это   
программные компоненты на языке Python, точно такие же, как функции   
и модули: они представляют собой еще один способ упаковки выполняемого

ООП с высоты 30 000 футов   
697  
кода и данных. Кроме того, классы определяют свои пространства имен, так   
же как и модули. Но в отличие от других программных компонентов, которые   
мы уже видели, классы имеют три важных отличия, которые делают их более   
полезными, когда дело доходит до создания новых объектов:  
Множество экземпляров  
Классы по своей сути являются фабриками по созданию объектов. Каждый   
раз, когда вызывается класс, создается новый объект со своим собственным   
пространством имен. Каждый объект, созданный из класса, имеет доступ   
к атрибутам класса и получает в свое распоряжение собственное простран-  
ство имен для своих собственных данных, отличных от данных других объ-  
ектов.  
Адаптация через наследование  
Классы также поддерживают такое понятие ООП, как наследование, – мы   
можем расширять возможности класса, переопределяя его атрибуты за   
пределами самого класса. В более общем смысле классы могут создавать   
иерархии пространств имен, которые определяют имена для использования   
объектами, созданными из классов в иерархии.  
Перегрузка операторов  
Обеспечивая специальный протокол оформления методов, классы могут   
определять объекты, над которыми можно производить какие-то из опера-  
ций, которые мы знаем по работе со встроенными типами. Например, объ-  
екты, созданные из классов, могут подвергаться операции извлечения сре-  
за, конкатенации, извлечения элементов по их индексам и так далее. Язык   
Python предоставляет специальные обработчики, которые могут использо-  
 предоставляет специальные обработчики, которые могут использо-  
ваться классами для перехвата и реализации любой встроенной операции.  
ООП с высоты 30 000 футов  
Прежде чем мы увидим, что все это означает в терминах программного кода,   
я хотел бы сказать несколько слов об общих идеях, лежащих в основе ООП. Если   
раньше вам не приходилось делать что-нибудь в объектно-ориентированном   
стиле, некоторые термины в этой главе при первом знакомстве могут вам пока-  
заться немного сложными. Кроме того, смысл этих терминов может ускользать   
от вас, пока вы не получите возможность познакомиться с тем, как программи-  
сты применяют их в крупных программах. ООП – это не только технология, но   
еще и опыт.  
Поиск унаследованных атрибутов  
Самое интересное, что ООП в языке Python проще в изучении и использова-  
Python проще в изучении и использова-  
 проще в изучении и использова-  
нии, чем в других языках программирования, таких как C++ и Java. Будучи   
языком сценариев с динамической типизацией, Python ликвидирует синтак-  
Python ликвидирует синтак-  
 ликвидирует синтак-  
сическую перегруженность и сложность, свойственные ООП в других языках.   
Фактически ООП в языке Python сводится к выражению:  
object.attribute  
Мы использовали это выражение на протяжении всей книги для организации   
доступа к атрибутам модуля, вызова методов объектов и так далее. Однако,   
когда подобное выражение применяется к объекту, полученному с помощью

698   
Глава 25. ООП: общая картина   
инструкции class, интерпретатор начинает поиск – поиск в дереве связанных   
объектов, который заканчивается, как только будет встречено первое появле-  
ние атрибута attribute. Когда в дело вступают классы, это выражение на языке   
Python можно перевести в следующее выражение на естественном языке:  
Найти первое вхождение атрибута attribute, просмотрев объект object, а по-  
том все классы в дереве наследования выше него, снизу вверх и слева на-  
право.  
Другими словами, выборка атрибутов производится в результате простого по-  
иска по дереву. Мы называем эту процедуру поиском в дереве наследования,   
потому что объекты, расположенные в дереве ниже, наследуют атрибуты,   
имеющиеся у объектов, расположенных в дереве выше. Так как поиск проис-  
ходит в направлении снизу вверх, объекты в некотором смысле оказываются   
связаны в древовидную структуру, представляющую собой объединение всех   
атрибутов, определяемых всеми их родителями в дереве.  
В языке Python все это понимается буквально: с помощью программного кода   
мы действительно создаем деревья связанных объектов, и интерпретатор дей-  
ствительно поднимается вверх по дереву, пытаясь во время выполнения оты-  
скать атрибуты всякий раз, когда мы используем выражение object.attribute.   
Чтобы было более понятно, на рис. 25.1 приводится пример одного из таких   
деревьев.  
C1  
C2  
C3  
I1  
I2  
.x  
.z  
.w  
.z  
.x  
.y  
.name  
.name  
Рис. 25.1. Дерево классов с двумя экземплярами внизу (I1 и I2), классом выше   
их (C1) и двумя суперклассами в самом верху (C2 и C3). Все эти объекты   
являются пространствами имен (пакетами переменных), а наследование   
означает просто поиск в дереве снизу вверх, цель которого найти самое ниж-  
нее вхождение атрибута с требуемым именем. Программный код задает   
структуру таких деревьев  
На этом рисунке изображено дерево из пяти объектов, помеченных именами   
переменных. Каждый из этих объектов обладает набором атрибутов. Если го-  
ворить более точно, это дерево связывает вместе три объекта классов (в овалах,   
C1, C2 и C3) и два объекта экземпляров (в прямоугольниках, I1 и I2) в иерархию   
наследования. Обратите внимание, что в модели объектов в языке Python клас-  
сы и экземпляры порождаются от двух разных типов объектов:  
Классы  
Играют роль фабрик экземпляров. Их атрибуты обеспечивают поведение –   
данные и функции – то есть наследуются всеми экземплярами, созданны-

ООП с высоты 30 000 футов   
699  
ми от них (например, функция, вычисляющая зарплату служащего, исходя   
из часового тарифа).  
Экземпляры  
Представляют конкретные элементы программы. Их атрибуты хранят дан-  
ные, которые могут отличаться в конкретных объектах (например, номер   
карточки социального страхования служащего).  
В терминах деревьев поиска экземпляры наследуют атрибуты своих классов,   
а классы наследуют атрибуты всех других классов, находящихся в дереве выше.  
На рис. 25.1 можно продолжить категоризацию классов по относительным по-  
ложениям овалов в дереве. Классы, расположенные в дереве выше (такие как   
C2 и C3), мы обычно называем суперклассами, а классы, расположенные ниже   
(такие как C1), называются подклассами.1 Эти термины обозначают относи-  
тельное положение в дереве и исполняемые роли. Суперклассы реализуют по-  
ведение, общее для всех их подклассов, но из-за того, что поиск производится   
снизу верх, подклассы могут переопределять поведение, определяемое их су-  
перклассами, переопределяя имена суперклассов ниже в дереве.  
Т.к. эти последние несколько слов отражают основную суть адаптации про-  
граммного обеспечения в ООП, давайте подробнее рассмотрим эту концепцию.   
Предположим, что мы создали дерево, приведенное на рис. 25.1, и затем пишем:  
I2.w  
Этот программный код демонстрирует использование механизма наследова-  
ния. Так как это выражение вида object.attribute, оно приводит к запуску по-  
иска в дереве, изображенном на рис. 25.1, – интерпретатор приступает к поис-  
ку атрибута w, начиная с I2, и движется вверх по дереву. В частности, он будет   
просматривать объекты в следующем порядке:   
I2, C1, C2, C3  
и остановится, как только будет найден первый атрибут с таким именем (или   
возбудит исключение, если атрибут w вообще не будет найден). В этом случае   
поиск будет продолжаться, пока не будет достигнут объект C3, поскольку атри-  
бут w имеется только в этом объекте. Другими словами, имя I2.w в терминах   
автоматического поиска будет обнаружено, как C3.w. В терминологии ООП это   
называется I2 «наследует» атрибут w от C3.  
В конечном итоге два экземпляра наследуют от своих классов четыре атрибута:   
w, x, y и z. Другие атрибуты будут найдены в различных местах в дереве. На-  
пример:  
 •  
Для I1.x и I2.x атрибут x будет найден в C1, где поиск остановится, потому   
что C1 находится в дереве ниже, чем C2.  
 •  
Для I1.y и I2.y атрибут y будет найден в C1, где поиск остановится, потому что   
это единственное место, где он появляется.  
 •  
Для I1.z и I2.z атрибут z будет найден в C2, потому что C2 находится в дереве   
левее, чем C3.  
1   
В других книгах можно также встретить такие термины, как базовые классы и до-  
черние классы, которые используются для обозначения суперклассов и подклассов   
соответственно.

700   
Глава 25. ООП: общая картина   
 •  
Для I2.name атрибут name будет найден в I2, в этом случае поиск по дереву во-  
обще осуществляться не будет.  
Проследите эти пути поиска в дереве на рис. 25.1, чтобы понять, как выполня-  
ется поиск по дереву наследования в языке Python.  
Первый элемент в предыдущем списке является, пожалуй, самым важным,   
потому что C1 переопределяет атрибут x ниже в дереве, тем самым замещая   
версию атрибута, расположенную выше, в C2. Как вы увидите через мгновение,   
такие переопределения составляют основу адаптации программного обеспече-  
ния в ООП – переопределяя и замещая атрибут, C1 эффективно изменяет свое   
поведение, унаследованное от своего суперкласса.  
Классы и экземпляры  
Являясь отдельными типами объектов в модели языка Python, классы и экзем-  
пляры, которые мы объединили в иерархические деревья, выполняют прак-  
тически одну и ту же роль: каждый из этих типов служит некоторым пред-  
ставлением пространства имен – пакета переменных и места, где определя-  
ются атрибуты. Если вследствие этого классы и экземпляры покажутся вам   
похожими на модули, то можно считать и так, но при этом объекты в деревьях   
классов содержат автоматически определяемые ссылки на другие объекты   
пространств имен, и классы соответствуют инструкциям, а не файлам.  
Основное различие между классами и экземплярами состоит в том, что клас-  
сы представляют собой своего рода фабрики по производству экземпляров. На-  
пример, в реалистичном приложении у нас мог бы быть класс Employee, опреде-  
ляющий характеристики служащего, – из этого класса мы можем создавать   
фактические экземпляры класса Employee. Это еще одно различие между клас-  
сами и модулями: у нас всегда имеется всего один экземпляр заданного модуля   
в памяти (именно по этой причине приходится перезагружать модуль, чтобы   
загрузить в память новый программный код), но в случае с классами можно   
создать столько экземпляров, сколько потребуется.  
В оперативном отношении, у классов обычно имеются функции, присоединен-  
ные к ним (например, computeSalary), а у экземпляров – элементы данных, ис-  
пользуемые функциями класса (например, hoursWorked). Фактически объектно-  
ориентированная модель не так сильно отличается от классической модели   
обработки данных, основанной на программах и записях, – в ООП экземпляры   
подобны записям с «данными», а классы – «программам», обрабатывающими   
эти записи. Однако в ООП имеется также понятие иерархии наследования, ко-  
торая обеспечивает более широкие возможности адаптации программного обе-  
спечения, чем более ранние модели.  
Вызовы методов классов  
В предыдущем разделе мы видели, как атрибут I2.w в нашем примере дерева   
классов транслируется в C2.w при выполнении поиска в дереве наследования.   
Не менее важно понять, что точно так же наследуются и методы (то есть функ-  
ции, присоединенные к классам в виде атрибутов).  
Если ссылка I2.w – это вызов функции, тогда в действительности это выраже-  
ние означает: «вызвать функцию С3.w для обработки I2». То есть интерпретатор   
Python автоматически отобразит вызов I2.w() на вызов C3.w(), передав унасле-  
дованной функции экземпляр в виде первого аргумента.

ООП с высоты 30 000 футов   
701  
Фактически всякий раз, когда вызывается функция, присоединенная к клас-  
су, подразумевается не класс целиком, а экземпляр класса. Этот подразуме-  
ваемый экземпляр, или контекст, является одной из причин, почему данная   
модель называется объектно-ориентированной, – всегда существует объект,   
над которым выполняются действия. В более реалистичном примере мы могли   
бы вызывать метод с именем giveRaise, присоединенный как атрибут к классу   
Employee, – вызов этого метода был бы бессмысленным без указания служаще-  
го, которому дается надбавка к зарплате.  
Как мы увидим позднее, Python передает методам подразумеваемый экзем-  
пляр в виде специального первого аргумента, в соответствии с соглашением   
именуемого self. Мы также узнаем, что методы могут вызываться как через эк-  
земпляры (например, bob.giveRaise()), так и через классы (например, Employee.  
giveRaise(bob)), причем обе формы играют одну и ту же роль в наших сценари-  
ях. Чтобы увидеть, как методы принимают свои подразумеваемые экземпля-  
ры, нам необходимо рассмотреть примеры программного кода.  
Создание деревьев классов  
Несмотря на всю отвлеченность наших разговоров, тем не менее за всеми эти-  
ми идеями стоит реальный программный код. Мы создаем деревья и объекты   
в них с помощью инструкций class и вызовов классов, которые позднее мы рас-  
смотрим более подробно. В двух словах:  
 •  
Каждая инструкция class создает новый объект класса.  
 •  
Каждый раз, когда вызывается класс, он создает новый объект экземпляра.  
 •  
Экземпляры автоматически связываются с классами, из которых они были   
созданы.  
 •  
Классы связаны со своими суперклассами, перечисленными в круглых   
скобках в заголовке инструкции class, – порядок следования в списке опре-  
деляет порядок расположения в дереве.  
Чтобы создать дерево, изображенное на рис. 25.1, например, мы могли бы ис-  
пользовать следующий программный код (здесь я опустил реализацию классов):  
class C2: ... # Создать объекты классов (овалы)  
class C3: ...  
class C1(C2, C3): ... # Связанные с суперклассами  
   
I1 = C1() # Создать объекты экземпляров (прямоугольники),  
I2 = C1() # связанные со своими классами  
Здесь мы построили дерево объектов классов, выполнив три инструкции class   
и сконструировав два объекта экземпляров, вызвав класс C1 дважды, как если   
бы это была обычная функция. Экземпляры помнят класс, из которого они   
были созданы, а класс C1 помнит о своих суперклассах.  
С технической точки зрения в этом примере используется то, что называется   
множественным наследованием, которое означает, что некий класс имеет бо-  
лее одного суперкласса над собой в дереве классов. В языке Python, если в ин-  
струкции class в круглых скобках перечислено более одного суперкласса (как   
в случае с классом C1 в данном примере), их порядок следования слева направо   
определяет порядок поиска атрибутов в суперклассах.  
Из-за особенностей поиска в дереве наследования большое значение имеет,   
к какому из объектов присоединяется тот или иной атрибут, – тем самым опре-

702   
Глава 25. ООП: общая картина   
деляется его область видимости. Атрибуты, присоединяемые к экземплярам,   
принадлежат только этим конкретным экземплярам, но атрибуты, присоеди-  
няемые к классам, совместно используются всеми подклассами и экземпляра-  
ми. Позднее мы подробно изучим программный код, выполняющий присоеди-  
нение атрибутов к этим объектам. Мы увидим, что:  
 •  
Атрибуты обычно присоединяются к классам с помощью инструкций при-  
сваивания внутри инструкции class, а не во вложенных инструкциях def,   
определяющих функции.  
 •  
Атрибуты обычно присоединяются к экземплярам с помощью присваива-  
ния значений специальному аргументу с именем self, передаваемому функ-  
циям внутри классов.  
Например, классы определяют поведение своих экземпляров с помощью функ-  
ций, создаваемых инструкциями def внутри инструкции class. Поскольку   
такие вложенные инструкции def выполняют присваивание именам внутри   
класса, они присоединяются к объектам классов в виде атрибутов и будут уна-  
следованы всеми экземплярами и подклассами:  
class C1(C2, C3): # Создать и связать класс C1  
 def setname(self, who): # Присвоить: C1.setname  
 self.name = who # self – либо I1, либо I2  
   
I1 = C1() # Создать два экземпляра  
I2 = C1()  
I1.setname(‘bob’) # Записать ‘bob’ в I1.name  
I2.setname(‘mel’) # Записать ‘mel’ в I2.name  
Print(I1.name) # Выведет ‘bob’  
Синтаксис инструкции def в этом контексте – совершенно обычный. С функци-  
ональной точки зрения, когда инструкция def появляется внутри инструкции   
class, как в этом примере, она обычно называется методом и автоматически   
принимает специальный первый аргумент с именем self, который содержит   
ссылку на обрабатываемый экземпляр.1  
Так как классы являются фабриками, способными производить множество эк-  
земпляров, их методы обычно используют этот, получаемый автоматически,   
аргумент self для получения или изменения значений атрибутов конкретно-  
го экземпляра, который обрабатывается методом. В предыдущем фрагменте   
программного кода имя self используется для сохранения имени служащего   
в конкретном экземпляре.  
Подобно простым переменным, атрибуты классов и экземпляров не объявля-  
ются заранее, а появляются, когда им впервые выполняется присваивание зна-  
чений. Когда метод присваивает значение атрибуту с помощью имени self, он   
тем самым создает атрибут экземпляра, находящегося в нижнем уровне дерева   
классов (то есть в одном из прямоугольников), потому что имя self автоматиче-  
ски ссылается на обрабатываемый экземпляр.  
Фактически благодаря тому, что все объекты в дереве классов – это всего лишь   
объекты пространств имен, мы можем получать или устанавливать любой   
1   
Если когда-нибудь вам приходилось использовать язык C++ или Java, вы без тру-  
да поймете, что в языке Python имя self – это то же, что указатель this, но в языке   
Python аргумент self всегда используется явно, чтобы сделать обращения к атрибу-  
там более очевидными.

ООП с высоты 30 000 футов   
703  
из их атрибутов, используя соответствующие имена. Например, выражение   
C1.setname является таким же допустимым, как и I1.setname, при условии, что   
имена C1 и I1 находятся в области видимости программного кода.  
В настоящий момент класс C1 не присоединяет атрибут name к экземплярам,   
пока не будет вызван метод setname. Фактически попытка обратиться к имени   
I1.name до вызова I1.setname приведет к появлению сообщения об ошибке, изве-  
щающего о неопределенном имени. Если в классе потребуется гарантировать,   
что атрибут, такой как name, всегда будет присутствовать в экземплярах, то та-  
кой атрибут должен создаваться на этапе создания класса, как показано ниже:  
class C1(C2, C3):  
 def \_\_init\_\_(self, who): # Создать имя при создании класса  
 self.name = who # Self – либо I1, либо I2  
   
I1 = C1(‘bob’) # Записать ‘bob’ в I1.name  
I2 = C1(‘mel’) # Записать ‘mel’ в I2.name  
Print(I1.name) # Выведет ‘bob’  
В этом случае интерпретатор Python автоматически будет вызывать метод   
с именем \_\_init\_\_ каждый раз при создании экземпляра класса. Новый экзем-  
пляр будет передаваться методу \_\_init\_\_ в виде первого аргумента self, а любые   
значения, перечисленные в круглых скобках при вызове класса, будут переда-  
ваться во втором и последующих за ним аргументах. В результате инициализа-  
ция экземпляров будет выполняться в момент их создания, без необходимости   
вызывать дополнительные методы.  
Метод \_\_init\_\_ известен как конструктор, так как он запускается на этапе   
конструирования экземпляра. Этот метод является типичным представителем   
большого класса методов, которые называются методами перегрузки операто-  
ров. Более подробно эти методы будут рассматриваться в последующих главах.   
Такие методы наследуются в дереве классов как обычно, а их имена начина-  
ются и заканчиваются двумя символами подчеркивания, чтобы подчеркнуть   
их особенное назначение. Интерпретатор Python вызывает их автоматически,   
когда экземпляры, поддерживающие их, участвуют в соответствующих опе-  
рациях, и они, главным образом, являются альтернативой вызовам простых   
методов. Кроме того, они являются необязательными: при их отсутствии соот-  
ветствующие операции экземплярами не поддерживаются.  
Например, чтобы реализовать пересечение множеств, класс может предусмо-  
треть реализацию метода intersect или перегрузить оператор &, описав логику   
его работы в методе с именем \_\_and\_\_. Поскольку использование операторов де-  
лает экземпляры более похожими на встроенные типы, это позволяет опреде-  
ленным классам обеспечивать непротиворечивый и естественный интерфейс   
и быть совместимыми с программным кодом, который предполагает выполне-  
ние операций над объектами встроенных типов.  
ООП – это многократное использование   
программного кода  
Вот, в основном, и все описание ООП в языке Python, за исключением неко-  
торых синтаксических особенностей. Конечно, в ООП присутствует не только   
наследование. Например, перегрузка операторов может применяться гораздо   
шире, чем описывалось до сих пор, – классы могут предоставлять собственные   
реализации таких операций, как доступ к элементам по их индексам, получе-

704   
Глава 25. ООП: общая картина   
ние значений атрибутов, вывод и многие другие. Но, вообще говоря, ООП реа-  
лизует поиск атрибутов в деревьях.  
Тогда зачем нам погружаться в тонкости создания деревьев объектов и выпол-  
нения поиска в них? Нужно накопить некоторый опыт, чтобы увидеть, как при   
грамотном использовании классы поддерживают возможность многократного   
использования программного кода способами, которые недоступны в других   
программных компонентах. Используя классы, мы программируем, адапти-  
руя написанное программное обеспечение, вместо того, чтобы изменять суще-  
ствующий программный код или писать новый код в каждом новом проекте.  
С фундаментальной точки зрения, классы – это действительно всего лишь па-  
кеты функций и других имен, которые во многом напоминают модули. Однако   
автоматический поиск атрибутов в дереве наследования, который мы получа-  
ем при использовании классов, обеспечивает возможности по адаптации про-  
граммного обеспечения более широкие, чем это возможно с помощью модулей   
и функций. Кроме того, классы представляют собой удобную структуру, обе-  
спечивающую компактное размещение выполняемого кода и переменных, что   
помогает в отладке.  
Например, методы – это обычные функции со специальным первым аргумен-  
том, поэтому мы можем подражать некоторым чертам их поведения, вручную   
передавая объекты для обработки обычным функциям. Однако участие мето-  
дов в наследовании классов позволяет нам естественным образом адаптировать   
существующее программное обеспечение, создавая новые подклассы, опреде-  
ляющие новые методы, вместо того, чтобы изменять существующий программ-  
ный код. Подобное невозможно в случае с модулями и функциями.  
В качестве примера предположим, что вас привлекли к реализации приложе-  
ния базы данных, где хранится информация о служащих. Как программист,   
использующий объектно-ориентированные особенности языка Python, вы мог-  
ли бы начать работу с реализации общего суперкласса, который определяет по-  
ведение, общее для всех категорий служащих в вашей организации:  
class Employee: # Общий суперкласс  
 def computeSalary(self): ... # Общее поведение  
 def giveRaise(self): ...  
 def promote(self): ...  
 def retire(self): ...  
Реализовав это общее поведение, можно специализировать его для каждой ка-  
тегории служащих, чтобы отразить отличия разных категорий от стандарта.   
То есть можно создать подклассы, которые изменяют лишь ту часть поведения,   
которая отличает их от типового представления служащего, – остальное по-  
ведение будет унаследовано от общего класса. Например, если зарплата инже-  
неров начисляется в соответствии с какими-то особыми правилами (то есть не   
по почасовому тарифу), в подклассе можно переопределить всего один метод:  
class Engineer(Employee): # Специализированный подкласс  
 def computeSalary(self): ... # Особенная реализация  
Поскольку эта версия computeSalary находится в дереве классов ниже, она бу-  
дет замещать (переопределять) общую версию метода в классе Employee. Затем   
можно создать экземпляры разновидностей классов служащих в соответствии   
с принадлежностью имеющихся служащих классам, чтобы обеспечить кор-  
ректное поведение:

ООП с высоты 30 000 футов   
705  
bob = Employee() # Поведение по умолчанию  
mel = Engineer() # Особые правила начисления зарплаты  
Обратите внимание, что существует возможность создавать экземпляры лю-  
бых классов в дереве, а не только тех, что находятся внизу, – класс, экземпляр   
которого создается, определяет уровень, откуда будет начинаться поиск атри-  
бутов. В перспективе эти два объекта экземпляров могли бы быть встроены   
в больший контейнерный объект (например, в список или в экземпляр другого   
класса), который представляет отдел или компанию, реализуя идею компози-  
ции, упомянутую в начале главы.  
Когда позднее вам потребуется узнать размер зарплаты этих служащих, их   
можно будет вычислить в соответствии с правилами классов, из которых были   
созданы объекты, благодаря поиску в дереве наследования:1  
company = [bob, mel] # Составной объект  
for emp in company:  
 print(emp.computeSalary()) # Вызвать версию метода для данного объекта  
Это еще одна разновидность полиморфизма – идеи, которая была представле-  
на в главе 4 и повторно рассматривалась в главе 16. Вспомните, полиморфизм   
означает, что смысл операции зависит от объекта, над которым она выполня-  
ется. Здесь метод computeSalary определяется в ходе поиска в каждом объекте   
в дереве наследования, прежде чем он будет вызван. В других приложениях   
полиморфизм может также использоваться для сокрытия (то есть для инкап-  
суляции) различий интерфейсов. Например, программа, которая обрабатывает   
потоки данных, может работать с объектами, имеющими методы ввода и выво-  
да, не заботясь о том, что эти методы делают в действительности:  
def processor(reader, converter, writer):  
 while 1:  
 data = reader.read()  
 if not data: break  
 data = converter(data)  
 writer.write(data)  
Передавая экземпляры классов с необходимыми интерфейсными методами   
read и write, специализированными под различные источники данных, мы мо-  
жем использовать одну и ту же функцию processor для работы с любыми ис-  
точниками данных, как уже существующими, так и с теми, что появятся в бу-  
дущем:  
class Reader:  
 def read(self): ... # Поведение и инструменты по умолчанию  
 def other(self): ...  
class FileReader(Reader):  
 def read(self): ... # Чтение из локального файла  
class SocketReader(Reader):  
1   
Обратите внимание, что список company из этого примера мог бы сохраняться в фай-  
ле, чтобы обеспечить постоянное хранение базы данных с информацией о служащих   
(с помощью модуля pickle, представленного в главе 9, когда мы знакомились с файла-  
ми). Кроме того, в состав Python входит модуль shelve, который мог бы позволить со-  
хранять экземпляры классов в файлах с доступом по ключу, – сторонняя разработка,   
система ZODB, позволяет реализовать то же самое, но имеет более качественную под-  
держку объектно-ориентированных баз данных для промышленного использования.

706   
Глава 25. ООП: общая картина   
 def read(self): ... # Чтение из сокета  
...  
processor(FileReader(...), Converter, FileWriter(...))  
processor(SocketReader(...), Converter, TapeWriter(...))  
processor(FtpReader(...), Converter, XmlWriter(...))  
Кроме того, благодаря тому, что внутренняя реализация этих методов read   
и write была разделена по типам источников данных, их можно изменять, не   
трогая программный код, подобный приведенному, который использует их.   
Фактически функция processor сама может быть классом, реализующим ло-  
гику работы функции преобразования converter, которую могут унаследовать   
подклассы, и позволяющим встраивать экземпляры, выполняющие чтение   
и запись, в соответствии с принципом композиции (далее в этой части книги   
будет показано, как это реализуется).  
Когда вы привыкнете программировать в этом стиле (адаптации программно-  
го обеспечения), то, начиная писать новую программу, обнаружите, что боль-  
шая часть вашей задачи уже реализована, и ваша задача в основном сводится   
к тому, чтобы подобрать уже имеющиеся суперклассы, которые реализуют по-  
ведение, требуемое вашей программе. Например, возможно, кто-то другой, для   
совершенно другой программы, уже написал классы Employee, Reader и Writer из   
данного примера. В этом случае вы сможете воспользоваться уже готовым про-  
граммным кодом «за так».  
На практике во многих прикладных областях вы можете получить или купить   
библиотеки суперклассов, известных как фреймворки, в которых реализованы   
наиболее часто встречающиеся задачи программирования на основе классов,   
готовые к использованию в ваших приложениях. Такие фреймворки могут   
предоставлять интерфейсы к базам данных, протоколы тестирования, сред-  
ства создания графического интерфейса и так далее. В среде такого фреймвор-  
ка вам часто будет достаточно создать свой подкласс, добавив в него один-два   
своих метода, а основная работа будет выполняться классами фреймворка, рас-  
положенными выше в дереве наследования. Программирование в мире ООП –   
это лишь вопрос сборки уже отлаженного программного кода и настройки его   
путем написания своих собственных подклассов.  
Безусловно, чтобы понять, как использовать классы для достижения такого   
объектно-ориентированного идеала, потребуется время. На практике ООП вле-  
чет за собой большой объем предварительного проектирования, на этапе которо-  
го осмысливаются преимущества, получаемые от использования классов� с этой   
целью программисты начали составлять список наиболее часто встречающих-  
ся решений в ООП, известных как шаблоны  проектирования,  – помогающих   
решать проблемы проектирования. При этом объектно-ориентированный про-  
граммный код на языке настолько прост, что он сам по себе не будет препятстви-  
ем в освоении ООП. Чтобы убедиться в этом, вам следует перейти к главе 26.  
В заключение  
В этой главе мы в общих чертах рассмотрели ООП и классы, получив пред-  
ставление о них, прежде чем углубиться в детали синтаксиса. Как мы узнали,   
основу ООП составляет поиск атрибутов в дереве связанных объектов� мы на-  
зываем его поиском в дереве наследования. Объекты в нижней части дерева на-  
следуют атрибуты объектов, расположенных в дереве выше, – эта особенность   
позволяет создавать программы за счет адаптации программного кода, а не за

Закрепление пройденного   
707  
счет его изменения или создания нового кода. При грамотном использовании   
эта модель программирования может привести к существенному сокращению   
времени, необходимого на разработку.  
В следующей главе мы приступим к рассмотрению подробностей, стоящих за   
общей картиной, представленной здесь. Мы начнем углубляться в изучение   
классов, однако не стоит забывать, что объектно-ориентированная модель   
в языке Python очень проста, – как я уже заметил, она, по сути, сводится к по-  
иску атрибутов в деревьях объектов. Прежде чем двинуться дальше, ответьте   
на контрольные вопросы, чтобы освежить в памяти все то, о чем рассказыва-  
лось здесь.  
Закрепление пройденного  
Контрольные вопросы  
1. Каково основное назначение ООП в языке Python?  
2. Где выполняется поиск унаследованных атрибутов?  
3. В чем разница между объектом класса и объектом экземпляра?  
4. В чем состоит особенность первого аргумента в методах классов?  
5. Для чего служит метод \_\_init\_\_?  
6. Как создать экземпляр класса?  
7. Как создать класс?  
8. Как определяются суперклассы для класса?  
Ответы  
1. Основное назначение ООП состоит в том, чтобы обеспечить многократное   
использование программного кода, – программный код разлагается на со-  
ставляющие, чтобы снизить его избыточность, и при создании новых про-  
грамм выполняется адаптация имеющегося программного кода, а не изме-  
нение существующего или создание нового кода.  
2. Поиск унаследованных атрибутов выполняется сначала в объекте экзем-  
пляра, затем в классе, из которого был создан экземпляр, затем во всех су-  
перклассах, в направлении снизу вверх и слева направо в дереве объектов   
(по умолчанию). Как только будет найдено первое вхождение атрибута, по-  
иск прекращается. Поскольку результатом поиска является самая нижняя   
версия имени из имеющихся в дереве, иерархия классов естественным об-  
разом поддерживает возможность адаптации за счет создания подклассов.  
3. И классы, и экземпляры – это пространства имен (пакеты переменных, ко-  
торые играют роль атрибутов). Главное различие между ними состоит в том,   
что классы представляют собой своего рода фабрики, позволяющие произ-  
водить множество экземпляров. Кроме того, классы поддерживают методы   
перегрузки операторов, которые наследуются экземплярами, а функции,   
вложенные в классы, интерпретируются как специальные методы обработ-  
ки экземпляров.  
4. Первый аргумент в методах классов играет особую роль, так как через него   
всегда передается ссылка на объект экземпляра, который подразумевается

708   
Глава 25. ООП: общая картина   
при вызове метода. Согласно общепринятым соглашениям он обычно на-  
зывается self. Так как по умолчанию методы всегда принимают этот под-  
разумеваемый объект, мы говорим, что они «объектно-ориентированные»,   
то есть предназначенные для обработки или изменения объектов.  
5. Если метод \_\_init\_\_ присутствует в классе или наследуется им, интерпре-  
татор автоматически будет вызывать его при создании каждого экземпляра   
этого класса. Этот метод также называют конструктором – он неявно полу-  
чает новый экземпляр, а также все аргументы, которые были указаны при   
вызове имени класса. Кроме того, он является типичным представителем   
методов перегрузки операторов. В отсутствие метода \_\_init\_\_ экземпляры   
начинают свое существование как пустые пространства имен.  
6. Экземпляр класса создается вызовом имени класса, как если бы это была   
функция, – любые аргументы в вызове будут переданы конструктору \_\_init\_\_   
как второй и следующие аргументы. Новый экземпляр запоминает, из како-  
го класса он был создан, благодаря чему обеспечивается наследование.  
7. Класс создается с помощью инструкции class, так же как определения   
функций. Эти инструкции обычно выполняются во время импортирования   
файла вмещающего модуля (подробнее об этом рассказывается в следую-  
щей главе).  
8. Суперклассы для заданного класса определяются в виде списка в круглых   
скобках в инструкции class после имени нового класса. Порядок следова-  
ния суперклассов в списке определяет порядок поиска в дереве наследова-  
ния.

Глава 26.  
   
Основы программирования классов  
Теперь, когда мы поговорили об ООП в общем, настало время посмотреть, как   
это выглядит в фактическом программном коде. В этой и следующей главах   
подробно будет рассказываться о синтаксисе, который составляет основу моде-  
ли классов в языке Python.   
Если ранее вам не приходилось сталкиваться с ООП, будет сложно сразу усво-  
ить все сведения о классах. Чтобы облегчить освоение программирования   
классов, мы начнем наше детальное исследование ООП с того, что рассмотрим   
в этой главе несколько простых классов в действии. Классы в языке Python   
в самой простой их форме легко понять, а детали будут рассмотрены подробнее   
в следующих главах.  
Классы обладают тремя основными отличительными особенностями. На са-  
мом простом уровне они представляют собой всего лишь пространства имен, во   
многом похожие на модули, которые мы изучали в пятой части книги. Но, в от-  
личие от модулей, классы также поддерживают создание множества объектов,   
реализуют наследуемое пространство имен и перегрузку операторов. Начнем   
наше путешествие с инструкции class и исследуем каждую из этих трех осо-  
бенностей.  
Классы генерируют   
множество экземпляров объектов  
Чтобы понять, каким образом обеспечивается возможность создания множе-  
ства объектов, для начала нужно понять, что в объектно-ориентированной мо-  
дели языка Python существует две разновидности объектов: объекты классов   
и объекты экземпляров. Объекты классов реализуют поведение по умолчанию   
и играют роль фабрик по производству объектов экземпляров. Объекты эк-  
земпляров – это настоящие объекты, обрабатываемые программой, – каждый   
представляет собой самостоятельное пространство имен, но наследует имена   
(то есть автоматически имеет доступ к ним) класса, из которого он был создан.   
Объекты классов создаются инструкциями, а объекты экземпляров – вызова-  
ми. Каждый раз, когда вы вызываете класс, вы получаете новый экземпляр   
этого класса.

710   
Глава 26. Основы программирования классов   
Эта объектная концепция существенно отличается от любых других программ-  
ных конструкций, которые мы видели до сих пор в этой книге. В действитель-  
ности, классы – это фабрики, способные производить множество экземпляров.   
В противоположность этому каждый модуль в программе может присутство-  
вать в единственном экземпляре. (Фактически это одна из причин, почему не-  
обходимо использовать функцию imp.reload для обновления объекта модуля,   
чтобы отразить внесенные в модуль изменения.)  
Ниже приводится краткий обзор основных особенностей ООП в языке Python.   
Как вы увидите, классы в языке Python сочетают в себе черты, напоминающие   
функции и модули, но они совершенно не похожи на классы в других языках   
программирования.  
Объекты классов реализуют поведение по умолчанию  
Когда выполняется инструкция class, она создает объект класса. Ниже при-  
водятся несколько основных отличительных характеристик классов в языке   
Python.  
 •  
Инструкция class создает объект класса и присваивает ему имя. Так же   
как и инструкция def, инструкция class является выполняемой инструкци-  
ей. Когда она выполняется, она создает новый объект класса и присваивает   
его имени, указанному в заголовке инструкции class. Кроме того, как и ин-  
струкции def, инструкции class обычно выполняются при первом импорти-  
ровании содержащих их файлов.  
 •  
Операции присваивания внутри инструкции class создают атрибуты клас-  
са. Так же как и в модулях, инструкции присваивания на верхнем уровне   
внутри инструкции class (не вложенные в инструкции def) создают атрибу-  
ты объекта класса. С технической точки зрения инструкция class преобра-  
зует свою область видимости в пространство имен атрибутов объекта клас-  
са, так же, как глобальная область видимости модуля преобразуется в его   
пространство имен. После выполнения инструкции class атрибуты класса   
становятся доступны по их составным (полным) именам: object.name.  
 •  
Атрибуты класса описывают состояние объекта и его поведение. Атрибуты   
объекта класса хранят информацию о состоянии и описывают поведение,   
которым обладают все экземпляры класса, – инструкции def, вложенные   
в инструкцию class, создают методы, которые обрабатывают экземпляры.  
Объекты экземпляров – это конкретные элементы  
Когда вызывается объект класса, возвращается объект экземпляра. Ниже при-  
водятся несколько отличительных характеристик экземпляров классов:  
 •  
Вызов объекта класса как функции создает новый объект экземпляра.   
Всякий раз, когда вызывается класс, создается и возвращается новый объ-  
ект экземпляра. Экземпляры представляют собой конкретные элементы   
данных в вашей программе.  
 •  
Каждый объект экземпляра наследует атрибуты класса и приобретает   
свое собственное пространство имен. Объекты экземпляров создаются из   
классов и представляют собой новые пространства имен� они первоначаль-  
но пустые, но наследуют атрибуты классов, из которых были созданы.

Классы генерируют множество экземпляров объектов   
711  
 •  
Операции присваивания значений атрибутам через ссылку self в методах   
создают атрибуты в каждом отдельном экземпляре. Методы класса по-  
лучают в первом аргументе (с именем self в соответствии с соглашениями)   
ссылку на обрабатываемый объект экземпляра – присваивание атрибутам   
через ссылку self создает или изменяет данные экземпляра, а не класса.  
Первый пример  
Обратимся к первому примеру, демонстрирующему работу этих идей на прак-  
тике. Для начала определим класс с именем FirstClass, выполнив инструкцию   
class в интерактивной оболочке интерпретатора Python:  
>>> class FirstClass: # Определяет объект класса  
... def setdata(self, value): # Определяет метод класса  
... self.data = value # self – это экземпляр  
... def display(self):  
... print(self.data) # self.data: данные экземпляров  
...  
Здесь мы работаем в интерактивной оболочке, но обычно такие инструкции   
выполняются во время импортирования вмещающего файла модуля. Подобно   
функциям, создаваемым с помощью инструкции def, классы не существуют,   
пока интерпретатор Python не достигнет этих инструкций и не выполнит их.  
Как и все составные инструкции, инструкция class начинается со строки за-  
головка, содержащей имя класса, за которой следует тело класса из одной или   
более вложенных инструкций (обычно) с отступами. Здесь вложенными ин-  
струкциями являются инструкции def – они определяют функции, реализую-  
щие поведение класса.   
Как мы уже знаем, инструкции def в действительности являются операциями   
присваивания – в данном случае они присваивают объекты функций именам   
setdata и display в области видимости инструкции class и тем самым создают   
атрибуты, присоединяемые к классу: FirstClass.setdata и FirstClass.display.   
Фактически любое имя, которому присваивается значение на верхнем уровне   
во вложенном блоке класса, становится атрибутом этого класса.  
Функции внутри классов обычно называются методами. Это обычные ин-  
струкции def, и для них верно все, что мы уже знаем о функциях (они могут   
иметь аргументы со значениями по умолчанию, возвращать значения и так   
далее). Но в первом аргументе методы автоматически получают подразумева-  
емый объект экземпляра – контекст вызова метода. Нам необходимо создать   
пару экземпляров, чтобы понять, как это делается:  
>>> x = FirstClass() # Создаются два экземпляра  
>>> y = FirstClass() # Каждый является отдельным пространством имен  
Вызывая класс таким способом (обратите внимание на круглые скобки), мы   
создаем объекты экземпляров, которые являются всего лишь пространствами   
имен и обладают доступом к атрибутам класса. Собственно говоря, к настоя-  
щему моменту у нас имеется три объекта – два экземпляра и один класс. В дей-  
ствительности у нас имеется три связанных пространства имен, как показа-  
но на рис. 26.1. В терминах ООП мы говорим, что объект x «наследует» класс   
FirstClass, как и y.

712   
Глава 26. Основы программирования классов   
наследует  
FirstClass  
наследует  
– setdata  
– display  
Y  
– data  
X  
– data  
Рис. 26.1. Экземпляры и классы – это связанные между собой объекты про-  
странств имен в дереве наследования классов, внутри которого выполняет-  
ся поиск. Здесь атрибут «data» обнаруживается в экземплярах, а «setdata»   
и «display» – в классе, расположенном выше их  
Изначально оба экземпляра пустые, но они связаны с классом, из которо-  
го были созданы. Если через имя экземпляра обратиться к атрибуту объекта   
класса, то в результате поиска по дереву наследования интерпретатор вернет   
значение атрибута класса (при условии, что в экземпляре отсутствует одно-  
именный атрибут):  
>>> x.setdata(“King Arthur”) # Вызов метода: self – это x  
>>> y.setdata(3.14159) # Эквивалентно: FirstClass.setdata(y, 3.14159)  
Ни x, ни y не имеют собственного атрибута setdata, поэтому, чтобы отыскать   
его, интерпретатор следует по ссылке от экземпляра к классу. В этом заключа-  
ется суть наследования в языке Python: механизм наследования привлекается   
в момент разрешения имени атрибута, и вся его работа заключается лишь в по-  
иске имен в связанных объектах (например, следуя по ссылкам «наследует»,   
как показано на рис. 26.1).  
В функции setdata внутри класса FirstClass значение аргумента записывается   
в self.data. Имя self внутри метода – имя самого первого аргумента, в соответ-  
ствии с общепринятыми соглашениями, – автоматически ссылается на обра-  
батываемый экземпляр (x или y), поэтому операция присваивания сохраняет   
значения в пространстве имен экземпляра, а не класса (так создаются имена   
data, показанные на рис. 26.1).  
Поскольку классы способны генерировать множество экземпляров, методы   
должны использовать аргумент self, чтобы получить доступ к обрабатываемо-  
му экземпляру. Когда мы вызываем метод класса display, чтобы вывести зна-  
чения атрибутов self.data, мы видим, что для каждого экземпляра они разные�   
с другой стороны, имя display само по себе одинаковое в x и y, так как оно при-  
шло (унаследовано) из класса:  
>>> x.display() # В каждом экземпляре свое значение self.data  
King Arthur  
>>> y.display()  
3.14159  
Обратите внимание, что в атрибутах data экземпляров мы сохранили объекты   
различных типов (строку и вещественное число). Как и повсюду в языке Py-  
Py-  
thon, атрибуты экземпляров (иногда называются членами) никак не объявля-  
ются – они появляются, как только будет выполнена первая операция присва-

Классы адаптируются посредством наследования   
713  
ивания, точно так же, как и в случае с переменными. Фактически если вызвать   
метод display до вызова метода setdata, будет получено сообщение об ошибке   
обращения к неопределенному имени – атрибут с именем data не существует   
в памяти, пока ему не будет присвоено какое-либо значение в методе setdata.  
Еще один способ, дающий возможность оценить, насколько динамична эта   
модель, позволяет изменять атрибуты экземпляров в самом классе, выполняя   
присваивание как с помощью аргумента self внутри методов, так и за предела-  
ми класса, когда экземпляр явно участвует в операции присваивания:  
>>> x.data = “New value” # Можно получать/записывать значения атрибутов  
>>> x.display() # И за пределами класса тоже  
New value  
Хотя это и нечасто применяется, тем не менее существует возможность созда-  
ния новых атрибутов в пространстве имен экземпляра, присваивая значения   
именам за пределами методов класса:  
>>> x.anothername = “spam” # Здесь также можно создавать новые атрибуты  
Эта операция присоединит новый атрибут с именем anothername, который за-  
тем сможет использоваться любыми методами класса в объекте экземпляра x.   
Обычно классы создают все атрибуты экземпляров за счет присваивания значе-  
ний аргументу self, но это не обязательно – программы могут получать, изме-  
нять или создавать атрибуты в любых объектах, к которым они имеют доступ.  
Классы адаптируются   
посредством наследования  
Помимо роли фабрик по созданию объектов экземпляров, классы также по-  
зволяют нам вносить изменения за счет введения новых компонентов (которые   
называются подклассами), а не за счет изменения существующего программ-  
ного кода. Объекты экземпляров, созданные из класса, наследуют атрибуты   
класса. В языке Python классы также могут наследовать другие классы, что   
открывает дверь к созданию иерархий классов, поведение которых специали-  
зируется за счет переопределения более обобщенных атрибутов, находящихся   
выше в дереве иерархии, в подклассах, находящихся ниже в иерархии. В ре-  
зультате, чем ниже мы опускаемся в дереве иерархии, тем более специализи-  
рованными становятся классы. Здесь также нет никакого сходства с модуля-  
ми: их атрибуты находятся в едином плоском пространстве имен, которое не   
поддается адаптации.  
В языке Python экземпляры наследуют классы, а классы наследуют супер-  
Python экземпляры наследуют классы, а классы наследуют супер-  
 экземпляры наследуют классы, а классы наследуют супер-  
классы. Ниже приводятся основные идеи, лежащие в основе механизма насле-  
дования атрибутов:  
 •  
Суперклассы перечисляются в круглых скобках в заголовке инструкции   
class. Чтобы унаследовать атрибуты другого класса, достаточно указать   
этот класс в круглых скобках в заголовке инструкции class. Наследующий   
класс называется подклассом, а наследуемый класс называется его супер-  
классом.  
 •  
Классы наследуют атрибуты своих суперклассов. Как экземпляры насле-  
дуют имена атрибутов, определяемых их классами, так же и классы насле-

714   
Глава 26. Основы программирования классов   
дуют все имена атрибутов, определяемые в их суперклассах, – интерпре-  
татор автоматически отыскивает их, когда к ним выполняется обращение,   
если эти атрибуты отсутствуют в подклассах.  
 •  
Экземпляры наследуют атрибуты всех доступных классов. Каждый эк-  
земпляр наследует имена из своего класса, а также из всех его суперклас-  
сов. Во время поиска имен интерпретатор проверяет сначала экземпляр, по-  
том его класс, а потом все суперклассы.  
 •  
Каждое обращение object.attribute вызывает новый независимый поиск.   
Интерпретатор выполняет отдельную процедуру поиска в дереве классов   
для каждого атрибута, который ему встречается в выражении запроса.   
Сюда входят ссылки на экземпляры и классы из инструкции class (напри-  
мер, X.attr), а также ссылки на атрибуты аргумента экземпляра self в мето-  
дах класса. Каждое выражение self.attr в методе вызывает поиск attr в self   
и выше.  
 •  
Изменения в подклассах не затрагивают суперклассы. Замещение имен   
суперкласса в подклассах ниже в иерархии (в дереве классов) изменяет под-  
классы и тем самым изменяет унаследованное поведение.  
Результат и главная цель такой процедуры поиска состоит в том, что классы   
обеспечивают разложение на отдельные операции и адаптацию программного   
кода лучше, чем это могут сделать другие компоненты языка, которые мы рас-  
сматривали ранее. С одной стороны, классы позволяют минимизировать избы-  
точность программного кода (и тем самым снизить стоимость обслуживания)   
за счет создания единой и общей реализации операций, а с другой – они обе-  
спечивают возможность адаптировать уже существующий программный код   
вместо того, чтобы изменять его или писать заново.  
Второй пример  
Проиллюстрируем роль механизма на втором примере, который основан на   
предыдущем. Для начала определим новый класс, SecondClass, который насле-  
дует все имена из класса FirstClass и добавляет свои собственные:  
>>> class SecondClass(FirstClass): # Наследует setdata  
... def display(self): # Изменяет display  
... print(‘Current value = “%s”’ % self.data)  
...  
Класс SecondClass определяет метод display, осуществляющий вывод в другом   
формате. Определяя атрибут с тем же именем, что и атрибут в классе FirstClass,   
класс SecondClass замещает атрибут display своего суперкласса.  
Вспомните, что поиск в дереве наследования выполняется снизу вверх – от эк-  
земпляров к классам и далее к суперклассам и останавливается, как только   
будет найдено первое вхождение искомого имени атрибута. В данном случае   
имя display в классе SecondClass будет найдено раньше, чем это же имя в клас-  
се FirstClass. В этом случае мы говорим, что класс SecondClass переопределяет   
метод display класса FirstClass. Иногда такая замена атрибутов за счет их пере-  
определения ниже в дереве классов называется перегрузкой.  
Главный результат здесь состоит в том, что класс SecondClass специализирует   
класс FirstClass, изменяя поведение метода display. С другой стороны, класс   
SecondClass (и все экземпляры, созданные из него) по-прежнему наследует из

Классы адаптируются посредством наследования   
715  
класса FirstClass метод setdata. Создадим экземпляр, чтобы продемонстриро-  
вать это:  
>>> z = SecondClass()  
>>> z.setdata(42) # Найдет setdata в FirstClass  
>>> z.display() # Найдет переопределенный метод в SecondClass  
Current value = “42”  
Как и прежде, мы создали объект экземпляра, вызвав класс SecondClass. При   
обращении к setdata все так же вызывается версия метода из FirstClass, но при   
обращении к атрибуту display вызывается версия метода из SecondClass, кото-  
рая выводит измененный текст сообщения. Схема пространств имен, вовлечен-  
ных в действие, изображена на рис. 26.2.  
наследует  
Z (экземпляр)  
наследует  
SecondClass  
FirstClass  
– data  
– display  
– setdata  
– display  
Z.data  
Z.display  
Z.setdata  
Рис. 26.2. Специализация путем переопределения унаследованных имен   
посредством повторного их определения ниже в дереве классов. Здесь класс   
SecondClass переопределяет и тем самым адаптирует метод «display» для   
экземпляров этого класса  
Здесь очень важно заметить следующее: специализация, выполненная в клас-  
се SecondClass, находится полностью за пределами класса FirstClass. Как след-  
ствие, она не оказывает влияния на уже созданные или создаваемые впослед-  
ствии объекты класса FirstClass, такие как x в предыдущем примере:  
>>> x.display() # x по-прежнему экземпляр FirstClass (старое сообщение)  
New value  
Вместо того, чтобы изменять класс FirstClass, мы адаптировали его. Безу-  
словно, это достаточно искусственный пример, но он демонстрирует, как на-  
следование позволяет вносить изменения с помощью внешних компонентов (то   
есть с помощью подклассов). Часто классы поддерживают возможность расши-  
рения и повторного использования гораздо лучше, чем функции или модули.  
Классы – это атрибуты в модулях  
Прежде чем двинуться дальше, вспомним, что в именах классах нет ничего   
необычного. Это всего лишь переменные, которым присваиваются объекты во   
время выполнения инструкций class, а ссылки на объекты можно получить   
с помощью обычных выражений. Например, если бы определение класса First-  
Class находилось в файле модуля, а не было введено в интерактивной оболочке,   
мы могли бы импортировать этот модуль и использовать имя в строке заголов-  
ка инструкции class:

716   
Глава 26. Основы программирования классов   
from modulename import FirstClass # Скопировать имя в мою область видимости  
class SecondClass(FirstClass): # Использовать имя класса непосредственно  
 def display(self): ...  
Или эквивалентный вариант:  
import modulename # Доступ ко всему модулю целиком  
class SecondClass(modulename.FirstClass): # Указать полное имя  
 def display(self): ...  
Имена классов, как и все остальное, всегда находятся в модулях, и потому при   
их употреблении необходимо следовать правилам, которые мы рассматрива-  
ли в пятой части книги. Например, в одном файле модуля могут находиться   
определения сразу нескольких классов – подобно другим инструкциям в моду-  
ле, инструкции class выполняются в ходе операции импортирования и опреде-  
ляют имена, которые в свою очередь становятся атрибутами модуля. Вообще,   
любой модуль может содержать самые произвольные сочетания из любого чис-  
ла переменных, функций и классов, и все эти имена будут вести себя в модуле   
одинаково. Это демонстрирует файл food.py:  
# food.py  
var = 1 # food.var  
def func(): # food.func  
 ...  
class spam: # food.spam  
 ...  
class ham: # food.ham  
 ...  
class eggs: # food.eggs  
 ...  
Это правило остается справедливым, даже если модуль и класс имеют одина-  
ковые имена. Например, пусть имеется следующий файл person.py:  
class person:  
 ...  
Чтобы получить доступ к классу, нам необходимо обратиться к модулю, как   
обычно:  
import person # Импортировать модуль  
x = person.person() # Класс внутри модуля  
Хотя этот способ может показаться избыточным, он совершенно необходим:   
имя person.person ссылается на класс person внутри модуля person. Если исполь-  
зовать просто имя person, мы обратимся к модулю, а не к классу� кроме случая,   
когда используется инструкция from:  
from person import person # Получить класс из модуля  
x = person() # Использовать имя класса  
Как и любые другие переменные, мы не увидим класс в файле модуля, пока не   
импортируем его или как-то иначе не извлечем класс из вмещающего файла.   
Если вам это кажется сложным, то не используйте одинаковые имена для мо-  
дулей и классов в них. Согласно общепринятым соглашениям, имена классов   
в языке Python должны начинаться с заглавной буквы, чтобы обеспечить ви-  
Python должны начинаться с заглавной буквы, чтобы обеспечить ви-  
 должны начинаться с заглавной буквы, чтобы обеспечить ви-  
зуальное отличие:

Классы могут переопределять операторы языка Python   
717  
import person # Имена модулей начинаются с прописных букв  
x = person.Person() # Имена классов – с заглавных   
Кроме того, имейте в виду, несмотря на то, что классы и модули являются   
пространствами имен для подключения атрибутов, они представляют собой   
совершенно разные структуры: модуль является отражением целого файла,   
а класс – это лишь инструкция внутри файла. Подробнее об этих различиях   
мы поговорим позднее в этой части книги.  
Классы могут переопределять   
операторы языка Python  
Теперь давайте рассмотрим третье основное отличие классов и модулей: пере-  
грузку операторов. Говоря простым языком, перегрузка операторов позволяет   
объектам, созданным из классов, перехватывать и участвовать в операциях,   
которые применяются к встроенным типам: сложение, получение среза, вы-  
вод, квалификация имен и так далее. По большей части это автоматический   
механизм: при выполнении выражений и других встроенных операций интер-  
претатор передает управление реализации классов. В модулях нет ничего по-  
добного: модули могут реализовать функции, но не операторы выражений.  
Мы можем полностью реализовать поведение класса в виде методов, однако пе-  
регрузка операторов позволяет объектам теснее интегрироваться в объектную   
модель языка Python. Кроме того, перегрузка операторов помогает нашим объ-  
ектам действовать так же, как действуют встроенные объекты, потому что она   
позволяет создавать менее противоречивые и более простые в изучении интер-  
фейсы объектов и обеспечивает возможность обрабатывать объекты, создан-  
ные из классов, программным кодом, который предполагает взаимодействие   
со встроенными типами. Ниже перечислены основные идеи, лежащие в основе   
механизма перегрузки операторов:  
 •  
Имена методов, начинающиеся и заканчивающиеся двумя символами   
подчеркивания (\_\_X\_\_), имеют специальное назначение. Перегрузка опе-  
раторов в языке Python реализуется за счет создания методов со специаль-  
ными именами для перехватывания операций. Язык Python определяет   
фиксированные и неизменяемые имена методов для каждой из операций.  
 •  
Такие методы вызываются автоматически, когда экземпляр участвует   
во встроенных операциях. Например, если объект экземпляра наследует   
метод \_\_add\_\_, этот метод будет вызываться всякий раз, когда объект будет   
появляться в операции сложения (+). Возвращаемое значение метода стано-  
вится результатом соответствующей операции.  
 •  
Классы могут переопределять большинство встроенных операторов. Су-  
ществует множество специальных имен методов для реализации перегруз-  
ки почти всех операторов, доступных для встроенных типов. Сюда входят   
операторы выражений, а также такие базовые операции, как вывод и соз-  
дание объекта.  
 •  
В методах перегрузки операторов не существует аргументов со значения-  
ми по умолчанию, и ни один из таких методов не является обязательным   
для реализации. Если класс не определяет и не наследует методы перегруз-  
ки операторов, это означает лишь, что экземпляры класса не поддерживают

718   
Глава 26. Основы программирования классов   
эти операции. Например, если отсутствует метод \_\_add\_\_, попытка выпол-  
нить операцию сложения + будет приводить к возбуждению исключения.  
 •  
Операторы позволяют интегрировать классы в объектную модель языка   
Python. Благодаря перегрузке операторов, объекты, реализованные на базе   
классов, действуют подобно встроенным типам и тем самым обеспечивают   
непротиворечивые и совместимые интерфейсы.  
Перегрузка операторов является необязательной функциональной особенно-  
стью – она используется в основном специалистами, создающими инструмен-  
тальные средства для других программистов, а не разработчиками приклад-  
ных программ. И честно говоря, вам не стоит использовать ее только потому,   
что это «круто». Если не требуется, чтобы класс имитировал поведение встро-  
енных типов, лучше ограничиться использованием простых методов. Напри-  
мер, зачем приложению, работающему с базой данных служащих, поддерж-  
ка таких операторов, как \* и +? Методы с обычными именами, такими как   
giveRaise и promote, обычно более уместны.  
Вследствие этого мы не будем далее углубляться в подробности реализации   
методов перегрузки каждого оператора, доступного в языке Python. Однако   
имеется один метод перегрузки оператора, который можно встретить практи-  
чески в любом классе: метод \_\_init\_\_, который известен как конструктор и ис-  
пользуется для инициализации состояния объектов. Методу \_\_init\_\_ следует   
уделить особое внимание, потому что он, наряду с аргументом self, является   
одним из ключей к пониманию ООП в языке Python.  
Третий пример  
На этот раз мы определим подкласс, производный от SecondClass и реализую-  
щий три специальных метода, которые будут вызываться интерпретатором ав-  
томатически:   
 •  
Метод \_\_init\_\_ вызывается, когда создается новый объект экземпляра (ар-  
гумент self представляет новый объект ThirdClass).1  
 •  
Метод \_\_add\_\_ вызывается, когда экземпляр ThirdClass участвует в опера-  
ции +.   
 •  
Метод \_\_str\_\_ вызывается при выводе объекта (точнее, когда он преобразу-  
ется в строку для вывода вызовом встроенной функции str или ее эквива-  
лентом внутри интерпретатора).  
Кроме того, новый подкласс определяет метод с обычным именем mul, который   
изменяет сам объект в памяти. Ниже приводится определение нового подкласса:  
>>> class ThirdClass(SecondClass): # Наследует SecondClass  
... def \_\_init\_\_(self, value): # Вызывается из ThirdClass(value)  
... self.data = value  
... def \_\_add\_\_(self, other): # Для выражения “self + other”  
... return ThirdClass(self.data + other)  
... def \_\_str\_\_(self): # Вызывается из print(self), str()  
... return ‘[ThirdClass: %s]’ % self.data  
1   
Не путайте с файлом \_\_init\_\_.py в пакетах модулей! За дополнительными подробно-  
стями обращайтесь к главе 23.

Классы могут переопределять операторы языка Python   
719  
... def mul(self, other): # Изменяет сам объект: обычный метод  
... self.data \*= other  
...  
>>> a = ThirdClass(“abc”) # Вызывается новый метод \_\_init\_\_  
>>> a.display() # Унаследованный метод  
Current value = “abc”  
>>> print(a) # \_\_str\_\_: возвращает строку  
[ThirdClass: abc]  
   
>>> b = a + ‘xyz’ # Новый \_\_add\_\_: создается новый экземпляр  
>>> b.display()  
Current value = “abcxyz”  
>>> print(b) # \_\_str\_\_: возвращает строку  
[ThirdClass: abcxyz]  
   
>>> a.mul(3) # mul: изменяется сам экземпляр  
>>> print(a)   
[ThirdClass: abcabcabc]  
Класс ThirdClass «наследует» класс SecondClass, поэтому его экземпляры насле-  
дуют метод display от SecondClass. Но теперь при создании экземпляра класса   
ThirdClass ему передается дополнительный аргумент (например, «abc»). Это   
значение передается конструктору \_\_init\_\_ в аргументе value, где присваивает-  
ся атрибуту self.data. В результате при создании экземпляра класса ThirdClass   
значение атрибута data устанавливается автоматически, благодаря чему отпа-  
дает необходимость вызывать метод setdata после создания экземпляра.  
Далее, объекты ThirdClass могут участвовать в операциях + и в вызовах функ-  
ции print. При выполнении операции сложения объект экземпляра слева от   
оператора передается методу \_\_add\_\_ в виде аргумента self, а значение справа –   
в виде аргумента other, как показано на рис. 26.3. Независимо от того, что вер-  
нет метод \_\_add\_\_, это значение будет интерпретироваться, как результат опе-  
рации сложения. Когда объект участвует в вызове функции print, интерпрета-  
тор вызывает метод \_\_str\_\_ объекта и передает ему сам объект – любая строка,   
которую вернет этот метод, будет расцениваться, как строковое представление   
объекта для вывода. Переопределив метод \_\_str\_\_, мы получаем возможность   
использовать обычную функцию print для отображения объектов этого класса,   
вместо того чтобы вызывать метод display.  
a + 3  
\_add\_(self, other)  
Рис. 26.3. Механизм перегрузки отображает операторы выражений и другие   
встроенные операции, выполняемые над экземплярами классов, на специ-  
альные имена методов в классе. Эти специальные методы являются необяза-  
тельными и могут наследоваться, как обычные методы. Здесь использование   
оператора «+» приводит к вызову метода \_\_add\_\_

720   
Глава 26. Основы программирования классов   
Методы со специальными именами, такими как \_\_init\_\_, \_\_add\_\_ и \_\_str\_\_, на-  
следуются подклассами и экземплярами, как любые другие имена, которым   
выполняется присваивание в инструкции class. Если методы отсутствуют   
в классе, интерпретатор, как обычно, продолжит их поиск в суперклассах.   
Кроме того, имена методов перегрузки операторов не являются встроенными   
именами или зарезервированными словами – это обычные атрибуты, которые   
отыскиваются интерпретатором в случае появления объектов в различных   
контекстах. Как правило, интерпретатор вызывает их автоматически, но они   
могут вызываться и вашим программным кодом – метод \_\_init\_\_, например,   
часто вызывается вручную, с целью запустить конструктор суперкласса (под-  
робнее об этом мы поговорим позднее).  
Обратите внимание, что метод \_\_add\_\_ создает и возвращает новый объект эк-  
земпляра этого класса (вызывая ThirdClass, которому передается значение ре-  
зультата), а метод mul изменяет текущий объект экземпляра (выполняя при-  
сваивание атрибуту аргумента self). Мы могли бы перегрузить оператор \*,   
вместо того, чтобы создавать этот метод, но такое поведение отличается от по-  
ведения встроенных типов, таких как числа и строки, которые всегда создают   
новые объекты при выполнении оператора \*. Поскольку перегрузка операто-  
ров – это в действительности всего лишь механизм отображения выражений   
на методы, вы можете интерпретировать операторы в своих объектах классов,   
как вам будет угодно.  
Когда следует использовать перегрузку операторов?  
Создавая свои классы, вы можете выбирать – использовать перегрузку опе-  
раторов или нет. Выбор зависит от того, насколько близко ваш класс должен   
имитировать поведение встроенных типов. Как упоминалось выше, если метод   
перегрузки оператора отсутствует в определении класса и не наследуется из   
суперкласса, соответствующая операция не будет поддерживаться экземпля-  
рами этого класса – если попытаться выполнить такую операцию, интерпре-  
татор возбудит исключение (или будет выполнено действие, предусмотренное   
по умолчанию).  
Откровенно говоря, методы перегрузки операторов в большинстве своем ис-  
пользуются только при реализации объектов с математической природой –   
класс вектора или матрицы может, например, перегружать оператор сложе-  
ния, а класс служащего, скорее всего, нет. Чтобы упростить классы, вы можете   
вообще не использовать перегрузку и опираться на явные вызовы методов.  
C другой стороны, вы могли бы использовать перегрузку операторов, чтобы   
иметь возможность передавать объекты, определяемые пользователем, в функ-  
цию, которая выполняет операции, поддерживаемые встроенными типами,   
такими как списки или словари. Наличие реализации того же самого набора   
операторов в вашем классе обеспечит поддержку вашими объектами тех же са-  
мых интерфейсов и, как следствие, совместимость с используемой функцией.   
В этой книге мы не будем подробно рассматривать каждый из имеющихся ме-  
тодов перегрузки операторов, тем не менее в главе 29 вашему вниманию будут   
представлены дополнительные приемы перегрузки операторов в примерах.  
Один из методов перегрузки присутствует практически во всех реалистичных   
классах: метод-конструктор \_\_init\_\_. Он позволяет классам немедленно запол-  
нять атрибуты вновь созданных экземпляров, поэтому конструктор полезно   
использовать практически во всех разновидностях ваших классов. Фактиче-

Самый простой в мире класс на языке Python   
721  
ски даже при том, что в языке Python атрибуты не объявляются, вы без труда   
сможете определить, какие атрибуты принадлежат экземплярам, просмотрев   
программный код метода \_\_init\_\_.   
Самый простой в мире класс на языке Python  
В этой главе мы приступили к детальному изучению синтаксиса инструкции   
class, но я еще раз хотел бы напомнить, что сама модель наследования, на ко-  
торой основаны классы, очень проста – суть ее заключается всего лишь в орга-  
низации поиска атрибутов в деревьях взаимосвязанных объектов. Фактически   
мы можем создавать вообще пустые классы. Следующая инструкция создает   
класс без присоединенных к нему атрибутов (объект пустого пространства   
имен):  
>>> class rec: pass # Объект пустого пространства имен  
Инструкция пустой операции pass (обсуждавшаяся в главе 13) необходима по-  
тому, что здесь у нас нет никаких методов с программным кодом. После созда-  
ния класса, выполнив инструкцию в интерактивной оболочке, мы можем при-  
ступать к присоединению атрибутов, выполняя операции присваивания из-за   
пределов класса:  
>>> rec.name = ‘Bob’ # Так же для объектов с атрибутами  
>>> rec.age = 40  
После того как атрибуты будут созданы, мы можем обращаться к ним с помо-  
щью обычного синтаксиса. Когда класс используется таким способом, он напо-  
минает структуры в языке C или записи в языке Pascal – объект, с присоеди-  
C или записи в языке Pascal – объект, с присоеди-  
 или записи в языке Pascal – объект, с присоеди-  
Pascal – объект, с присоеди-  
 – объект, с присоеди-  
ненными к нему полями (то же самое можно проделывать с ключами словарей,   
но для этого потребуется вводить дополнительные символы):  
>>> print(rec.name) # Как структура в языке C или запись  
Bob  
Обратите внимание: такой подход будет работать даже в случае, когда еще не   
было создано ни одного экземпляра класса. Классы – это полноценные объек-  
ты, даже если нет ни одного экземпляра. Фактически они всего лишь самосто-  
ятельные пространства имен, поэтому, пока у нас имеется ссылка на класс, мы   
можем в любое время добавлять или изменять его атрибуты по своему усмотре-  
нию. Однако посмотрим, что произойдет, когда будут созданы два экземпляра   
класса:  
>>> x = rec() # Экземпляры наследуют имена из класса  
>>> y = rec()  
Эти экземпляры начинают свое существование как объекты абсолютно пустых   
пространств имен. Однако из-за того, что они помнят класс, из которого были   
созданы, они по наследству получат атрибуты, которые мы присоединили   
к классу:  
>>> x.name, y.name # Сейчас имена хранятся только в классе  
(‘Bob’, ‘Bob’)  
В действительности у этих экземпляров нет собственных атрибутов – они про-  
сто получают атрибут name из класса. Тем не менее если выполнить присваива-  
ние атрибуту экземпляра, будет создан (или изменен) атрибут этого объекта,

722   
Глава 26. Основы программирования классов   
а не другого – атрибуты обнаруживаются в результате поиска по дереву насле-  
дования, но операция присваивания значения атрибуту воздействует только на   
тот объект, к которому эта операция применяется. Ниже экземпляр x получает   
свой собственный атрибут name, а экземпляр y по-прежнему наследует атрибут   
name, присоединенный к классу выше его:  
>>> x.name = ‘Sue’ # Но присваивание изменит только объект x   
>>> rec.name, x.name, y.name  
(‘Bob’, ‘Sue’, ‘Bob’)  
Фактически, как будет более подробно рассказано в главе 28, атрибуты объек-  
та пространства имен обычно реализованы в виде словарей, и деревья наследо-  
вания классов (вообще говоря) тоже всего лишь словари со ссылками на другие   
словари. Если знать, куда смотреть, в этом можно убедиться явно.   
Например, в большинстве объектов, созданных на базе классов, имеется атри-  
бут \_\_dict\_\_, который является словарем пространства имен (некоторые клас-  
сы могут также определять атрибуты в \_\_slots\_\_ – с этой дополнительной и не-  
часто используемой особенностью мы познакомимся в главах 30 и 31). Ниже   
приводится пример интерактивного сеанса в Python 3.0 – порядок следования   
имен и перечень внутренних имен вида \_\_X\_\_ может изменяться от версии   
к версии, но имена, которые использовали, присутствуют во всех версиях:  
>>> rec.\_\_dict\_\_.keys()  
[‘\_\_module\_\_’, ‘name’, ‘age’, ‘\_\_dict\_\_’, ‘\_\_weakref\_\_’, ‘\_\_doc\_\_’]  
>>> list(x.\_\_dict\_\_.keys())  
[‘name’]  
>>> list(y.\_\_dict\_\_.keys()) # В Python 2.6 функцию list()   
[] # можно не использовать  
Здесь в словаре класса присутствуют атрибуты name и age, которые мы создали   
ранее, объект x имеет свой собственный атрибут name, а объект y по-прежнему   
пуст. Каждый экземпляр имеет ссылку на свой наследуемый класс, она назы-  
вается \_\_class\_\_, если вам захочется проверить ее:  
>>> x.\_\_class\_\_  
<class \_\_main\_\_.rec>  
Классы также имеют атрибут \_\_bases\_\_, который представляет собой кортеж   
его суперклассов:  
>>> rec.\_\_bases\_\_ # () пустой кортеж в Python 2.6  
(<class ‘object’>,)  
Эти два атрибута описывают, как деревья классов размещаются в памяти.  
Главное, что следует из этого взгляда на внутреннее устройство, это то, что мо-  
дель классов в языке Python чрезвычайно динамична. Классы и экземпляры –   
это всего лишь объекты пространств имен с атрибутами, создаваемыми на лету   
с помощью операции присваивания. Обычно эти операции присваивания вы-  
полняются внутри инструкции class, но они могут находиться в любом другом   
месте, где имеется ссылка на один из объектов в дереве.  
Даже методы, которые обычно создаются инструкциями def, вложенными   
в инструкцию class, могут создаваться совершенно независимо от объекта

Классы и словари   
723  
класса. Например, ниже определяется простая функция вне какого-либо клас-  
са, которая принимает единственный аргумент:  
>>> def upperName(self):  
... return self.name.upper() # Аргумент self по-прежнему необходим  
Здесь еще ничего не говорится о классе – это простая функция и она может   
вызываться как обычная функция при условии, что объект, получаемый ею,   
имеет атрибут name (в данном случае имя аргумента self не имеет никакого осо-  
бого смысла).   
>>> upperName(x) # Вызов, как обычной функции  
‘SUE’  
Однако, если эту простую функцию присвоить атрибуту нашего класса, она   
станет методом, вызываемым из любого экземпляра (а также через имя самого   
класса при условии, что функции вручную будет передан экземпляр):1  
>>> rec.method = upperName  
   
>>> x.method() # Вызвать метод для обработки x  
‘SUE’  
   
>>> y.method() # То же самое, но в self передается y  
‘BOB’  
   
>>> rec.method(x) # Можно вызвать через имя экземпляра или класса  
‘SUE’  
Обычно заполнение классов производится внутри инструкции class, а атри-  
буты экземпляров создаются в результате присваивания значений атрибутам   
аргумента self в методах. Однако отметим снова, что все это не является обяза-  
тельным, поскольку ООП в языке Python – это в основном поиск атрибутов во   
взаимосвязанных объектах пространств имен.  
Классы и словари   
Простые классы из предыдущего раздела призваны лишь проиллюстрировать   
основные особенности модели классов. Тем не менее представленные приемы   
могут также использоваться в настоящих программах. Например, в главе 8   
демонстрировалось, как использовать словари для хранения записей свойств   
сущностей в программах. Как оказывается, классы тоже способны играть эту   
роль – они могут хранить информацию, как словари, но при этом могут вклю-  
чать логику обработки этой информации в виде методов. Для справки ниже   
1   
Фактически это одна из причин, почему в языке Python аргумент self всегда дол-  
жен явно объявляться в методах, – потому что методы могут создаваться как простые   
функции, независимо от классов, и им необходим явный аргумент со ссылкой на под-  
разумеваемый экземпляр. В противном случае интерпретатор не смог бы обеспечить   
превращение простой функции в метод класса. Однако основная причина, по кото-  
рой аргумент self объявляется явно, заключается в том, чтобы сделать назначение   
имен более очевидным. Имена, к которым обращаются не через аргумент self, явля-  
ются простыми переменными, тогда как имена, обращение к которым происходит   
через аргумент self, очевидно являются атрибутами экземпляра.

724   
Глава 26. Основы программирования классов   
приводится пример использования записи на основе словаря, использовав-  
шийся ранее:  
>>> rec = {}  
>>> rec[‘name’] = ‘mel’ # Запись на основе словаря  
>>> rec[‘age’] = 40  
>>> rec[‘job’] = ‘trainer/writer’  
>>>  
>>> print rec[‘name’]  
mel  
Этот фрагмент имитирует инструмент, напоминающий записи и структуры   
в других языках программирования. Однако, как мы уже видели, существует   
еще множество способов сделать то же самое с помощью классов. Ниже приво-  
дится, пожалуй, самый простой из них – замена ключей атрибутами:  
>>> class rec: pass  
...  
>>> rec.name = ‘mel’ # Запись на основе класса  
>>> rec.age = 40  
>>> rec.job = ‘trainer/writer’  
>>>  
>>> print rec.age  
40  
Этот вариант существенно компактнее, чем эквивалент на базе словаря. Здесь   
для создания объекта пустого пространства имен используется пустая ин-  
струкция class. Создав пустой класс, мы заполняем его, присваивая значения   
его атрибутам.  
Этот прием работает, но для каждой отдельной записи придется писать новую   
инструкцию class. Пожалуй, более удобным будет создавать экземпляры клас-  
са всякий раз, когда нам потребуется новая запись:  
>>> class rec: pass  
...  
>>> pers1 = rec() # Запись на основе экземпляра  
>>> pers1.name = ‘mel’  
>>> pers1.job = ‘trainer’  
>>> pers1.age = 40  
>>>  
>>> pers2 = rec()  
>>> pers2.name = ‘dave’  
>>> pers2.job = ‘developer’  
>>>  
>>> pers1.name, pers2.name  
(‘mel’, ‘dave’)  
Здесь из одного и того же класса были созданы две записи – экземпляры клас-  
са начинают свое существование пустыми, как и классы. После этого произво-  
дится заполнение записей путем присваивания значений атрибутам. Однако   
на этот раз существует два отдельных объекта и, соответственно, два разных   
атрибута name. Фактически у экземпляров одного и того же класса не обяза-  
тельно должны быть одинаковые наборы имен атрибутов. В данном примере   
один из экземпляров имеет уникальный атрибут age. Экземпляры класса дей-  
ствительно являются разными пространствами имен: каждый из них имеет

В заключение   
725  
свой словарь атрибутов. Обычно экземпляры единообразно наполняются атри-  
бутами в методах класса, тем не менее они обладают большей гибкостью, чем   
можно было бы ожидать.  
Наконец, для реализации записи мы могли бы написать более полноценный   
класс:  
>>> class Person:  
... def \_\_init\_\_(self, name, job): # Класс = данные + логика  
... self.name = name  
... self.job = job  
... def info(self):  
... return (self.name, self.job)  
...  
>>> rec1 = Person(‘mel’, ‘trainer’)  
>>> rec2 = Person(‘vls’, ‘developer’)  
>>>  
>>> rec1.job, rec2.info()  
(‘trainer’, (‘vls’, ‘developer’))  
Такая схема также допускает создание множества экземпляров, но на этот   
раз класс уже не пустой: мы добавили в него логику (методы) инициализации   
экземпляров на этапе создания и сбора атрибутов в кортеж. Конструктор на-  
лагает некоторые ограничения целостности, требуя значения для двух атрибу-  
тов – name и job. Методы класса и атрибуты экземпляра вместе образуют пакет,   
объединяющий данные и логику.  
Мы могли бы продолжить расширение этой реализации, добавляя методы для   
вычисления зарплаты, разбора имен и так далее. В конце концов, мы могли   
бы связать класс в иерархию, чтобы обеспечить наследование набора суще-  
ствующих методов через процедуру автоматического поиска атрибутов клас-  
сов, и даже записывать экземпляры класса в файл, чтобы обеспечить их по-  
стоянное хранение. И мы действительно сделаем это – в следующей главе мы   
дополним эту аналогию между классами и записями более реалистичным при-  
мером, демонстрирующим основы использования классов.  
Наконец, несмотря на всю гибкость таких типов, как словари, классы позволя-  
ют наделять объекты поведением таким способом, который встроенными типа-  
ми и простыми функциями напрямую не поддерживается. Хотя мы и можем   
сохранять функции в словарях, тем не менее использование их для обработки   
данных в словарях не выглядит столь же естественным, как такое их исполь-  
зование в классах.  
В заключение  
В этой главе были представлены основы программирования классов в языке   
Python. Мы изучили синтаксис инструкции class и увидели, как ее можно ис-  
пользовать для построения дерева классов. Мы также узнали, что интерпрета-  
тор Python автоматически заполняет первый аргумент методов, что атрибуты   
присоединяются к объектам в дереве классов простым присваиванием и что   
существуют специальные имена методов перегрузки операторов, позволяющие   
реализовать выполнение встроенных операций в наших объектах (например,   
операции выражений и вывод).

726   
Глава 26. Основы программирования классов   
Теперь, когда мы познакомились с основами программирования классов на   
языке Python, в следующей главе мы обратимся к большому и более реали-  
Python, в следующей главе мы обратимся к большому и более реали-  
, в следующей главе мы обратимся к большому и более реали-  
стичному примеру, в котором используется многое из того, что мы уже зна-  
ем об ООП. После этого мы продолжим изучение программирования классов   
и повторно рассмотрим модель, чтобы восполнить недостающие подробности,   
которые были опущены здесь для простоты. Однако прежде ответьте на кон-  
трольные вопросы, чтобы освежить в памяти основы, которые были рассмотре-  
ны здесь.  
Закрепление пройденного  
Контрольные вопросы  
1. Как классы связаны с модулями?  
2. Как создаются классы и экземпляры классов?  
3. Где и как создаются атрибуты классов?  
4. Где и как создаются атрибуты экземпляров классов?  
5. Что в языке Python означает слово self для классов?  
6. Как производится перегрузка операторов в классах на языке Python?  
7. Когда может потребоваться перегрузка операторов в ваших классах?  
8. Какой метод перегрузки оператора используется наиболее часто?  
9. Какие две концепции ООП являются наиболее важными в языке Python?  
Ответы  
1. Классы всегда находятся внутри модулей – они являются атрибутами объ-  
екта модуля. Классы и модули являются пространствами имен, но классы   
соответствуют инструкциям (а не целым файлам) и поддерживают такие   
понятия ООП, как экземпляры класса, наследование и перегрузка опера-  
торов. В некотором смысле модули напоминают классы с единственным эк-  
земпляром, без наследования, которые соответствуют целым файлам.  
2. Классы создаются с помощью инструкций class, а экземпляры создаются   
вызовом класса, как если бы это была функция.  
3. Атрибуты класса создаются присваиванием атрибутам объекта класса.   
Обычно они создаются инструкциями верхнего уровня в инструкции class –   
каждое имя, которому будет присвоено значение внутри инструкции class,   
становится атрибутом объекта класса (с технической точки зрения область   
видимости инструкции class преобразуется в пространство имен атрибутов   
объекта класса). Атрибуты класса могут также создаваться через присваи-  
вание атрибутам класса в любом месте, где доступна ссылка на объект клас-  
са, то есть даже за пределами инструкции class.  
4. Атрибуты экземпляра создаются присваиванием атрибутам объекта эк-  
земпляра. Обычно они создаются внутри методов класса, в инструкции   
class – присваиванием значений атрибутам аргумента self (который всегда   
является подразумеваемым экземпляром). Повторюсь: возможно создавать   
атрибуты с помощью операции присваивания в любом месте программы,   
где доступна ссылка на экземпляр, даже за пределами инструкции class.

Закрепление пройденного   
727  
Обычно все атрибуты экземпляров инициализируются в конструкторе \_\_  
init\_\_ – благодаря этому при последующих вызовах методов можно быть   
уверенным в существовании необходимых атрибутов.  
5. self – это имя, которое обычно дается первому (самому левому) аргументу   
в методах классов. Интерпретатор Python автоматически записывает в него   
объект экземпляра, который подразумевается при вызове метода. Этот ар-  
гумент не обязательно должен носить имя self, главное – это положение   
аргумента. (Бывшие программисты на C++ или Java могут предпочесть ис-  
пользовать имя this, потому что в этих языках программирования это имя   
является отражением той же идеи, только в языке Python этот аргумент   
всегда должен присутствовать явно.)  
6. Перегрузка операторов в языке Python выполняется с помощью методов со   
специальными именами – они начинаются и заканчиваются двумя симво-  
лами подчеркивания. Эти имена не являются встроенными или зарезер-  
вированными именами – интерпретатор Python просто автоматически вы-  
зывает методы с этими именами, когда экземпляр появляется в составе со-  
ответствующей операции. Язык Python определяет порядок отображения   
операций на специальные имена методов.  
7. Перегрузка операторов может использоваться для реализации объектов, ко-  
торые имитируют поведение встроенных типов (например, последователь-  
ностей или числовых объектов, таких как матрицы), и для реализации ин-  
терфейсов встроенных типов, которые ожидают получить те или иные части   
программного кода. Кроме того, имитация интерфейсов встроенных типов   
позволяет передавать в экземплярах классов информацию о состоянии – то   
есть атрибуты, в которых сохраняются данные между вызовами операций.   
Однако не следует использовать перегрузку операторов, когда достаточно ис-  
пользовать простые методы.  
8. Наиболее часто используется метод конструктора \_\_init\_\_ – этот метод   
присутствует практически во всех классах и используется для установки   
начальных значений атрибутов экземпляров и выполнения других началь-  
ных операций.  
9. Наиболее важными концепциями ООП в языке Python являются аргумент   
self в методах и конструктор \_\_init\_\_.

Глава 27.  
   
Более реалистичный пример  
В следующей главе мы углубимся в тонкости синтаксиса классов. Однако   
перед этим я хотел бы вам показать более реалистичный пример использова-  
ния классов, имеющий более высокую практическую ценность, чем примеры,   
которые мы рассматривали до сих пор. В этой главе мы создадим множество   
классов, решающих вполне конкретные задачи, – сохранение и обработку ин-  
формации о людях. Как вы увидите далее, те программные компоненты, кото-  
рые мы называем экземплярами и классами, часто могут играть роль записей   
и программ в более традиционном понимании.  
Более конкретно, в этой главе мы создадим два класса:  
 •  
Person – класс, который представляет и обрабатывает информацию о людях  
 •  
Manager – адаптированная версия класса Person, модифицирующая унасле-  
дованное поведение  
Попутно мы создадим экземпляры обоих классов и протестируем их возмож-  
ности. По окончании я покажу вам отличный пример использования классов –   
мы сохраним наши экземпляры в хранилище, в объектно-ориентированной   
базе данных, обеспечивающей долговременной их хранение. Благодаря этому   
вы сможете использовать программный код примера как шаблон для создания   
своей собственной, полноценной базы данных, целиком написанной на языке   
Python.  
Однако, помимо получения практических результатов, наша основная цель   
заключается в обучении: эта глава представляет собой учебник по объектно-  
ориентированному программированию на языке Python. Многие в достаточ-  
Python. Многие в достаточ-  
. Многие в достаточ-  
ной степени осваивают синтаксис после прочтения предыдущей главы, но за-  
частую не знают, с чего начать, когда сталкиваются с необходимостью создать   
новый класс с самого начала. Поэтому мы сделаем еще один шаг в освоении   
основ – мы постепенно будем создавать классы, чтобы вы могли увидеть, как из   
отдельных особенностей составляются законченные программы.  
В конечном итоге мы получим классы, не очень большие, с точки зрения объ-  
ема программного кода, но позволяющие рассмотреть все основные идеи, ле-  
жащие в основе модели ООП на языке Python. Если отвлечься от особенностей   
синтаксиса, система классов в языке Python на практике сводится к поиску

Шаг 1: создание экземпляров   
729  
атрибутов в дереве объектов и наличию специального первого аргумента в слу-  
чае функций.  
Шаг 1: создание экземпляров   
Итак, все, что требовалось сказать о наших целях, уже сказано, поэтому теперь   
перейдем к реализации. Наша первая задача – начать создание главного клас-  
са Person. Откройте в своем любимом текстовом редакторе новый файл, куда   
мы будем записывать программный код. В языке Python существует соглаше-  
Python существует соглаше-  
 существует соглаше-  
ние, согласно которому имена модулей начинаются со строчной буквы, а имена   
классов – с прописной. Точно так же, в соответствии с соглашениями, первому   
аргументу методов присваивается имя self. Эти соглашения не являются обя-  
зательными, но они получили настолько широкое распространение, что отказ   
от следования им может ввести в заблуждение тех, кто позднее будет читать   
ваш программный код. В соответствии с этими соглашениями мы назовем наш   
файл person.py, а классу дадим имя Person, как показано ниже:  
# Файл person.py (начало)  
class Person:  
До определенного момента в данной главе мы будем работать с этим файлом.   
Мы можем запрограммировать в одном файле любое количество функций   
и классов, поэтому название person.py может потерять свой смысл, если позд-  
нее мы добавим в него дополнительные компоненты, никак не связанные с его   
начальным предназначением. Но пока мы будем полагать, что все, что на-  
ходится в этом файле, так или иначе связано с классом Person. В идеале так   
и должно быть – как мы уже знаем, модуль только выигрывает, когда он создан   
ради единственной, логически связной цели.  
Конструкторы  
Первое, что нам требуется сделать с классом Person, – это записать основные   
сведения о человеке, то есть заполнить поля записи. На языке Python они на-  
Python они на-  
 они на-  
зываются атрибутами объекта и обычно создаются с помощью операций при-  
сваивания значений атрибутам аргумента self в методах класса. Обычно пер-  
вые значения атрибутам экземпляра присваиваются в методе конструктора   
\_\_init\_\_, который вызывается автоматически всякий раз, когда создается но-  
вый экземпляр. Давайте добавим этот конструктор к нашему классу:  
# Добавим инициализацию полей записи  
   
class Person:  
 def \_\_init\_\_(self, name, job, pay): # Конструктор принимает 3 аргумента  
 self.name = name # Заполняет поля при создании  
 self.job = job # self – новый экземпляр класса  
 self.pay = pay  
Это достаточно распространенный прием: мы передаем конструктору аргумен-  
ты с данными, которые будут храниться экземпляром, и присваиваем их атри-  
бутам аргумента self. В терминах ООП аргумент self представляет вновь соз-  
данный экземпляр, а аргументы name, job и pay превращаются в информацию   
о состоянии – данные, сохраняемые в объекте для последующего использова-  
ния. Для сохранения информации могут использоваться и другие приемы (та-

730   
Глава 27. Более реалистичный пример   
кие как ссылки на объемлющую область видимости), но атрибуты экземпляра   
являются более очевидными и простыми для понимания.  
Обратите внимание, что имена аргументов дважды используются в операциях   
присваивания. Этот программный код может на первый взгляд показаться из-  
быточным, но это не так. Аргумент job, например, – это локальная переменная   
в области видимости функции \_\_init\_\_, а self.job – это атрибут экземпляра,   
который является подразумеваемым контекстом вызова метода. Это две раз-  
ные переменные, которые по совпадению имеют одно и то же имя. Присваивая   
значение локальной переменной job атрибуту self.job с помощью операции self.  
job=job, мы сохраняем его в экземпляре для последующего использования. Как   
обычно, место, где выполняется присваивание значения имени, определяет   
смысл этого имени.  
Продолжая разговор об аргументах, отмечу, что в методе \_\_init\_\_ нет ничего   
необычного, кроме того, что он вызывается автоматически в момент создания   
экземпляра и первый его аргумент имеет специальное значение. Несмотря на   
непривычное название, это самая обычная функция, обладающая всеми осо-  
бенностями функций, о которых мы уже знаем. Мы можем, например, опреде-  
лить значения по умолчанию для некоторых аргументов, чтобы их можно было   
не указывать в тех случаях, когда какие-то определенные значения недоступ-  
ны или бессмысленны.  
Для демонстрации сделаем аргумент job необязательным – по умолчанию ис-  
пользуем значение None, означающее, что данный человек (в настоящий мо-  
мент) является безработным. Если аргумент job получает значение по умолча-  
нию None, вероятно, имеет смысл дать аргументу pay (зарплата) значение 0 (если   
только у вас нет знакомых, которые, будучи безработными, получают зарпла-  
ту!). В действительности, мы вынуждены указать значение по умолчанию для   
аргумента pay, потому что этого требует синтаксис языка Python, – любые ар-  
Python, – любые ар-  
, – любые ар-  
гументы в заголовке функции, следующие за первым аргументом, имеющим   
значение по умолчанию, также должны иметь значения по умолчанию:  
# Добавим значения по умолчанию для аргументов конструктора  
   
class Person:  
 def \_\_init\_\_(self, name, job=None, pay=0): # Normal function args  
 self.name = name  
 self.job = job  
 self.pay = pay  
Этот программный код означает, что при создании экземпляров класса Person   
нам достаточно будет передавать только значение аргумента name, а аргументы   
job и pay теперь являются необязательными – они по умолчанию будут полу-  
чать значения None и 0. Аргумент self, как обычно, будет заполняться интер-  
претатором автоматически, и в нем будет передаваться ссылка на экземпляр   
созданного объекта – присваивание значений атрибутам объекта self будет   
присоединять их к новому экземпляру.  
Тестирование в процессе разработки  
Этот класс пока может не очень много – по сути, он всего лишь заполняет поля   
новой записи, – но это настоящий действующий класс. Теперь мы могли бы   
добавить в него реализацию дополнительных особенностей, но пока мы оста-  
новимся на достигнутом. Как вы уже, вероятно, начинаете понимать, про-

Шаг 1: создание экземпляров   
731  
граммирование на языке Python в действительности сводится к постепенному   
наращиванию возможностей – вы пишете некоторый программный код, тести-  
руете его, добавляете еще программный код, снова тестируете и так далее. По-  
скольку Python предоставляет интерактивную оболочку и возможность прак-  
Python предоставляет интерактивную оболочку и возможность прак-  
 предоставляет интерактивную оболочку и возможность прак-  
тически сразу опробовать изменения в программном коде, более естественным   
представляется выполнять тестирование по мере движения вперед, а не после   
создания огромного объема программного кода.  
Прежде чем добавлять дополнительные возможности, давайте протестируем   
то, что у нас уже получилось, создав несколько экземпляров нашего класса   
и посмотрев содержимое их атрибутов, созданных конструктором. Мы могли   
бы выполнить такое тестирование в интерактивном сеансе, но, как вы уже   
наверняка понимаете, тестирование в интерактивном сеансе имеет свои огра-  
ничения – достаточно утомительно всякий раз при запуске нового сеанса вы-  
полнять повторное импортирование модулей и повторно вводить инструкции.   
Программисты на языке Python используют интерактивный сеанс лишь для   
простых тестов, а для проведения более полного тестирования предпочитают   
добавлять программный код в конец файла, содержащего тестируемые объек-  
ты, например:  
# Добавляем программный код для самопроверки  
   
class Person:  
 def \_\_init\_\_(self, name, job=None, pay=0):  
 self.name = name  
 self.job = job  
 self.pay = pay  
   
bob = Person(‘Bob Smith’) # Тестирование класса  
sue = Person(‘Sue Jones’, job=’dev’, pay=100000) # Запустит \_\_init\_\_   
 # автоматически  
print(bob.name, bob.pay) # Извлечет атрибуты  
print(sue.name, sue.pay) # Атрибуты в объектах sue и   
 # отличаются  
Обратите внимание, что объект bob получает значения атрибутов job и pay по   
умолчанию, а для объекта sue значения всех атрибутов указываются явно. От-  
метьте также, что при создании объекта sue мы использовали именованные   
аргументы, – мы могли бы передать значения в виде позиционных аргумен-  
тов, однако именованные аргументы позволят нам позднее вспомнить, какие   
данные передавались (кроме того, они позволяют нам указывать аргументы   
в любом порядке). Напомню еще раз, что за исключением необычного имени,   
метод \_\_init\_\_ – это самая обычная функция, поддерживающая все особенно-  
сти функций, которые нам известны, включая возможность определения зна-  
чений аргументов по умолчанию и передачу именованных аргументов.  
Если запустить этот файл как сценарий, программный код в конце файла соз-  
даст два экземпляра нашего класса и выведет значения двух атрибутов для   
каждого из них (name и pay):  
C:\misc> person.py  
Bob Smith 0  
Sue Jones 100000  
Этот же программный код, выполняющий тестирование, можно было бы вве-  
сти в интерактивном сеансе (предварительно импортировав класс Person), но

732   
Глава 27. Более реалистичный пример   
оформление тестов внутри модуля, как в данном примере, существенно упро-  
щает повторный запуск тестов в будущем.  
Хотя это достаточно простой программный код, тем не менее он демонстрирует   
некоторые важные моменты. Обратите внимание, что значение атрибута name   
в объекте bob не совпадает со значением атрибута name в объекте sue, а значение   
атрибута pay в объекте sue не совпадает со значением атрибута pay в объекте bob.   
Каждый из объектов является независимой записью с собственной информа-  
цией. Технически bob и sue являются пространствами имен – подобно всем   
экземплярам класса, каждый из них обладает собственной копией информа-  
ции о состоянии. Так как каждый экземпляр класса обладает собственным   
набором атрибутов, классы обеспечивают естественный способ сохранения   
информации о множестве объектов – так же, как и встроенные типы, классы   
играют роль своеобразной фабрики объектов. Другие программные конструк-  
ции в языке Python, такие как функции и модули, не обладают такой возмож-  
Python, такие как функции и модули, не обладают такой возмож-  
, такие как функции и модули, не обладают такой возмож-  
ностью.  
Двоякое использование программного кода  
Тестовый программный код в конце файла работает без нареканий, но здесь   
кроется одно большое неудобство – инструкции print на верхнем уровне будут   
выполняться и при запуске файла как сценария, и при импортировании его   
как модуля. Это означает, что если мы решим импортировать класс из этого   
файла в какой-нибудь программе (что мы и сделаем далее в этой главе), мы   
будем наблюдать результаты тестирования всякий раз, когда будем импорти-  
ровать этот файл. Однако это не очень хорошо для модулей: клиентская про-  
грамма, вероятно, не должна заботиться о внутреннем тестировании, и было   
бы нежелательно, чтобы выводимые программой данные перемежались ре-  
зультатами тестирования модуля.  
Мы могли бы поместить программный код тестов в отдельный файл, однако   
гораздо удобнее, когда тесты находятся в одном файле с тестируемыми компо-  
нентами. Гораздо лучше оформить тесты так, чтобы они выполнялись, только   
когда файл запускается как сценарий для тестирования, а не при его импорти-  
ровании. Как вы уже знаете из предыдущей части книги, для этой цели можно   
использовать проверку атрибута \_\_name\_\_ модуля. Соответствующие измене-  
ния в файле приводятся ниже:  
# Предусмотреть возможность импортировать файл и запускать его, как   
# самостоятельный сценарий для самотестирования  
   
class Person:  
 def \_\_init\_\_(self, name, job=None, pay=0):  
 self.name = name  
 self.job = job  
 self.pay = pay  
   
if \_\_name\_\_ == ‘\_\_main\_\_’: # Только когда файл запускается для тестирования  
 # реализация самотестирования  
 bob = Person(‘Bob Smith’)  
 sue = Person(‘Sue Jones’, job=’dev’, pay=100000)  
 print(bob.name, bob.pay)  
 print(sue.name, sue.pay)  
Теперь мы получили именно то поведение, которого добивались, – тестирова-  
ние выполняется, когда файл запускается как самостоятельный сценарий, по-

Шаг 2: добавление методов, определяющих поведение   
733  
тому что атрибут \_\_name\_\_ модуля получает значение ‘\_\_main\_\_’, а при импорти-  
ровании этого не происходит:  
C:\misc> person.py  
Bob Smith 0  
Sue Jones 100000  
   
c:\misc> python  
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) ...  
>>> import person  
>>>  
Теперь при импортировании файла интерпретатор создаст новый класс, но не   
будет использовать его. При запуске файла в качестве сценария интерпретатор   
создаст два экземпляра класса и выведет значения двух атрибутов для каждого   
из них – напомню еще раз, что каждый экземпляр является независимым про-  
странством имен, поэтому их атрибуты могут иметь различающиеся значения.  
Примечание о переносимости между версиями  
Я запускал примеры из этой главы под управлением Python 3.0 и ис-  
пользовал синтаксис вызова функции print в версии 3.0. Если вы поль-  
зуетесь Python 2.6, этот программный код тоже будет работать, но вы   
будете замечать круглые скобки вокруг некоторых строк, потому что   
дополнительные круглые скобки в инструкции print превращают   
множество выводимых элементов в кортеж:  
c:\misc> c:\python26\python person.py  
(‘Bob Smith’, 0)  
(‘Sue Jones’, 100000)  
Если такие отличия могут лишить вас сна, тогда просто удалите кру-  
глые скобки в вызовах инструкций print при опробовании примеров   
в версии 2.6. Вы также можете избежать появления лишних круглых   
скобок, используя выражения форматирования с целью получения   
единственного объекта для вывода. Любой из следующих приемов   
одинаково работает в версиях 2.6 и 3.0, хотя способ, основанный на   
применении метода, является более современным:  
print(‘{0} {1}’.format(bob.name, bob.pay)) # Новый метод format  
print(‘%s %s’ % (bob.name, bob.pay)) # Выражение форматирования  
Шаг 2: добавление методов,   
определяющих поведение  
Пока все идет неплохо – к настоящему моменту наш класс фактически играет   
роль фабрики записей – он создает записи и заполняет их поля (атрибуты экзем-  
пляров, если говорить на языке Python). Однако даже при такой ограниченной   
реализации мы уже можем применять к таким записям некоторые операции   
над объектами. Хотя классы и добавляют дополнительный структурный уро-  
вень, тем не менее большую часть своей работы они выполняют за счет внедре-  
ния и обработки данных базовых типов, таких как списки и строки. Другими

734   
Глава 27. Более реалистичный пример   
словами, если вы уже знаете, как использовать простые базовые типы данных   
языка Python, значит, вы уже многое знаете о классах – в действительности   
классы являются лишь небольшой структурной надстройкой.  
Например, поле name в наших объектах является обычной строкой, поэтому мы   
в состоянии извлекать фамилии людей из наших объектов, разбивая значение   
атрибута по пробелам и используя операцию индексирования. Все это – опе-  
рации над базовыми типами данных, которые работают независимо от того,   
входит ли объект операции в состав экземпляра класса или нет:  
>>> name = ‘Bob Smith’ # Простая строка, за пределами класса  
>>> name.split() # Извлечение фамилии  
[‘Bob’, ‘Smith’]  
>>> name.split()[-1] # Или [1], если имя всегда состоит из 2 компонентов  
‘Smith’  
Точно так же мы можем увеличить зарплату, изменив значение поля pay, – то   
есть, изменив информацию о состоянии с помощью присваивания. Данная опе-  
рация также относится к базовым операциям в языке Python, действие кото-  
Python, действие кото-  
, действие кото-  
рой не зависит от того, является объект операции самостоятельным объектом   
или частью структуры класса:  
>>> pay = 100000 # Простая переменная, за пределами класса  
>>> pay \*= 1.10 # Поднять на 10%. Или: pay = pay \* 1.10,  
>>> print(pay) # Или: pay = pay \* 1.10, если вы любите вводить с клавиатуры  
110000.0 # Или: pay = pay + (pay \* .10), если быть более точными!  
Чтобы применить те же самые операции к объектам класса Person, созданным   
нашим сценарием, просто подставьте имена bob.name и sue.pay на место name   
и pay. Операции останутся теми же самыми, но в качестве объектов операций   
будут использоваться атрибуты нашего класса:  
# Обработка встроенных типов: строки, изменяемость  
   
class Person:  
 def \_\_init\_\_(self, name, job=None, pay=0):  
 self.name = name  
 self.job = job  
 self.pay = pay  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 bob = Person(‘Bob Smith’)  
 sue = Person(‘Sue Jones’, job=’dev’, pay=100000)  
 print(bob.name, bob.pay)  
 print(sue.name, sue.pay)  
 print(bob.name.split()[-1]) # Извлечь фамилию  
 sue.pay \*= 1.10 # Повысить зарплату  
 print(sue.pay)  
Здесь мы добавили в конец три новые строки – они извлекают фамилию из объ-  
екта bob, используя простые операции над строками и списками, и поднимают   
зарплату sue, изменяя значение атрибута pay с помощью простой числовой опе-  
рации. В некотором смысле объект sue является изменяемым – он допускает   
непосредственное изменение своей информации о состоянии, подобно списку,   
при вызове метода append:  
Bob Smith 0  
Sue Jones 100000

Шаг 2: добавление методов, определяющих поведение   
735  
Smith  
110000.0  
Предыдущий программный код действует именно так, как и задумывалось, но   
если показать его опытному программисту, он наверняка сообщит, что такой   
подход нежелательно применять на практике. Выполнение операций за преде-  
лами класса, как в данном примере, может привести к проблемам при сопро-  
вождении.  
Например, представьте, что в самых разных местах программы присутствуют   
одинаковые фрагменты, извлекающие фамилию. Если вам потребуется изме-  
нить их (например, в случае изменения структуры поля name), вам придется   
отыскать и изменить все такие фрагменты. То же относится и к операции изме-  
нения заработной платы (например, в случае, когда такое изменение потребует   
подтверждения или обновления базы данных) – вам придется изменить мно-  
жество копий одного и того же фрагмента. Один только поиск всех фрагментов   
в крупных программах может оказаться весьма проблематичной задачей – они   
могут быть разбросаны по разным файлам, разбиты на отдельные операции   
и так далее.  
Методы реализации  
Что нам действительно сейчас необходимо, так это реализовать концепцию   
проектирования, которая называется инкапсуляцией. Идея инкапсуляции   
заключается в том, чтобы спрятать логику операций за интерфейсами и тем   
самым добиться, чтобы каждая операция имела единственную реализацию   
в нашей программе. Благодаря такому подходу, если в дальнейшем нам по-  
требуется вносить какие-либо изменения, модифицировать программный код   
придется только в одном месте. Кроме того, мы сможем изменять внутреннюю   
реализацию операции практически как угодно, не рискуя нарушить работо-  
способность программного кода, использующего ее.  
В терминах языка Python это означает, что мы должны реализовать операции   
над объектами в виде методов класса, а не разбрасывать их по всей програм-  
ме. Фактически возможность сосредоточить программный код в одном месте,   
устранить избыточность и тем самым упростить его сопровождение является   
одной из самых сильных сторон классов. Как дополнительное преимущество,   
оформление операций в виде методов позволяет применять их к любым экзем-  
плярам класса, а не только к тем, которые явно задействованы в обработке.  
На практике все это выглядит гораздо проще, чем в теории. В следующем ли-  
стинге мы переместили реализацию двух операций из программы в методы   
класса, добившись инкапсуляции. Давайте попутно изменим программный   
код самопроверки внизу файла и заменим в нем жестко запрограммированные   
операции вызовами методов:  
# Добавлены методы, инкапсулирующие операции, для удобства в сопровождении  
   
class Person:  
 def \_\_init\_\_(self, name, job=None, pay=0):  
 self.name = name  
 self.job = job  
 self.pay = pay  
 def lastName(self): # Методы, реализующие поведение экземпляров  
 return self.name.split()[-1] # self – подразумеваемый экземпляр  
 def giveRaise(self, percent):

736   
Глава 27. Более реалистичный пример   
 self.pay = int(self.pay \* (1 + percent)) # Изменения придется вносить   
 # только в одном месте  
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 bob = Person(‘Bob Smith’)  
 sue = Person(‘Sue Jones’, job=’dev’, pay=100000)  
 print(bob.name, bob.pay)  
 print(sue.name, sue.pay)  
 print(bob.lastName(), sue.lastName()) # Вместо жестко определенных   
 sue.giveRaise(.10) # операций используются методы  
 print(sue.pay)  
Как мы уже знаем, методы – это самые обычные функции, которые присоеди-  
няются к классам и предназначены для обработки экземпляров этих классов.   
Экземпляр – это подразумеваемый контекст вызова метода, который автома-  
тически передается методу в виде аргумента self.  
Преобразование операций в методы в данной версии программы оказалось до-  
статочно простым делом. Новый метод lastName, например, просто выполняет   
над объектом self ту же операцию, которая в предыдущей версии выполнялась   
над объектом bob. Здесь self – это подразумеваемый объект, являющийся кон-  
текстом вызова метода. Кроме того, метод lastName возвращает результат. Дан-  
ная операция фактически является вызовом функции – она вычисляет зна-  
чение, которое затем может использоваться вызывающей программой, пусть   
даже просто для вывода. Точно так же новый метод giveRaise выполняет над   
объектом self операцию, которая в предыдущей версии выполнялась над объ-  
ектом sue.  
Если теперь запустить наш сценарий, он выведет те же результаты, что и пре-  
жде, – мы всего лишь реструктурировали программный код, чтобы упростить   
возможность его модификации в будущем, не изменяя его поведения:  
Bob Smith 0  
Sue Jones 100000  
Smith Jones  
110000  
Здесь следует отметить несколько интересных особенностей. Во-первых, об-  
ратите внимание, что поле pay (заработная плата) в объекте sue по-прежнему   
получает целочисленное значение после его увеличения – внутри метода мы   
преобразовали результат арифметической операции в целое число с помощью   
встроенной функции int. Выбор типа значения int или float, вероятно, не имеет   
существенного значения в большинстве случаев (целые и вещественные числа   
обладают одинаковыми интерфейсами и могут смешиваться в выражениях),   
но в действующей системе мы должны позаботиться о проблеме округления   
(деньги все-таки имеют определенную значимость для людей!).  
Из главы 5 мы знаем, что могли бы также реализовать округление до центов   
с помощью встроенной функции round(N, 2), использовать тип decimal для обе-  
спечения фиксированной точности или хранить денежные суммы в виде ве-  
щественных чисел и отображать их с применением строки формата %.2f или   
{0:.2f}. В нашем примере мы просто отсекаем центы с помощью функции int.   
(Еще один способ можно увидеть в модуле formats.py, в главе 24, – вы можете   
импортировать функцию money и с ее помощью отображать сумму заработной   
платы с запятыми, центами и знаком доллара.)  
Во-вторых, обратите также внимание, что на этот раз мы добавили вывод фа-  
милии из объекта sue, – поскольку теперь логика получения фамилии была

Шаг 3: перегрузка операторов   
737  
инкапсулирована в виде метода, мы можем применить ее к любому экземпляру   
класса. Как видно из примера, интерпретатор сообщает методу, какой экзем-  
пляр должен обрабатываться, автоматически передавая его в первом аргумен-  
те, которому обычно дается имя self. В частности:  
 •  
В первом вызове, bob.lastName(), в аргументе self передается подразумевае-  
мый объект bob.  
 •  
Во втором вызове, sue.lastName(), в аргументе self передается уже объект   
sue.   
Проследите, как выполняются эти вызовы, чтобы увидеть, как экземпляры   
оказываются в аргументе self. Суть заключается в том, что каждый раз метод   
извлекает значение атрибута name подразумеваемого объекта. То же относит-  
ся и к методу giveRaise. Мы могли бы попытаться поднять зарплату и персоне,   
представленной объектом bob, вызвав метод giveRaise для обоих экземпляров,   
но, к сожалению, нулевая зарплата в объекте bob сведет на нет наши усилия,   
поскольку именно так реализована программа в настоящее время (возможно,   
мы пожелаем решить эту проблему в будущей версии 2.0 нашей программы).  
Наконец, обратите внимание, что метод giveRaise предполагает, что в аргумен-  
те percent он получит вещественное число в диапазоне от нуля до единицы. Воз-  
можно, это слишком радикальное предположение (хотя увеличение зарплаты   
на 1000%, вероятно, было бы воспринято как ошибка большинством из нас!)�   
в нашем прототипе мы позволим передавать числа, выходящие за этот диапа-  
зон, но на следующем этапе разработки программы было бы желательно до-  
бавить проверку значения аргумента или хотя бы описать эту особенность в до-  
кументации. В одной из следующих глав мы познакомимся с одним из спосо-  
бов решения этой проблемы, когда будем рассматривать декораторы функций   
и исследовать инструкцию assert – альтернативы, позволяющие реализовать   
автоматическую проверку в ходе разработки.  
Шаг 3: перегрузка операторов  
К настоящему моменту у нас имеется полноценный класс, позволяющий соз-  
давать и инициализировать экземпляры и обладающий двумя новыми мето-  
дами, выполняющими обработку экземпляров. Пока все идет очень неплохо.  
Однако тестирование все еще выполняется не так удобно, как могло бы, – для   
проверки объектов нам приходится вручную извлекать и выводить значения   
отдельных атрибутов (например, bob.name, sue.pay). Было бы совсем неплохо,   
если бы вывод экземпляра целиком предоставлял нам некоторую нужную ин-  
формацию. К сожалению, формат вывода объектов экземпляров, используе-  
мый по умолчанию, выглядит не очень удобочитаемо – он предусматривает   
вывод имени класса объекта и его адреса в памяти (который в языке Python   
не имеет практической ценности, кроме того, что идентифицирует объект уни-  
кальным образом).  
Чтобы увидеть, как выглядит вывод объектов в этом формате, замените послед-  
нюю строку в сценарии на вызов print(sue), который отобразит объект целиком.   
Ниже приводится результат запуска измененного сценария (здесь видно, что   
sue – это «object» в версии 3.0 и «instance» в версии 2.6):  
Bob Smith 0  
Sue Jones 100000

738   
Глава 27. Более реалистичный пример   
Smith Jones  
<\_\_main\_\_.Person object at 0x02614430>  
Реализация отображения  
К счастью, большего успеха можно добиться, задействовав возможность пере-  
грузки операторов, – добавив в класс метод, который перехватывает и выпол-  
няет встроенную операцию, когда она применяется к экземплярам класса.   
В частности, мы могли бы реализовать метод перегрузки операторов, занимаю-  
щий, пожалуй, второе место по частоте использования после метода \_\_init\_\_:   
метод \_\_str\_\_, представленный в предыдущей главе. Метод \_\_str\_\_ вызывается   
автоматически всякий раз, когда экземпляр преобразуется в строку для вы-  
вода. Поскольку этот метод используется для вывода объекта, фактически   
все, что мы получаем при выводе объекта, является возвращаемым значением   
метода \_\_str\_\_ этого объекта, который может быть определен в классе объекта   
или унаследован от суперкласса (методы, имена которых начинаются и окан-  
чиваются двумя символами подчеркивания, наследуются точно так же, как   
любые другие).  
С технической точки зрения метод конструктора \_\_init\_\_, который мы уже   
реализовали, также является методом перегрузки операторов – он автомати-  
чески вызывается на этапе конструирования для инициализации вновь соз-  
данного экземпляра. Конструкторы используются настолько часто, что они   
практически не выглядят, как нечто особенное. Более специализированные   
методы, такие как \_\_str\_\_, позволяют нам перехватывать определенные опера-  
ции и предусматривать специфическую реализацию поведения объектов, уча-  
ствующих в этих операциях.  
Давайте добавим реализацию этого метода в наш класс. Ниже приводится   
расширенная версия класса, который выводит список атрибутов при отобра-  
жении экземпляров целиком и не полагается на менее полезную реализацию   
вывода по умолчанию:  
# Добавлен метод \_\_str\_\_, реализующий вывод объектов  
   
class Person:  
 def \_\_init\_\_(self, name, job=None, pay=0):  
 self.name = name  
 self.job = job  
 self.pay = pay  
 def lastName(self):  
 return self.name.split()[-1]  
 def giveRaise(self, percent):  
 self.pay = int(self.pay \* (1 + percent))  
 def \_\_str\_\_(self): # Добавленный метод  
 return ‘[Person: %s, %s]’ % (self.name, self.pay) # Строка для вывода  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 bob = Person(‘Bob Smith’)  
 sue = Person(‘Sue Jones’, job=’dev’, pay=100000)  
 print(bob)  
 print(sue)  
 print(bob.lastName(), sue.lastName())  
 sue.giveRaise(.10)  
 print(sue)

Шаг 4: адаптация поведения с помощью подклассов   
739  
Обратите внимание, что здесь в методе \_\_str\_\_ для создания строки вывода мы   
применили оператор форматирования %, – для реализации необходимых дей-  
ствий классы могут использовать встроенные типы объектов и операции, как   
в данном случае. Напомню еще раз, все, что мы уже знаем о встроенных типах   
данных и функциях, может применяться при создании классов. По большому   
счету, классы всего лишь добавляют дополнительный структурный уровень,   
позволяющий организовать функции и данные в виде единого объекта и обе-  
спечивающий возможность расширения.  
Мы также изменили программный код самопроверки – теперь он выводит не   
отдельные атрибуты объектов, а объекты целиком. Если теперь запустить этот   
сценарий, мы получим более понятные и осмысленные результаты – функции   
print автоматически будут вызывать наш новый метод \_\_str\_\_, возвращающий   
строки вида «[...]»:  
[Person: Bob Smith, 0]  
[Person: Sue Jones, 100000]  
Smith Jones  
[Person: Sue Jones, 110000]  
Несколько важных замечаний: как мы узнаем в следующей главе, родствен-  
ный метод \_\_repr\_\_ перегрузки операторов возвращает представление объек-  
та в виде программного кода. Иногда классы переопределяют оба метода: \_\_  
str\_\_ – для отображения объектов в удобочитаемом формате, для пользователя,   
и \_\_repr\_\_ – для вывода дополнительных сведений об объектах, которые могут   
представлять интерес для разработчика. Поскольку операция вывода автома-  
тически вызывает метод \_\_str\_\_, а интерактивная оболочка выводит результа-  
ты с помощью метода \_\_repr\_\_, подходящие варианты вывода могут предостав-  
ляться обеим категориям потенциальных клиентов. Так как нас не интересует   
представление объектов в виде программного кода, нам вполне будет достаточ-  
но одного метода \_\_str\_\_.  
Шаг 4: адаптация поведения   
с помощью подклассов  
В настоящий момент в нашем классе задействовано большинство механизмов   
ООП, имеющихся в языке Python: класс создает экземпляры, обеспечивает   
особенности поведения с помощью методов и даже использует перегрузку опе-  
раторов, перехватывая операции вывода с помощью метода \_\_str\_\_. Он факти-  
чески объединяет логику и данные в единый самостоятельный программный   
компонент, упрощая поиск и модификацию программного кода, что может   
потребоваться в будущем. Возможность инкапсуляции также позволяет нам   
организовать программный код так, чтобы избежать избыточности и связан-  
ных с ней проблем при сопровождении.  
Единственное основное понятие ООП, которое еще не было задействовано, – это   
адаптация программного кода за счет наследования. В некотором смысле мы   
уже использовали наследование – экземпляры наследуют методы своего клас-  
са. Однако для демонстрации истинной мощи ООП нам следует определить от-  
ношение типа суперкласс/подкласс, которое позволит нам расширить возмож-  
ности нашего программного обеспечения и немного изменить унаследованное   
поведение. В конце концов, в этом заключается основная идея ООП – возмож-

740   
Глава 27. Более реалистичный пример   
ность адаптации уже имеющегося и действующего программного кода позво-  
ляет существенно сократить время, затрачиваемое на разработку.  
Создание подклассов  
В качестве следующего шага попробуем применить методологию ООП и адап-  
тировать наш класс Person, расширив иерархию объектов. Для этого мы опреде-  
лим подкласс с именем Manager, наследующий класс Person, в котором мы заме-  
стим унаследованный метод giveRaise более узкоспециализированной версией.   
Ниже приводится начало определения нашего нового класса:   
class Manager(Person): # Определить подкласс класса Person  
Это объявление означает, что мы определяем новый класс с именем Manager,   
который наследует и может адаптировать суперкласс Person. Говоря простыми   
словами, класс Manager во многом схож с классом Person (слишком долгое всту-  
пление для маленькой особенности...), но при этом класс Manager реализует свой   
способ увеличения зарплаты.  
Предположим, что из некоторых соображений менеджер (экземпляр класса   
Manager) получает не только прибавку, которая передается в виде процентов,   
как обычно, но еще и дополнительную премию, по умолчанию составляющую   
10%. Например, если прибавка к зарплате менеджера составляет 10%, то в дей-  
ствительно, зарплата будет увеличена на 20%. (Любое совпадение с реальными   
лицами, конечно, абсолютно случайно.) Наш новый метод начинается, как по-  
казано ниже, – поскольку переопределенный метод giveRaise в дереве насле-  
дования оказывается ближе к экземплярам класса Manager, чем оригинальная   
реализация в классе Person, он фактически замещает и тем самым адаптиру-  
ет операцию. Напомню, что согласно правилам, поиск в дереве наследования   
оканчивается, как только будет найден первый метод с подходящим именем:  
class Manager(Person): # Наследует атрибута класса Person  
 def giveRaise(self, percent, bonus=.10): # Переопределить для адаптации  
Расширение методов: неправильный способ  
Далее, у нас имеется два способа адаптации программного кода в классе Manag-  
er: правильный и неправильный. Начнем с неправильного способа, потому что   
он проще для понимания. Неправильный способ заключается в простом копи-  
ровании реализации метода giveRaise из класса Person и его изменении в классе   
Manager, как показано ниже:  
class Manager(Person):  
 def giveRaise(self, percent, bonus=.10):  
 self.pay = int(self.pay \* (1 + percent + bonus)) # Неправильно:   
 # копирование  
Этот метод будет действовать, как и предполагалось, – когда позднее мы будем   
вызывать метод giveRaise относительно экземпляра класса Manager, будет вы-  
полняться адаптированная версия, которая добавляет дополнительную пре-  
мию. Но что же здесь неправильного?  
Проблема здесь самая обычная: всякий раз, когда вы копируете программный   
код, вы фактически усложняете его сопровождение в будущем. Представьте   
себе: из-за того, что мы скопировали оригинальную версию, нам придется из-  
менять программный код уже не в одном, а в двух местах, если позднее нам по-

Шаг 4: адаптация поведения с помощью подклассов   
741  
требуется (а это наверняка произойдет) изменить способ увеличения зарплаты.   
Это достаточно маленький и достаточно искусственный пример, тем не менее   
он наглядно демонстрирует общую проблему – всякий раз, когда у вас появ-  
ляется соблазн скопировать программный код, вам наверняка стоит поискать   
более правильный подход.  
Расширение методов: правильный способ   
В действительности нам требуется лишь дополнить оригинальный метод   
giveRaise, а не заменить его полностью. Правильный способ состоит в том, что-  
бы вызвать оригинальную версию с измененными аргументами, как показано   
ниже:  
class Manager(Person):  
 def giveRaise(self, percent, bonus=.10):  
 Person.giveRaise(self, percent + bonus) # Правильно: дополняет   
 # оригинал  
Данная реализация учитывает то обстоятельство, что методы класса могут   
вызываться либо обращением к экземпляру (обычный способ, когда интерпре-  
татор автоматически передает экземпляр в аргументе self), либо обращением   
к классу (менее распространенный способ, когда экземпляр передается вруч-  
ную). Вспомним, что вызов метода:  
instance.method(args...)  
автоматически транслируется интерпретатором в эквивалентную форму:  
class.method(instance, args...)  
где класс, содержащий вызываемый метод, определяется в соответствии   
с теми правилами поиска в дереве наследования, которые действуют и для ме-  
тодов. В своих сценариях вы можете использовать любую форму вызова, но не   
забывайте о различиях между ними – при обращении непосредственно к клас-  
су вы должны передавать объект экземпляра вручную. Метод всегда должен   
получать объект экземпляра тем или иным способом, однако интерпретатор   
обеспечивает автоматическую его передачу только при вызове метода через об-  
ращение к экземпляру. При вызове метода через обращение к классу вы сами   
должны передавать экземпляр в аргументе self – внутри метода, такого как   
giveRaise, аргумент self уже содержит подразумеваемый объект вызова, то есть   
сам экземпляр.  
Вызов через обращение к классу фактически отменяет поиск в дереве насле-  
дования, начиная с экземпляра, и запускает поиск, начиная с определенного   
класса и выше по дереву классов. В нашем случае мы можем использовать этот   
прием для вызова метода giveRaise по умолчанию, находящегося в классе Per-  
son, невзирая на то, что он был переопределен в классе Manager. Строго говоря,   
мы должны были вызвать этот метод именно через обращение к классу Person,   
потому что инструкция self.giveRaise() внутри метода giveRaise класса Manager   
привела бы к рекурсии – так как self уже является экземпляром класса Manager,   
инструкция self.giveRaise() интерпретировалась бы, как вызов метода Manager.  
giveRaise, и так далее, и так далее, пока не исчерпалась бы доступная память.  
На первый взгляд «правильная» версия мало чем отличается от предыдущей,   
«неправильной», но это может иметь огромное значение для  сопровождения   
в будущем – поскольку основная логика работы метода giveRaise теперь нахо-

742   
Глава 27. Более реалистичный пример   
дится только в одном месте (метод класса Person), в случае необходимости нам   
придется изменять всего одну версию. И действительно, такой способ расшире-  
ния метода более четко отражает наши намерения – нам требуется выполнить   
стандартную операцию giveRaise и просто добавить дополнительную премию.   
Ниже приводится полное содержимое файла модуля после выполнения этого   
последнего шага:  
# Добавлен подкласс, адаптирующий поведение суперкласса  
   
class Person:  
 def \_\_init\_\_(self, name, job=None, pay=0):  
 self.name = name  
 self.job = job  
 self.pay = pay  
 def lastName(self):  
 return self.name.split()[-1]  
 def giveRaise(self, percent):  
 self.pay = int(self.pay \* (1 + percent))  
 def \_\_str\_\_(self):  
 return ‘[Person: %s, %s]’ % (self.name, self.pay)  
   
class Manager(Person):  
 def giveRaise(self, percent, bonus=.10): # Переопределение метода  
 Person.giveRaise(self, percent + bonus) # Вызов версии из   
 # класса Person  
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 bob = Person(‘Bob Smith’)  
 sue = Person(‘Sue Jones’, job=’dev’, pay=100000)  
 print(bob)  
 print(sue)  
 print(bob.lastName(), sue.lastName())  
 sue.giveRaise(.10)  
 print(sue)  
 tom = Manager(‘Tom Jones’, ‘mgr’, 50000) # Экземпляр Manager: \_\_init\_\_  
 tom.giveRaise(.10) # Вызов адаптированной версии  
 print(tom.lastName()) # Вызов унаследованного метода  
 print(tom) # Вызов унаследованного \_\_str\_\_  
Чтобы проверить работу класса Manager, мы добавили код самопроверки, ко-  
торый создает экземпляр класса Manager, вызывает его методы и выводит этот   
экземпляр. Ниже приводятся результаты работы обновленной версии модуля:  
[Person: Bob Smith, 0]  
[Person: Sue Jones, 100000]  
Smith Jones  
[Person: Sue Jones, 110000]  
Jones  
[Person: Tom Jones, 60000]  
Все выглядит совсем неплохо: результаты тестирования с участием объектов   
bob и sue выглядят, как и прежде, а когда для экземпляра tom класса Manager   
производится повышение зарплаты на 10%, действительное повышение со-  
ставляет 20% (его зарплата увеличилась с $50K до $60K), потому что адапти-  
K до $60K), потому что адапти-  
 до $60K), потому что адапти-  
K), потому что адапти-  
), потому что адапти-  
рованная версия метода giveRaise в классе Manager вызывается только для этого   
объекта. Обратите также внимание, что при выводе информации об объекте tom   
используется форматирование, определенное в методе \_\_str\_\_ Person: экземпля-

Шаг 4: адаптация поведения с помощью подклассов   
743  
ры класса Manager наследуют его, а также методы lastName и \_\_init\_\_ от класса   
Person.  
Полиморфизм в действии  
Чтобы еще более полно задействовать механизм наследования, мы можем до-  
бавить в конец файла следующий программный код:  
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 ...  
 print(‘--All three--’)  
 for object in (bob, sue, tom): # Обработка объектов обобщенным способом  
 object.giveRaise(.10) # Вызовет метод giveRaise этого объекта  
 print(object) # Вызовет общий метод \_\_str\_\_  
Ниже приводятся результаты его работы:  
[Person: Bob Smith, 0]  
[Person: Sue Jones, 100000]  
Smith Jones  
[Person: Sue Jones, 110000]  
Jones  
[Person: Tom Jones, 60000]  
--All three—  
[Person: Bob Smith, 0]  
[Person: Sue Jones, 121000]  
[Person: Tom Jones, 72000]  
В добавленном программном коде переменная object может ссылаться либо на   
экземпляр класса Person, либо на экземпляр класса Manager, а интерпретатор   
автоматически вызовет соответствующий метод giveRaise – для объектов bob   
и sue будет вызвана оригинальная версия метода из класса Person, а для объ-  
екта tom – адаптированная версия из класса Manager. Проследите сами, как ин-  
терпретатор выбирает нужную версию метода giveRaise для каждого объекта.  
Этот пример демонстрирует действие понятия полиморфизма в языке Python,   
с которым мы познакомились ранее, – действие операции giveRaise зависит от   
того, к какому объекту она применяется. Проявление полиморфизма особенно   
очевидно, когда его можно наблюдать на примере выбора метода из классов, на-  
писанных нами. Так как выбор версии метода giveRaise основывается на типе   
объекта, в результате sue получает прибавку в 10%, а tom – прибавку в 20%.   
Как мы уже знаем, полиморфизм составляет основную долю гибкости языка   
Python. Передача любого из трех объектов в функцию, вызывающую метод   
giveRaise, например, привела бы к тем же самым результатам: в зависимости   
от типа полученного объекта автоматически была бы вызвана соответствую-  
щая версия метода.  
С другой стороны, операция вывода вызывает одну  и  ту  же  версию метода   
\_\_str\_\_ для всех трех объектов, потому что в программном коде присутствует   
только одна его версия – в классе Person. Класс Manager может не только адап-  
тировать, но и использовать оригинальную реализацию в классе Person. Не-  
смотря на небольшой объем, этот пример наглядно демонстрирует широкие   
возможности ООП в адаптации и многократном использовании программного   
кода – при использовании классов это в большинстве случаев выполняется ав-  
томатически.

744   
Глава 27. Более реалистичный пример   
Наследование, адаптация и расширение  
В действительности классы могут обладать еще более высокой гибкостью, чем   
можно было бы предположить, исходя из нашего примера. В общем случае   
классы могут наследовать, адаптировать и расширять существующую реали-  
зацию суперклассов. В нашем примере мы все свое внимание сосредоточили на   
адаптации имеющегося программного кода, однако мы точно так же могли бы   
добавить в класс Manager уникальные методы, отсутствующие в классе Person,   
если бы для класса Manager потребовалось реализовать нечто совсем иное (что   
естественно вытекает из имени класса). Такую возможность иллюстрирует   
следующий фрагмент. Здесь метод giveRaise переопределяет метод суперклас-  
са, адаптируя его, а метод someThingElse является совершенно новым дополне-  
нием к классу Manager:  
class Person:  
 def lastName(self): ...  
 def giveRaise(self): ...  
 def \_\_str\_\_(self): ...  
   
class Manager(Person): # Наследование  
 def giveRaise(self, ...): ... # Адаптация  
 def someThingElse(self, ...): ... # Расширение  
   
tom = Manager()  
tom.lastName() # Унаследованный метод  
tom.giveRaise() # Адаптированная версия  
tom.someThingElse() # Дополнительный метод  
print(tom) # Унаследованный метод перегрузки  
Дополнительные методы, такие как метод someThingElse в этом примере, рас-  
ширяют возможности существующего программного обеспечения и доступ-  
ны только для объектов класса Manager. Для нужд обучения мы ограничились   
адаптацией некоторых методов класса Person за счет их переопределения, но   
отказались от добавления новых методов.  
ООП: основная идея  
Несмотря на небольшой объем, наш программный код достаточно функциона-  
лен. И в действительности он иллюстрирует основное преимущество ООП: ис-  
пользуя объектно-ориентированный стиль, мы адаптируем имеющийся про-  
граммный код, а не копируем и не изменяем его. Это преимущество не всегда   
очевидно для начинающих программистов, особенно на фоне дополнительных   
требований, предъявляемых при создании классов. Но в целом применение   
объектно-ориентированного стиля программирования способно существенно   
сократить время разработки, по сравнению с другими подходами.  
Так, в нашем примере мы теоретически могли бы реализовать отдельную опера-  
цию giveRaise, не прибегая к созданию подкласса, но ни один из других способов   
не позволил бы нам получить настолько же оптимальный программный код:  
 •  
Мы могли бы создать совершенно новый, независимый класс Manager, но при   
этом нам пришлось бы повторно реализовать все методы, уже присутствую-  
щие в классе Person и действующие одинаково в классе Manager.  
 •  
Мы могли бы просто изменить существующий класс Person, чтобы удовлет-  
ворить требованиям, предъявляемым к методу giveRaise класса Manager, но

Шаг 5: адаптация конструкторов   
745  
при этом нарушилась бы корректная работа там, где требуется оригиналь-  
ное поведение класса Person.   
 •  
Мы могли бы просто скопировать класс Person целиком, присвоить копии   
имя Manager и изменить метод giveRaise, но при этом наш программный код   
стал бы избыточным, что усложнило бы его сопровождение в будущем –   
изменения в классе Person не будут автоматически отражаться на классе   
Manager, и нам придется вручную переносить эти изменения в реализацию   
класса Manager. Прием, основанный на копировании, может показаться са-  
мым быстрым, но он удваивает объем работы, которую придется проделы-  
вать в будущем.  
Адаптируемые иерархии, которые мы можем конструировать с помощью клас-  
сов, обеспечивают более оптимальное решение для программного обеспечения,   
которое предполагается развивать в течение длительного времени. Никакие   
другие средства языка Python не поддерживают подобный режим разработки.   
Благодаря тому, что мы можем адаптировать и расширять наши предыдущие   
наработки с помощью новых подклассов, мы можем использовать то, что уже   
действует, и не создавать каждый раз все заново, ломая то, что уже работает,   
или добавляя множество копий программного кода, которые придется обнов-  
лять в будущем. При правильном применении ООП становится сильным союз-  
ником программиста.  
Шаг 5: адаптация конструкторов  
Наш программный код действует так, как он действует, но если вы вниматель-  
нее изучите текущую версию, вы можете заметить кое-что непонятное – ка-  
жется бессмысленным указывать значение ‘mgr’ (менеджер) в аргументе job   
(должность) при создании объекта класса Manager: эта должность уже подразу-  
мевается названием класса. Было бы лучше заполнять этот атрибут автомати-  
чески, при создании экземпляра класса Manager.  
Для этого мы можем проделать тот же трюк, что и в предыдущем разделе: мы   
можем адаптировать логику работы конструктора в классе Manager так, чтобы   
он автоматически подставлял название должности. С точки зрения реализа-  
ции, нам необходимо переопределить метод \_\_init\_\_ в классе Manager, чтобы он   
подставлял строку ‘mgr’ автоматически. Как и при адаптации метода giveRaise,   
нам также необходимо вызывать оригинальный метод \_\_init\_\_ из класса Person   
за счет обращения к имени класса, чтобы инициализировать остальные атри-  
буты объекта.  
В новой версии сценария, которая приводится ниже, мы создали новый кон-  
структор для класса Manager и изменили вызов, создающий объект tom, – теперь   
мы не передаем ему название должности ‘mgr’:  
# Добавлен адаптированный конструктор в подкласс  
   
class Person:  
 def \_\_init\_\_(self, name, job=None, pay=0):  
 self.name = name  
 self.job = job  
 self.pay = pay  
 def lastName(self):  
 return self.name.split()[-1]  
 def giveRaise(self, percent):

746   
Глава 27. Более реалистичный пример   
 self.pay = int(self.pay \* (1 + percent))  
 def \_\_str\_\_(self):  
 return ‘[Person: %s, %s]’ % (self.name, self.pay)  
   
class Manager(Person):  
 def \_\_init\_\_(self, name, pay): # Переопределенный конструктор  
 Person.\_\_init\_\_(self, name, ‘mgr’, pay) # Вызов оригинального   
 # конструктора со значением   
 # ‘mgr’ в аргументе job  
 def giveRaise(self, percent, bonus=.10):  
 Person.giveRaise(self, percent + bonus)  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 bob = Person(‘Bob Smith’)  
 sue = Person(‘Sue Jones’, job=’dev’, pay=100000)  
 print(bob)  
 print(sue)  
 print(bob.lastName(), sue.lastName())  
 sue.giveRaise(.10)  
 print(sue)  
 tom = Manager(‘Tom Jones’, 50000) # Указывать должность не требуется:  
 tom.giveRaise(.10) # Подразумевается/устанавливается   
 print(tom.lastName()) # классом  
 print(tom)  
Здесь мы снова использовали тот же прием расширения конструктора \_\_init\_\_,   
который выше использовался для расширения метода giveRaise, – вызвали вер-  
сию метода из суперкласса обращением к имени класса и явно передали экзем-  
пляр self. Несмотря на странный вид имени конструктора, конечный эффект   
получается тот же самый. Так как нам необходимо задействовать логику кон-  
структора класса Person (чтобы инициализировать атрибуты экземпляра), мы   
должны вызвать его именно так, как показано в примере, в противном случае   
экземпляры класса Manager окажутся без атрибутов.  
Такая форма вызова конструктора суперкласса из конструктора подкласса ши-  
роко используется при программировании на языке Python. Механизм насле-  
Python. Механизм насле-  
. Механизм насле-  
дования, реализованный в интерпретаторе, позволяет отыскать только один   
метод \_\_init\_\_ на этапе конструирования – самый нижний в дереве классов.   
Если во время конструирования объекта требуется вызвать метод \_\_init\_\_, рас-  
положенный выше (что обычно и делается), его необходимо вызывать вручную,   
обращением через имя суперкласса. Положительная сторона такого подхода   
заключается в том, что вы можете явно передать конструктору суперкласса   
только необходимые аргументы или вообще не вызывать его: возможность от-  
каза от вызова конструктора суперкласса позволяет полностью заместить ло-  
гику его работы, а не доподнять ее.  
В процессе работы эта версия сценария выводит ту же информацию, что и пре-  
жде, – мы не изменили логику его работы, мы просто реструктурировали про-  
граммный код, чтобы избавиться от некоторой избыточности:  
[Person: Bob Smith, 0]  
[Person: Sue Jones, 100000]  
Smith Jones  
[Person: Sue Jones, 110000]  
Jones  
[Person: Tom Jones, 60000]

Шаг 5: адаптация конструкторов   
747  
ООП проще, чем может показаться  
В своем законченном виде, несмотря на незначительные размеры, наши клас-  
сы задействовали практически все наиболее важные концепции механизма   
ООП в языке Python:  
 •  
Создание экземпляров – заполнение атрибутов экземпляров.  
 •  
Методы, реализующие поведение, – инкапсуляция логики в методах класса.  
 •  
Перегрузка операторов – реализация поддержки встроенных операций, та-  
ких как вывод.  
 •  
Адаптация поведения – переопределение специализированных версий ме-  
тодов в подклассах.  
 •  
Адаптация конструкторов – добавление логики инициализации, в допол-  
нение к логике суперкласса.  
Большая часть этих концепций основана на трех простых механизмах: поиске   
атрибутов в дереве наследования, специальном аргументе self методов и авто-  
матическом выборе нужного метода перегрузки операторов.  
Попутно мы также упростили возможность изменения нашего программного   
кода в будущем, использовав склонность классов к многократному использо-  
ванию программного кода для снижения избыточности. Например, мы офор-  
мили логику работы классов в виде методов и предусмотрели вызов методов   
суперкласса, чтобы избежать появления нескольких копий одного и того же   
программного кода. Большинство этих действий естественным образом проис-  
текает из мощных возможностей классов в структурировании программного   
кода.  
Вообще говоря, это все, что составляет основу ООП в языке Python. Конечно,   
классы могут быть гораздо больше, чем в данном примере, а кроме того, су-  
ществует еще ряд дополнительных концепций классов, такие как декораторы   
и метаклассы, с которыми мы познакомимся в последующих главах. Однако,   
что касается основ, наши классы используют все базовые механизмы ООП.   
Фактически если вы разобрались с описанными здесь особенностями рабо-  
ты с классами, вы сможете самостоятельно понять большую часть объектно-  
ориентированного программного кода на языке Python.  
Другие способы комбинирования классов  
После всего вышесказанного я должен сообщить, что, несмотря на простоту   
ООП в языке Python, объединение классов в крупных программах – это уже   
отчасти искусство. В этой главе основное свое внимание мы сконцентрирова-  
ли на механизме наследования, потому что он поддерживается самим языком,   
но иногда программисты используют иные способы комбинирования классов.   
Например, очень часто используется прием вложения объектов друг в друга   
для создания составных объектов. Более детально мы исследуем этот прием   
в главе 30, которая посвящена скорее вопросам проектирования, чем языку   
Python, – однако вкратце замечу, что мы могли бы использовать этот прием   
при создании нашего класса Manager, вложив в него объект класса Person, а не   
наследуя этот класс.  
Следующая альтернативная реализация демонстрирует такую возможность,   
используя метод \_\_getattr\_\_ перегрузки операторов, с которым мы познако-

748   
Глава 27. Более реалистичный пример   
мимся в главе 29, чтобы перехватывать попытки обращения к несуществую-  
щим атрибутам и делегировать эти обращения вложенному объекту, вызовом   
встроенной функции getattr. Здесь также имеется адаптированная версия ме-  
тода giveRaise, которая изменяет значение аргумента, передаваемого методу   
вложенного объекта. В результате класс Manager превращается в контроллер,   
который вызывает методы вложенного объекта, а не методы суперкласса:  
# Альтернативная версия класса Manager с вложенным объектом  
   
class Person:  
 ...то же самое...  
   
class Manager:  
 def \_\_init\_\_(self, name, pay):  
 self.person = Person(name, ‘mgr’, pay) # Вложенный объект Person   
 def giveRaise(self, percent, bonus=.10): # Перехватывает и делегирует  
 self.person.giveRaise(percent + bonus)   
 def \_\_getattr\_\_(self, attr): # Делегирует обращения   
 return getattr(self.person, attr) # ко всем остальным атрибутам  
 def \_\_str\_\_(self):  
 return str(self.person) # Требуется перегрузка (в 3.0)  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 ...то же самое...  
В действительности, этот альтернативный вариант класса Manager представ-  
ляет достаточно распространенный шаблон проектирования, известный как   
делегирование, – составная структура служит оберткой вокруг вложенного   
объекта, управляет им и перенаправляет ему вызовы методов. Мы сумели реа-  
лизовать этот шаблон в нашем примере, но для этого потребовалось написать   
вдвое больше программного кода, и он не так хорошо удовлетворяет нашим по-  
требностям, как механизм наследования, позволяющий выполнять непосред-  
ственную адаптацию. (Фактически ни один здравомыслящий программист не   
стал бы применять этот шаблон для реализации нашего примера, кроме тех,   
кто пишет учебники.) В этой реализации класс Manager в действительности не   
наследует класс Person, поэтому нам пришлось написать дополнительный про-  
граммный код, который вручную вызывает необходимые методы вложенного   
объекта – методы перегрузки операторов, такие как \_\_str\_\_, потребовалось   
переопределить (по крайней мере, в Python 3.0, о чем дополнительно рассказы-  
Python 3.0, о чем дополнительно рассказы-  
 3.0, о чем дополнительно рассказы-  
вается во врезке «Перехват обращений к встроенным атрибутам в версии 3.0»   
ниже), а реализация дополнительных особенностей в классе Manager выглядит   
сложнее, потому что информация о состоянии находится уровнем ниже.  
И все же прием, основанный на использовании вложенных объектов, с успе-  
хом может использоваться на практике, особенно когда круг взаимодействий   
контейнера с вложенными объектами уже, чем предполагает прием адапта-  
ции. Уровень контроллера, который представляет альтернативная реализация   
класса Manager, например, может пригодиться для отслеживания и проверки   
вызовов методов других объектов (мы будем использовать практически иден-  
тичный прием при изучении декораторов классов, далее в этой книге). Кроме   
того, для объединения других объектов в виде множества можно было бы ис-  
пользовать гипотетический агрегатный класс Department, как показано ниже.   
Добавьте его реализацию в конец файла person.py, чтобы получить возмож-  
ность опробовать его самостоятельно:

Шаг 5: адаптация конструкторов   
749  
# Объединение объектов в составной объект  
   
...  
bob = Person(...)  
sue = Person(...)  
tom = Manager(...)  
   
class Department:  
 def \_\_init\_\_(self, \*args):  
 self.members = list(args)  
 def addMember(self, person):  
 self.members.append(person)  
 def giveRaises(self, percent):  
 for person in self.members:  
 person.giveRaise(percent)  
 def showAll(self):  
 for person in self.members:  
 print(person)  
   
development = Department(bob, sue) # Встраивание объектов в составной объект  
development.addMember(tom)  
development.giveRaises(.10) # Вызов метода giveRaise вложенных объектов  
development.showAll() # Вызов метода \_\_str\_\_ вложенных объектов  
Интересно отметить, что в этом примере используются оба приема, наследова-  
ние и встраивание, – объекты класса Department являются составными объек-  
тами, которые управляют другими встроенными объектами, но сами встроен-  
ные объекты классов Person и Manager используют механизм наследования для   
адаптации своего поведения. В качестве еще одного примера можно привести   
графический интерфейс пользователя, в реализации которого точно так же   
для адаптации поведения или внешнего вида кнопок и меток может исполь-  
зоваться механизм наследования, а для создания пакетов встроенных видже-  
тов, таких как формы ввода, калькуляторы и текстовые редакторы, – прием   
встраивания. Структура такого класса зависит от объектов, которые требует-  
ся смоделировать.  
Проблемы проектирования таких составных объектов рассматриваются в гла-  
ве 30, поэтому мы пока отложим дальнейшие исследования. Замечу еще раз,   
что наши классы Person и Manager демонстрируют применение всех основных   
механизмов ООП в языке Python. После овладения основами ООП разработка   
обобщенных инструментов для применения их в своих сценариях часто явля-  
ется естественным следующим этапом – и темой следующего раздела.  
Перехват обращений к встроенным атрибутам   
в версии 3.0  
В Python 3.0 (и в Python 2.6, если используются классы нового стиля)   
альтернативная реализация класса Manager, основанная на применении   
приема делегирования, которую мы только что создали, не в состоянии   
перехватывать и делегировать вызовы методов перегрузки операторов,   
таких как \_\_str\_\_, без их переопределения. Это общая проблема клас-  
сов, основанных на делегировании, хотя и известно, что в данном при-  
мере используется единственное имя \_\_str\_\_.

750   
Глава 27. Более реалистичный пример   
Напомню, что встроенные операции, например вывод и обращение   
к элементу по индексу, неявно вызывают методы перегрузки операто-  
ров, такие как \_\_str\_\_ и \_\_getitem\_\_. В версии 3.0 встроенные операции,   
подобные этим, не используют менеджеры атрибутов для неявного по-  
лучения ссылок на атрибуты: они не используют ни метод \_\_getattr\_\_   
(вызывается при попытке обращения к неопределенным атрибутам),   
ни родственный ему метод \_\_getattribute\_\_ (вызывается при обращении   
к любым атрибутам). Именно по этой причине нам потребовалось пере-  
определить метод \_\_str\_\_ в альтернативной реализации класса Manager,   
чтобы обеспечить вызов метода встроенного объекта Person при запуске   
сценария под управлением Python 3.0.  
Технически это обусловлено тем, что при работе с классическими клас-  
сами интерпретатор пытается искать методы перегрузки операторов   
в экземплярах, а при работе с классами нового стиля – нет. Он вообще   
пропускает экземпляр и пытается отыскать требуемый метод в клас-  
се. В версии 2.6 встроенные операции, при применении к экземплярам   
классических классов, выполняют  поиск  атрибутов обычным спо-  
собом. Например, операция вывода пытается отыскать метод \_\_str\_\_   
с помощью метода \_\_getattr\_\_. Однако в версии 3.0 классы нового стиля   
наследуют метод \_\_str\_\_ по умолчанию, что мешает работе метода \_\_ge-  
tattr\_\_, а метод \_\_getattribute\_\_ вообще не перехватывает обращения   
к подобным именам.  
Это проблема, но вполне преодолимая, – классы, опирающиеся на при-  
ем делегирования, в версии 3.0 в общем случае могут переопределять   
методы перегрузки операторов, чтобы делегировать вызовы вложенным   
объектам, либо вручную, либо с помощью других инструментов или   
суперклассов. Эта тема слишком сложная, чтобы развивать ее дальше   
в этой главе, поэтому не старайтесь сейчас уяснить все тонкости. Мы   
вернемся к проблеме управления атрибутами в главе 37, а потом еще раз   
рассмотрим ее в главе 38, на примере декораторов класса Private.  
Шаг 6: использование инструментов   
интроспекции  
Давайте добавим последний штрих, прежде чем сохраним наши объекты   
в базе данных. Наши классы имеют законченный вид и демонстрируют боль-  
шую часть основ ООП. Однако остаются еще две проблемы, которые мы долж-  
ны сгладить, прежде чем запустить эти классы в работу:  
 •  
Во-первых, если внимательно посмотреть на то, как сейчас наши объек-  
ты выводятся на экран, можно заметить, что объект tom, принадлежащий   
к классу Manager, помечается как объект класса Person. С технической точки   
зрения это не является ошибкой, так как класс Manager является адапти-  
рованной и специализированной версией класса Person. Однако более пра-  
Person. Однако более пра-  
. Однако более пра-  
вильным было бы отображать как можно более точное имя класса объекта   
(то есть имя самого нижнего класса в иерархии).  
 •  
Во-вторых, что, пожалуй, более важно, в текущей версии отображается   
информация только о тех атрибутах, которые мы явно указали в методе

Шаг 6: использование инструментов интроспекции   
751  
\_\_str\_\_, чего может оказаться недостаточно в будущем. Например, сейчас   
у нас нет возможности убедиться, что атрибут job в объекте tom получает   
значение ‘mgr’ в конструкторе класса Manager, потому что метод \_\_str\_\_,   
который реализован в классе Person, не выводит его. Более того, если мы   
когда-нибудь расширим или как-то иначе изменим набор атрибутов, кото-  
рым выполняется присваивание в методе \_\_init\_\_, мы должны будем так-  
же добавить вывод новых атрибутов в методе \_\_str\_\_, в противном случае   
результаты, возвращаемые этим методом, со временем перестанут соответ-  
ствовать действительности.  
Последний пункт означает, что мы снова добавляем себе лишнюю работу в бу-  
дущем за счет добавления избыточного программного кода. Поскольку любое   
несоответствие в методе \_\_str\_\_ будет отражаться на выводе программы, эта из-  
быточность может оказаться более очевидной, чем ее разновидности, которые   
мы устраняли ранее, – попытки избежать выполнения лишней работы в буду-  
щем вообще достойны поощрения.  
Специальные атрибуты классов  
Обе упомянутые проблемы можно решить с помощью инструментов интро-  
спекции, имеющихся в языке Python, – специальных атрибутов и функций,   
обеспечивающих доступ к внутренней реализации объектов. Это узкоспециа-  
лизированные инструменты, и обычно они используются программистами,   
создающими инструменты для других программистов, и гораздо реже про-  
граммистами, разрабатывающими приложения. Но даже учитывая это обсто-  
ятельство, знание основных приемов использования этих инструментов будет   
полезным, потому что они позволяют писать программный код, оперирующий   
классами обобщенными способами. В нашем случае, например, мы могли бы   
использовать две особенности, которые были представлены в конце предыду-  
щей главы:  
 •  
Встроенный атрибут instance.\_\_class\_\_ в экземпляре ссылается на класс   
этого экземпляра. Классы, в свою очередь, имеют атрибут \_\_name\_\_, подобно   
модулям, и последовательность \_\_bases\_\_, обеспечивающую доступ к супер-  
классам. Мы можем использовать эти атрибуты при выводе имени класса,   
к которому принадлежит экземпляр, вместо того, чтобы выводить жестко   
заданное имя.  
 •  
Встроенный атрибут object.\_\_dict\_\_ содержит словарь с парами ключ/зна-  
чение, каждая из которых соответствует определенному атрибуту в про-  
странстве имен объекта (включая модули, классы и экземпляры). Посколь-  
ку значением этого атрибута является словарь, мы можем получать из него   
список ключей, значения атрибутов по ключам, выполнять итерации по   
ключам и так далее, и тем самым обеспечить обобщенный способ обработки   
всех атрибутов. На основе этого словаря мы можем реализовать вывод всех   
атрибутов, имеющихся в любом экземпляре, а не только тех, которые явно   
будут указаны в методе вывода.  
Ниже приводится пример использования этих инструментов в интерактивном   
сеансе Python. Обратите внимание, что мы импортируем класс Person с помо-  
щью инструкции from, – имена классов располагаются в модулях и могут им-  
портироваться как обычные функции и другие переменные:  
>>> from person import Person  
>>> bob = Person(‘Bob Smith’)

752   
Глава 27. Более реалистичный пример   
>>> print(bob) # Вызов метод \_\_str\_\_ объекта bob  
[Person: Bob Smith, 0]  
   
>>> bob.\_\_class\_\_ # Выведет класс объекта bob и его имя  
<class ‘person.Person’>  
>>> bob.\_\_class\_\_.\_\_name\_\_  
‘Person’  
   
>>> list(bob.\_\_dict\_\_.keys()) # Атрибуты – это действительно ключи словаря  
[‘pay’, ‘job’, ‘name’] # Функция list используется для получения   
 # полного списка в версии 3.0  
>>> for key in bob.\_\_dict\_\_:  
 print(key, ‘=>’, bob.\_\_dict\_\_[key]) # Обращение по индексам  
   
pay => 0  
job => None  
name => Bob Smith  
   
>>> for key in bob.\_\_dict\_\_:  
 print(key, ‘=>’, getattr(bob, key)) # Аналогично выражению obj.attr,   
 # где attr - переменная  
pay => 0  
job => None  
name => Bob Smith  
Как уже отмечалось в предыдущей главе, некоторые атрибуты экземпляров   
могут отсутствовать в словаре \_\_dict\_\_, если класс экземпляра определяет   
атрибут \_\_slots\_\_, который является дополнительной и малопонятной особен-  
ностью классов нового стиля (и всех классов в Python 3.0), которая использу-  
Python 3.0), которая использу-  
 3.0), которая использу-  
ется для организации хранения атрибутов в виде массива и которую мы будем   
обсуждать в главах 30 и 31. Поскольку в действительности слоты принадлежат   
классам, а не экземплярам, и они очень редко используются на практике, мы   
можем просто игнорировать их в нашем примере и сосредоточиться на обыч-  
ном атрибуте \_\_dict\_\_.  
Обобщенный инструмент отображения  
Мы можем задействовать эти инструменты в суперклассе для точного отобра-  
жения имен классов и вывода значений всех атрибутов экземпляров любых   
классов. Создайте новый файл в своем текстовом редакторе и добавьте в него   
программный код, что приводится ниже, – это новый, независимый модуль   
с именем classtools.py, реализующий единственный класс. В методе \_\_str\_\_   
перегрузки операции вывода этого класса используются обобщенные инстру-  
менты интроспекции, поэтому он может работать с любыми экземплярами, не-  
зависимо от того, какими атрибутами они обладают. А так как это – класс, он   
автоматически превращается в обобщенный инструмент отображения: благо-  
даря наследованию, он может добавляться в любые классы, где требуется обе-  
спечить вывод данной информации. Как дополнительное преимущество, если   
нам когда-нибудь понадобится изменить формат вывода информации об экзем-  
плярах, достаточно будет изменить только этот класс, потому что все классы   
будут наследовать метод \_\_str\_\_ этого класса и автоматически будут использо-  
вать новый формат вывода:  
# Файл classtools.py (новый)  
“Различные утилиты и инструменты для работы с классами”

Шаг 6: использование инструментов интроспекции   
753  
class AttrDisplay:  
 “””  
 Реализует наследуемый метод перегрузки операции вывода, отображающий   
 имена классов экземпляров и все атрибуты в виде пар имя=значение,  
 имеющиеся в экземплярах (исключая атрибуты, унаследованные от классов).   
 Может добавляться в любые классы и способен работать с любыми   
 экземплярами.  
 “””  
 def gatherAttrs(self):  
 attrs = []  
 for key in sorted(self.\_\_dict\_\_):  
 attrs.append(‘%s=%s’ % (key, getattr(self, key)))  
 return ‘, ‘.join(attrs)  
 def \_\_str\_\_(self):  
 return ‘[%s: %s]’ % (self.\_\_class\_\_.\_\_name\_\_, self.gatherAttrs())  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 class TopTest(AttrDisplay):  
 count = 0  
 def \_\_init\_\_(self):  
 self.attr1 = TopTest.count  
 self.attr2 = TopTest.count+1  
 TopTest.count += 2  
   
 class SubTest(TopTest):  
 pass  
   
 X, Y = TopTest(), SubTest()  
 print(X) # Выведет все атрибуты экземпляра  
 print(Y) # Выведет имя класса,   
 # самого близкого в дереве наследования  
Обратите внимание на строки документирования – так как мы создаем много-  
целевой инструмент, вполне естественно снабдить его описанием для потенци-  
альных пользователей. Как мы видели в главе 15, строки документирования   
могут помещаться в начало простых функций и модулей, однако точно так же   
они могут помещаться в начало классов и их методов – эти строки автоматиче-  
ски извлекаются и отображаются функцией help и инструментом PyDoc (мы   
еще вернемся к строкам документирования в главе 28).  
Если запустить этот модуль, как самостоятельный сценарий, его программный   
код самопроверки создаст два экземпляра и выведет их – реализованный здесь   
метод \_\_str\_\_ выведет имена классов экземпляров и все их атрибуты в виде пар   
имя�значение, отсортированные по именам атрибутов в алфавитном порядке:  
C:\misc> classtools.py  
[TopTest: attr1=0, attr2=1]  
[SubTest: attr1=2, attr2=3]  
Атрибуты экземпляров и атрибуты классов  
Если внимательно изучить программный код самопроверки в модуле class-  
tools, можно заметить, что реализованный нами класс отображает только   
атрибуты  экземпляров, присоединенные непосредственно к объекту, рас-  
положенному в самом низу дерева наследования, – то есть те, что содержатся   
в атрибуте \_\_dict\_\_ объекта self. Как результат, мы не получаем информации   
об атрибутах, унаследованных экземплярами от классов, находящихся выше

754   
Глава 27. Более реалистичный пример   
в дереве (таких как атрибут count в этом примере). Унаследованные атрибуты   
класса присоединяются только к объекту класса, и не повторяются в экзем-  
плярах.  
Если вам потребуется добавить вывод унаследованных атрибутов, вы можете   
с помощью ссылки \_\_class\_\_ получить доступ к классу экземпляра и извлечь   
из его словаря \_\_dict\_\_ атрибуты класса, а затем выполнить итерации через   
содержимое атрибута \_\_bases\_\_ класса, чтобы подняться до уровня суперклас-  
сов (настолько высоко, насколько это потребуется). Если вы предпочитаете не   
усложнять программный код, вместо атрибута \_\_dict\_\_ можно вызвать встро-  
енную функцию dir, передав ей экземпляр, и получить тот же результат, пото-  
му что функция dir включает в результат унаследованные имена и возвращает   
его в виде отсортированного списка:  
>>> from person import Person  
>>> bob = Person(‘Bob Smith’)  
   
# В Python 2.6:  
   
>>> bob.\_\_dict\_\_.keys() # Только атрибуты экземпляра  
[‘pay’, ‘job’, ‘name’]  
   
>>> dir(bob) # + унаследованные атрибуты классов  
[‘\_\_doc\_\_’, ‘\_\_init\_\_’, ‘\_\_module\_\_’, ‘\_\_str\_\_’, ‘giveRaise’, ‘job’,  
‘lastName’, ‘name’, ‘pay’]  
   
# В Python 3.0:  
   
>>> list(bob.\_\_dict\_\_.keys()) # В 3.0 метод keys возвращает представление,   
[‘pay’, ‘job’, ‘name’] # а не список  
   
>>> dir(bob) # В 3.0 включаются методы типа класса  
[‘\_\_class\_\_’, ‘\_\_delattr\_\_’, ‘\_\_dict\_\_’, ‘\_\_doc\_\_’, ‘\_\_eq\_\_’, ‘\_\_format\_\_’,  
‘\_\_ge\_\_’, ‘\_\_getattribute\_\_’, ‘\_\_gt\_\_’, ‘\_\_hash\_\_’, ‘\_\_init\_\_’, ‘\_\_le\_\_’,  
...часть строк опущена...  
‘\_\_setattr\_\_’, ‘\_\_sizeof\_\_’, ‘\_\_str\_\_’, ‘\_\_subclasshook\_\_’, ‘\_\_weakref\_\_’,  
‘giveRaise’, ‘job’, ‘lastName’, ‘name’, ‘pay’]  
Вывод в версиях Python 2.6 и 3.0 отличается, потому что в версии 3.0 возвра-  
Python 2.6 и 3.0 отличается, потому что в версии 3.0 возвра-  
 2.6 и 3.0 отличается, потому что в версии 3.0 возвра-  
щаемое значение метода dict.keys не является списком и функция dir в Py-  
Py-  
thon 3.0 возвращает дополнительные атрибуты, реализованные в типе класса.   
Если говорить точнее, в версии 3.0 функция dir возвращает большее количе-  
ство имен, потому что в этой версии все классы относятся к классам «нового   
стиля» и наследуют множество методов перегрузки операторов из типа класса.   
На практике вам, скорее всего, потребуется отфильтровать большую часть ме-  
тодов, с именами вида \_\_X\_\_, при использовании функции dir в версии 3.0, так   
как они относятся к особенностям внутренней реализации классов и не имеют   
прямого отношения к информации, которую обычно требуется вывести.  
Для экономии пространства в книге мы оставим вам в качестве упражнения   
реализацию вывода унаследованных атрибутов класса посредством итераций   
через дерево наследования или с помощью функции dir. Дополнительные под-  
сказки по этой теме вы можете найти в модуле classtree.py, реализующем обход   
дерева наследования, который мы создадим в главе 28, а также в модуле lister.  
py, реализующем вывод списка атрибутов, который мы создадим в главе 30.

Шаг 6: использование инструментов интроспекции   
755  
Выбор имен в инструментальных классах  
И последняя тонкость: поскольку наш класс AttrDisplay в модуле classtools   
является обобщенным инструментом, предназначенным служить суперклас-  
сом для любых других классов, мы должны помнить о возможности непред-  
намеренных конфликтов имен с клиентскими классами. Предполагается, что   
в клиентских подклассах будут использоваться оба метода, \_\_str\_\_ и gatherAt-  
trs, но может оказаться так, что последний из них не будет соответствовать   
ожиданиям подклассов, – если в подклассе по неосторожности будет переопре-  
делено имя gatherAttrs, это наверняка нарушит работу нашего класса, потому   
что вместо нашей версии будет использоваться версия метода, реализованная   
в подклассе.  
Чтобы убедиться в этом, добавим реализацию метода gatherAttrs в класс TopTest   
в программном коде самопроверки – если только новый метод не будет иденти-  
чен оригинальной версии, наш инструментальный класс не будет давать ожи-  
даемых результатов:  
class TopTest(AttrDisplay):  
 ....  
 def gatherAttrs(self): # Replaces method in AttrDisplay!  
 return ‘Spam’  
Это необязательно плохо – иногда бывает желательно создать другие методы,   
доступные в подклассах, либо для непосредственного использования, либо для   
адаптации. Однако если в действительности нам требуется предоставить толь-  
ко метод \_\_str\_\_, тогда такой подход нельзя назвать идеальным.  
Чтобы минимизировать вероятность конфликта имен, как в данном случае,   
программисты на языке Python часто добавляют символ подчеркивания в на-  
чало имени метода, не предназначенного для использования за пределами   
класса: \_gatherAttrs в нашем случае. Этот прием не избавляет нас полностью от   
ошибок (что, если в другом классе также будет определен метод \_gatherAttrs?),   
но обычно этого бывает достаточно, а кроме того – это общепринятое соглаше-  
ние об именовании внутренних методов классов в языке Python.  
Лучшее, но реже используемое решение состоит в том, чтобы добавить два сим-  
вола подчеркивания в начало имени метода: \_\_gatherAttrs. Интерпретатор ав-  
томатически дополняет такие имена, включая в них имя вмещающего класса,   
что обеспечивает им истинную уникальность. Эту особенность обычно называ-  
ют псевдочастные атрибуты класса. Мы будем рассматривать ее в главе 30.   
А пока оставим оба метода общедоступными.  
Окончательные версии наших классов  
Теперь, чтобы воспользоваться этим обобщенным инструментом в наших клас-  
сах, достаточно будет импортировать его из модуля, добавить в список насле-  
дуемых классов нашего базового класса и удалить из него реализацию мето-  
да \_\_str\_\_, которая была создана ранее. Новая реализация метода перегрузки   
операции вывода будет унаследована экземплярами классов Person и Manager –   
класс Manager унаследует метод \_\_str\_\_ от класса Person, который в свою очередь   
унаследует ее от класса AttrDisplay, реализованного в другом модуле. Ниже   
приводится окончательная версия нашего модуля person.py со всеми измене-  
ниями:

756   
Глава 27. Более реалистичный пример   
# Файл person.py (окончательная версия)  
   
from classtools import AttrDisplay # Импортирует обобщенный инструмент  
   
class Person(AttrDisplay):  
 “””  
 Создает и обрабатывает записи с информацией о людях  
 “””  
 def \_\_init\_\_(self, name, job=None, pay=0):  
 self.name = name  
 self.job = job  
 self.pay = pay  
 def lastName(self): # Предполагается, что фамилия   
 return self.name.split()[-1] # указана последней  
 def giveRaise(self, percent): # Процент – величина в диапазоне 0..1  
 self.pay = int(self.pay \* (1 + percent))  
   
class Manager(Person):  
 “””  
 Версия класса Person, адаптированная в соответствии   
 со специальными требованиями  
 “””  
 def \_\_init\_\_(self, name, pay):  
 Person.\_\_init\_\_(self, name, ‘mgr’, pay)  
 def giveRaise(self, percent, bonus=.10):  
 Person.giveRaise(self, percent + bonus)  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 bob = Person(‘Bob Smith’)  
 sue = Person(‘Sue Jones’, job=’dev’, pay=100000)  
 print(bob)  
 print(sue)  
 print(bob.lastName(), sue.lastName())  
 sue.giveRaise(.10)  
 print(sue)  
 tom = Manager(‘Tom Jones’, 50000)  
 tom.giveRaise(.10)  
 print(tom.lastName())  
 print(tom)  
Так как это окончательная версия, мы добавили в нее несколько коммента-  
риев, описывающих наши классы, – строки документирования с функцио-  
нальным описанием и небольшие примечания, начинающиеся с символа #,   
в соответствии общепринятыми соглашениями. Если теперь запустить этот   
сценарий, он выведет все атрибуты наших объектов, а не только те, что ранее   
были явно указаны в оригинальной версии метода \_\_str\_\_. Наша последняя   
проблема была разрешена: так как реализация класса AttrDisplay извлекает   
имя класса непосредственно из экземпляра, для каждого объекта выводится   
имя ближайшего к нему класса – для объекта tom теперь выводится имя класса   
Manager, а не Person, и мы, наконец-то, можем убедиться, что атрибут job был   
корректно инициализирован в конструкторе класса Manager:  
C:\misc> person.py  
[Person: job=None, name=Bob Smith, pay=0]  
[Person: job=dev, name=Sue Jones, pay=100000]  
Smith Jones  
[Person: job=dev, name=Sue Jones, pay=110000]

Шаг 7 (последний): сохранение объектов в базе данных   
757  
Jones  
[Manager: job=mgr, name=Tom Jones, pay=60000]  
На этот раз сценарий выводит больше полезной информации, чем прежде.   
С точки зрения отдаленной перспективы наш класс, реализующий вывод атри-  
бутов, можно рассматривать как обобщенный  инструмент, который можно   
использовать как суперкласс для любых других классов, обеспечивающий вы-  
вод атрибутов. Все клиенты, наследующие его, автоматически будут воспри-  
нимать все изменения, которые в дальнейшем будут производиться в нашем   
инструменте. Далее в этой книге мы познакомимся с еще более мощными кон-  
цепциями инструментальных классов, такими как декораторы и метаклассы.   
Наряду с инструментами интроспекции, имеющимися в языке Python, они по-  
Python, они по-  
, они по-  
зволяют писать программный код, расширяющий классы и управляющий ими   
структурированными и простыми в сопровождении способами.  
Шаг 7 (последний):   
сохранение объектов в базе данных  
К настоящему моменту мы почти закончили нашу работу. Теперь у нас имеется   
система из двух модулей, которая не только реализует поставленную задачу   
представления информации о людях, но и предоставляет обобщенный инстру-  
мент отображения атрибутов, который в будущем может использоваться нами   
и в других программах. Поместив функции и классы в модули, мы обеспечили   
возможность многократного их использования. А организовав программное   
обеспечение в виде классов, мы обеспечили возможность его расширения.  
Наши классы действуют так, как мы и задумывали, однако объекты, которые   
создаются с их помощью, не являются настоящими записями в базе данных.   
То есть по завершении программы все созданные экземпляры исчезнут – они   
являются обычными объектами в памяти компьютера и не сохраняются на   
устройствах долговременного хранения, например в файлах, поэтому их не   
удастся восстановить при следующем запуске программы. Однако, как оказы-  
вается, совсем не сложно организовать сохранение объектов с помощью такой   
особенности Python, как хранилище объектов, позволяющей восстанавливать   
объекты после того, как программа создаст их и завершит работу. На заключи-  
тельном шаге этого примера мы реализуем возможность сохранения объектов.  
Модули pickle, shelve и dbm  
Возможность сохранения объектов во всех версиях Python обеспечивают три   
модуля в стандартной библиотеке:  
pickle  
Преобразует произвольные объекты на языке Python в строку байтов и об-  
Python в строку байтов и об-  
 в строку байтов и об-  
ратно.  
dbm (в Python 2.6 называется anydbm)  
Реализует сохранение строк в файлах, обеспечивающих возможность обра-  
щения по ключу.  
shelve  
Использует первые два модуля, позволяя сохранять объекты в файлах-  
хранилищах, обеспечивающих возможность обращения по ключу.

758   
Глава 27. Более реалистичный пример   
Мы уже сталкивались с этими модулями в главе 9, когда изучали основы рабо-  
ты с файлами. Они реализуют мощные возможности сохранения данных. Мы   
не можем достаточно подробно останавливаться на особенностях этих модулей   
в данной книге, однако они настолько просты в обращении, что краткого введе-  
ния будет вполне достаточно, чтобы приступить к их использованию.  
Модуль pickle обеспечивает самые общие средства преобразования объектов:   
он способен превратить практически любой объект, находящийся в памяти,   
в строку байтов, которая затем может использоваться для восстановления ори-  
гинального объекта. Модуль pickle может обрабатывать почти все объекты,   
создаваемые вами, – списки, словари, вложенные комбинации из этих объ-  
ектов, а также экземпляры классов. Последнее особенно важно для нас, пото-  
му что эта возможность позволяет сохранять данные (атрибуты) и поведение   
(методы) – фактически эта комбинация эквивалентна «записям» и «програм-  
мам». Благодаря такой универсальности модуля pickle, он поможет нам избе-  
жать необходимости писать дополнительный программный код, реализующий   
создание и анализ текстовых файлов, содержащих наши объекты. Сохраняя   
объекты в файле в виде строк с помощью модуля pickle, вы фактически обе-  
спечиваете долговременное хранение этих объектов: позднее достаточно будет   
просто загрузить эти строки и восстановить из них оригинальные объекты.  
С помощью модуля pickle достаточно просто организовать сохранение объек-  
тов в простых файлах и загрузку их из файлов, однако модуль shelve обеспе-  
чивает дополнительные удобства, позволяя сохранять объекты, обработанные   
модулем pickle, по ключу. Модуль shelve преобразует объект в строку с помо-  
щью модуля pickle и сохраняет ее под указанным ключом в файле dbm. Позднее,   
когда это необходимо, модуль shelve извлекает строку по ключу и воссоздает   
оригинальный объект в памяти, опять же с помощью модуля pickle. На пер-  
вый взгляд все это кажется немного сложным, однако в программе обращение   
к объектам в хранилище выглядит как обращение к элементам словаря – вы   
обращаетесь к объекту по ключу, сохраняете его, выполняя присваивание по   
ключу, и можете использовать инструменты словарей, такие как len, in и dict.  
keys, чтобы получить дополнительную информацию. Модуль shelve автомати-  
чески отображает операции со словарем на объекты, хранящиеся в файле.  
Фактически единственное отличие между хранилищами объектов и обычны-  
ми словарями состоит в том, что хранилища необходимо предварительно от-  
крывать, а затем закрывать их после внесения изменений. Таким образом,   
хранилища можно рассматривать, как простейшие базы данных, позволяю-  
щие сохранять и извлекать объекты по ключу и тем самым обеспечивающие   
сохранность объектов между запусками программы. Хранилища не поддержи-  
вают возможность выполнения запросов, например, на языке SQL, и испыты-  
вают недостаток дополнительных возможностей, которыми обладают разви-  
тые базы данных (такие как обработка транзакций), однако объекты, находя-  
щиеся в хранилище, способны использовать всю широту возможностей языка   
Python после того, как они будут извлечены обратно.  
Сохранение объектов в хранилище  
Обсуждение модулей pickle и shelve – это достаточно сложная тема, поэтому   
мы не будем углубляться во все их особенности. Более подробную информацию   
о них вы сможете получить в руководствах по стандартной библиотеке, а также   
в книгах прикладного характера в таких как «Программирование на Python».

Шаг 7 (последний): сохранение объектов в базе данных   
759  
При этом описать работу с ними на языке Python проще, чем на обычном языке   
человеческого общения, поэтому давайте перейдем к программному коду.   
Напишем новый сценарий, который сохраняет экземпляры наших классов   
в хранилище. В своем текстовом редакторе создайте новый файл makedb.py.   
Так как это новый файл, в него необходимо импортировать наши классы, что-  
бы с их помощью создать несколько экземпляров для последующего сохране-  
ния. Ранее для загрузки класса в интерактивную оболочку мы использовали   
инструкцию from, но в действительности существует два способа загрузки клас-  
сов из модулей, так же как функций и других переменных (имена классов – это   
переменные, которые ничем не отличаются от любых других переменных и не   
являются чем-то особенным в этом смысле):  
import person # Загружает класс с помощью инструкции import  
bob = person.Person(...) # Обращение к классу через имя модуля  
   
from person import Person # Загружает класс с помощью инструкции from  
bob = Person(...) # Обращение по непосредственному имени класса  
Для загрузки классов в сценарий мы будем использовать инструкцию from,   
просто потому, что в этом случае придется чуть меньше вводить с клавиатуры.   
Скопируйте следующий фрагмент, создающий экземпляры наших классов,   
в новый сценарий, чтобы нам было что сохранять (это всего лишь демонстраци-  
онный пример, поэтому мы не будем беспокоиться об избыточности программ-  
ного кода самопроверки). Создав экземпляры, мы практически без труда мо-  
жем сохранить их в хранилище. Для этого достаточно просто импортировать   
модуль shelve, открыть новое хранилище, указав имя внешнего файла, выпол-  
нить присваивание объектов по ключам и по окончании закрыть хранилище:  
# Файл makedb.py: сохраняет объекты Person в хранилище  
   
from person import Person, Manager # Импортирует наши классы  
bob = Person(‘Bob Smith’) # Создание объектов для сохранения  
sue = Person(‘Sue Jones’, job=’dev’, pay=100000)  
tom = Manager(‘Tom Jones’, 50000)  
   
import shelve  
db = shelve.open(‘persondb’) # Имя файла хранилища  
for object in (bob, sue, tom): # В качестве ключа использовать атрибут name  
 db[object.name] = object # Сохранить объект в хранилище  
db.close() # Закрыть после внесения изменений  
Обратите внимание, что при присваивании объектов в качестве ключей ис-  
пользуются значения атрибутов name. Так сделано просто потому, что это удоб-  
но, – ключами в хранилище могут быть любые строки, которые можно было   
бы создать с применением уникальных характеристик, таких как идентифи-  
катор процесса и отметки времени (их можно получить с помощью модулей   
os и time стандартной библиотеки). Единственное ограничение – ключи могут   
быть только строками и должны быть уникальными, потому что под каждым   
ключом можно сохранить только один объект (впрочем, таким объектом может   
быть список или словарь, содержащий множество объектов). А вот значения-  
ми, которые сохраняются по ключу, могут быть объекты практически любого   
типа: это могут быть объекты встроенных типов, таких как строки, списки,   
словари и экземпляры пользовательских классов, а также вложенные комби-  
нации из них.

760   
Глава 27. Более реалистичный пример   
Вот, собственно, и все – если при запуске сценарий ничего не выводит, это озна-  
чает, что он, скорее всего, работает – мы не предусмотрели вывод какой-либо   
информации, просто создаем и сохраняем объекты:  
C:\misc> makedb.py  
Исследование хранилища в интерактивном сеансе  
К настоящему моменту у нас имеется в текущем рабочем каталоге один или   
более файлов, имена которых начинаются с «persondb». Реально создаваемые   
файлы могут отличаться в зависимости от платформы, и точно так же, как при   
использовании встроенной функции open, функция shelve.open() создает фай-  
лы в текущем рабочем каталоге, если указанное имя файла не содержит пол-  
ный путь. Но независимо от того, где сохраняются эти файлы, они обеспечива-  
ют доступ по ключу к представлениям объектов, созданных с помощью модуля   
pickle. Не удаляйте эти файлы – они являются вашей базой данных, которую   
вам придется копировать или перемещать, когда вы будете создавать резерв-  
ные копии вашего хранилища или переносить его.   
При желании вы можете заглянуть внутрь файлов хранилищ с помощью про-  
граммы Проводника �indows (�indows Explorer) или с помощью интерактив-  
�indows (�indows Explorer) или с помощью интерактив-  
 (�indows Explorer) или с помощью интерактив-  
�indows Explorer) или с помощью интерактив-  
 Explorer) или с помощью интерактив-  
Explorer) или с помощью интерактив-  
) или с помощью интерактив-  
ной оболочки Python, однако эти файлы имеют двоичный формат и их содер-  
Python, однако эти файлы имеют двоичный формат и их содер-  
, однако эти файлы имеют двоичный формат и их содер-  
жимое не имеет большого смысла вне контекста модуля shelve. В Python 3.0,   
в случае отсутствия дополнительного программного обеспечения, наша база   
данных сохраняется в трех файлах (в 2.6 – только в одном, с именем persondb,   
потому что в этой версии присутствует модуль расширения bsddb; в версии 3.0   
модуль bsddb является сторонним дополнением, которое распространяется   
с открытыми исходными текстами):  
# Модуль, позволяющий получить список файлов в каталоге:   
# проверка наличия файлов  
   
>>> import glob  
>>> glob.glob(‘person\*’)  
[‘person.py’, ‘person.pyc’, ‘persondb.bak’, ‘persondb.dat’, ‘persondb.dir’]  
   
# Тип файла: текстовый – для строк, двоичный – для байтов  
   
>>> print(open(‘persondb.dir’).read())  
‘Tom Jones’, (1024, 91)  
...часть строк опущена...  
   
>>> print(open(‘persondb.dat’, ‘rb’).read())  
b’\x80\x03cperson\nPerson\nq\x00)\x81q\x01}q\x02(X\x03\x00\x00\x00payq\x03K...  
...часть строк опущена...  
Расшифровать такое содержимое вполне возможно, но оно может изменяться   
в зависимости от платформы и уж определенно не может квалифицировать-  
ся как дружественный интерфейс к базе данных! Чтобы проверить результаты   
нашего труда, можно написать еще один сценарий или попробовать поработать   
с хранилищем в интерактивной оболочке. Поскольку сами хранилища явля-  
ются объектами на языке Python, содержащими другие объекты на языке Py-  
Python, содержащими другие объекты на языке Py-  
, содержащими другие объекты на языке Py-  
Py-  
thon, мы можем работать с ними, используя привычный синтаксис языка Py-  
, мы можем работать с ними, используя привычный синтаксис языка Py-  
Py-  
thon. Ниже приводится листинг интерактивного сеанса, который фактически   
играет роль клиента базы данных:

Шаг 7 (последний): сохранение объектов в базе данных   
761  
>>> import shelve  
>>> db = shelve.open(‘persondb’) # Открыть хранилище  
   
>>> len(db) # В хранилище содержится три ‘записи’  
3  
>>> list(db.keys()) # keys – это оглавление  
[‘Tom Jones’, ‘Sue Jones’, ‘Bob Smith’] # Функция list используется, чтобы   
 # получить список в 3.0  
>>> bob = db[‘Bob Smith’] # Извлечь объект bob по ключу  
>>> print(bob) # Вызовет \_\_str\_\_ из AttrDisplay  
[Person: job=None, name=Bob Smith, pay=0]  
   
>>> bob.lastName() # Вызовет lastName из Person  
‘Smith’  
   
>>> for key in db: # Итерации, извлечение, вывод  
 print(key, ‘=>’, db[key])  
   
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]  
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]  
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]  
   
>>> for key in sorted(db):  
 print(key, ‘=>’, db[key]) # Итерации через отсортированный   
 # список ключей  
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]  
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]  
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]  
Обратите внимание: от нас не требуется импортировать классы Person или Man-  
ager, чтобы загрузить и использовать сохраненные объекты. Например, мы   
можем вызвать метод lastName объекта bob и вывести его содержимое, которое   
будет отформатировано автоматически, даже при том что класс Person этого   
объекта находится вне области видимости. Это обусловлено тем, что, когда мо-  
дуль pickle преобразует экземпляр класса, он записывает атрибуты экземпля-  
ра self вместе с именем класса, из которого он был создан, и именем модуля,   
где находится определение этого класса. Когда позднее объект bob извлекается   
из хранилища, интерпретатор автоматически импортирует класс и связывает   
с ним объект bob.  
Благодаря такому поведению после загрузки экземпляры классов автоматиче-  
ски обретают поведение своего класса. Мы должны импортировать наши клас-  
сы, только если необходимо создавать новые экземпляры, но не для работы   
с существующими. Такая особенность поведения влечет за собой следующие   
последствия:  
 •  
Недостаток заключается в том, что позднее, когда выполняется загрузка   
экземпляров, классы и их модули должны быть доступны для импорти-  
рования. Если говорить более формально, классы сохраняемых объектов   
должны быть определены на верхнем уровне модуля, который находится   
в одном из каталогов в пути поиска sys.path (и не должны находиться в мо-  
дуле \_\_main\_\_ сценария, если только они не используются только в пределах   
этого модуля). Из-за этих требований, предъявляемых к внешним файлам   
модулей, в некоторых приложениях для сохранения используются более   
простые объекты, такие как словари или списки, особенно когда они пере-  
даются другим приложениям через Интернет.

762   
Глава 27. Более реалистичный пример   
 •  
Преимущество заключается в том, что изменения в реализации класса ав-  
томатически будут восприняты экземплярами после их загрузки – часто   
нет никакой необходимости обновлять сохраненные объекты, потому что   
обычно изменения касаются только реализации методов класса.  
Кроме того, модуль shelve имеет определенные ограничения (некоторые из них   
упоминаются в конце этой главы, в списке баз данных, предлагаемых к ис-  
пользованию). Тем не менее модули shelve и pickle являются отличными ин-  
струментами, когда речь идет о реализации простого хранилища объектов.  
Обновление объектов в хранилище  
Теперь создадим еще один, последний сценарий, который обновляет экземпля-  
ры (записи) при каждом запуске, чтобы мы могли убедиться, что наши объ-  
екты действительно сохраняются между запусками программы (то есть при   
каждом запуске программы доступны их текущие значения). Следующий   
файл, updatedb.py, выводит содержимое базы данных и увеличивает зарплату   
в одном из наших объектов при каждом запуске. Если внимательно проследить   
за тем, что делает этот сценарий, можно заметить, что он «бесплатно» пользу-  
ется массой возможностей – при выводе наших объектов автоматически вы-  
зывается наша реализация метода \_\_str\_\_ и повышение зарплаты выполняет-  
ся вызовом нашего метода giveRaise. Все это возможно благодаря особенностям   
ООП и механизму наследования объектов, даже если сами объекты находятся   
в файле хранилища:  
# Файл updatedb.py: обновляет объект класса Person в базе данных  
   
import shelve  
db = shelve.open(‘persondb’) # Открыть хранилище в файле с указанным именем  
   
for key in sorted(db): # Обойти и отобразить объекты в базе данных  
 print(key, ‘\t=>’, db[key]) # Вывод в требуемом формате  
   
sue = db[‘Sue Jones’] # Извлечь объект по ключу  
sue.giveRaise(.10) # Изменить объект в памяти вызовом метода  
db[‘Sue Jones’] = sue # Присвоить по ключу,   
 # чтобы обновить объект в хранилище  
db.close() # Закрыть после внесения изменений  
Благодаря тому, что при запуске этот сценарий выводит содержимое базы дан-  
ных, мы можем запустить его несколько раз и увидеть, как изменяются наши   
объекты. Ниже приводятся результаты нескольких запусков сценария, где   
можно наблюдать, как каждый раз повышается зарплата sue (отличный сце-  
нарий для sue...):  
c:\misc> updatedb.py  
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]  
Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]  
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]  
   
c:\misc> updatedb.py  
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]  
Sue Jones => [Person: job=dev, name=Sue Jones, pay=110000]  
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]

Рекомендации на будущее   
763  
c:\misc> updatedb.py  
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]  
Sue Jones => [Person: job=dev, name=Sue Jones, pay=121000]  
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]  
   
c:\misc> updatedb.py  
Bob Smith => [Person: job=None, name=Bob Smith, pay=0]  
Sue Jones => [Person: job=dev, name=Sue Jones, pay=133100]  
Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]  
Все, что мы наблюдаем здесь, – это результат работы модулей shelve и pickle,   
входящих в состав Python, и поведения, которое мы сами реализовали в на-  
Python, и поведения, которое мы сами реализовали в на-  
, и поведения, которое мы сами реализовали в на-  
ших классах. Напомню также, что мы можем проверить результаты запуска   
сценария с помощью интерактивной оболочки (эквивалент клиента базы дан-  
ных на основе модуля shelve):  
c:\misc> python  
>>> import shelve  
>>> db = shelve.open(‘persondb’) # Открыть базу данных  
>>> rec = db[‘Sue Jones’] # Извлечь объект по ключу  
>>> print(rec)  
[Person: job=dev, name=Sue Jones, pay=146410]  
>>> rec.lastName()  
‘Jones’  
>>> rec.pay  
146410  
Еще один пример сохранения объектов вы найдете в главе 30, во врезке «При-  
дется держать в уме: классы и их хранение». В нем демонстрируется возмож-  
ность сохранения крупного составного объекта в простом текстовом файле, но   
не с помощью модуля shelve, а с помощью модуля pickle, хотя результат получа-  
ется тот же самый. Дополнительную информацию о модулях pickle и shelve вы   
найдете в других книгах и справочниках по языку Python.  
Рекомендации на будущее  
Это заключительный раздел данной главы. К настоящему моменту вы увиде-  
ли все основные механизмы ООП языка Python в действии и познакомились   
со способами, позволяющими избежать избыточности программного кода   
и сопутствующих ей проблем при сопровождении. Вы создали полноценные   
классы, выполняющие настоящую работу. Дополнительно вы смогли превра-  
, выполняющие настоящую работу. Дополнительно вы смогли превра-  
выполняющие настоящую работу. Дополнительно вы смогли превра-  
 настоящую работу. Дополнительно вы смогли превра-  
настоящую работу. Дополнительно вы смогли превра-  
 работу. Дополнительно вы смогли превра-  
работу. Дополнительно вы смогли превра-  
. Дополнительно вы смогли превра-  
Дополнительно вы смогли превра-  
тить экземпляры этих классов в настоящие записи в базе данных, сохранив их   
в хранилище с помощью модуля shelve, и тем самым обеспечили возможность   
долговременного хранения информации.  
Конечно, существует гораздо больше особенностей, чем мы исследовали здесь.   
Например, мы могли бы расширить наши классы, чтобы сделать их еще более   
реалистичными, добавить в них новые черты поведения и так далее. Напри-  
мер, на практике в операцию увеличения зарплаты следовало бы добавить про-  
верку коэффициента повышения, чтобы гарантировать, что его значение нахо-  
дится в диапазоне от нуля до единицы, – мы добавим ее ниже, в этой же книге,   
когда познакомимся с декораторами. Кроме того, этот пример можно было бы   
преобразовать в персональную базу данных с контактной информацией, изме-

764   
Глава 27. Более реалистичный пример   
нив перечень атрибутов в объектах и методов, предназначенных для их обра-  
ботки. В общем, этот список предложений можно продолжить в соответствии   
с богатством вашего воображения.   
Кроме того, мы могли расширить круг наших интересов и использовать ин-  
струменты, поставляемые в составе Python или доступные в мире свободного   
программного обеспечения:  
Графический интерфейс пользователя  
В настоящий момент мы можем взаимодействовать с нашей базой данных   
только с помощью командного интерфейса интерактивной оболочки и сце-  
нариев. Однако мы могли бы повысить удобство использования базы данных   
объектов, добавив графический интерфейс пользователя, позволяющий   
просматривать и изменять записи. Имеется возможность создавать перено-  
симые графические интерфейсы с помощью пакета tkinter (Tkinter – в 2.6),   
входящего в состав стандартной библиотеки Python, или с помощью сторон-  
Python, или с помощью сторон-  
, или с помощью сторон-  
них инструментов, таких как �xPython и PyQt. Пакет tkinter распростра-  
няется в составе Python, позволяет быстро создавать простые графические   
интерфейсы и идеально подходит для изучения приемов разработки гра-  
фического интерфейса. �xPython и PyQt являются более сложными в ис-  
�xPython и PyQt являются более сложными в ис-  
 и PyQt являются более сложными в ис-  
PyQt являются более сложными в ис-  
 являются более сложными в ис-  
пользовании, но зачастую позволяют получать более высококачественные   
графические интерфейсы.  
Веб-сайты  
Хотя графические интерфейсы удобны в использовании и обладают вы-  
сокой скоростью работы, тем не менее они не могут состязаться с веб-  
интерфейсами в смысле доступности. Вместо или в дополнение к графиче-  
скому интерфейсу и интерактивной оболочке мы могли бы также реализо-  
вать веб-сайт, позволяющий просматривать и изменять записи. Веб-сайты   
можно строить на основе простых инструментов создания CGI-сценариев,   
входящих в состав Python, или использовать полноценные веб-фреймворки   
от сторонних производителей, такие как Django, TurboGears, Pylons, web-  
Django, TurboGears, Pylons, web-  
, TurboGears, Pylons, web-  
TurboGears, Pylons, web-  
, Pylons, web-  
Pylons, web-  
, web-  
web-  
2Py, Zope или Google App Engine. При использовании веб-интерфейса дан-  
Py, Zope или Google App Engine. При использовании веб-интерфейса дан-  
, Zope или Google App Engine. При использовании веб-интерфейса дан-  
Zope или Google App Engine. При использовании веб-интерфейса дан-  
 или Google App Engine. При использовании веб-интерфейса дан-  
Google App Engine. При использовании веб-интерфейса дан-  
 App Engine. При использовании веб-интерфейса дан-  
App Engine. При использовании веб-интерфейса дан-  
 Engine. При использовании веб-интерфейса дан-  
Engine. При использовании веб-интерфейса дан-  
. При использовании веб-интерфейса дан-  
ные по-прежнему могут сохраняться в файлах с помощью модулей shelve,   
pickle или других инструментов, предназначенных для использования   
в программах на языке Python. Сценарии, обрабатывающие эти файлы,   
вызываются сервером автоматически, в ответ на запросы, поступающие от   
веб-броузеров и других клиентов, и возвращают разметку HTML, обеспе-  
HTML, обеспе-  
, обеспе-  
чивающую взаимодействие с пользователем, либо непосредственно, либо   
с применением фреймворка.  
Веб-службы  
Веб-клиенты часто сами могут выполнять анализ информации, получен-  
ной в ответе от веб-сайта (этот прием известен под красочным называнием   
«screen scraping»1). Однако мы могли бы пойти еще дальше и предоставить   
более прямой способ извлечения записей из базы данных на стороне веб-  
сервера – посредством интерфейсов веб-служб, таких как SOAP или XML-  
RPC, поддержка которых или включена в состав Python, или может быть   
1   
Screen scraping (дословно – «соскоб с экрана») – технология извлечения и передачи   
данных с экрана. В данном случае подразумевается извлечение информации из раз-  
метки HTML. – Прим. перев.

В заключение   
765  
добавлена за счет установки сторонних, свободно распространяемых паке-  
тов и модулей. Веб-службы возвращают данные в более непосредственном   
виде, по сравнению со страницами HTML, возвращаемыми веб-сервером.   
Базы данных  
Если база данных должна хранить большой объем данных или эти данные   
имеют большое значение, мы могли бы отказаться от использования моду-  
ля shelve и использовать более полноценный механизм хранения данных,   
такой как ZODB (свободно распространяемая, объектно-ориентированная   
база данных, ООДБ), или более традиционную реляционную базу данных,   
такую как MySQL, Oracle, PostgreSQL или SQLite. В состав Python уже вхо-  
MySQL, Oracle, PostgreSQL или SQLite. В состав Python уже вхо-  
, Oracle, PostgreSQL или SQLite. В состав Python уже вхо-  
Oracle, PostgreSQL или SQLite. В состав Python уже вхо-  
, PostgreSQL или SQLite. В состав Python уже вхо-  
PostgreSQL или SQLite. В состав Python уже вхо-  
 или SQLite. В состав Python уже вхо-  
SQLite. В состав Python уже вхо-  
. В состав Python уже вхо-  
Python уже вхо-  
 уже вхо-  
дит поддержка встраиваемой системы баз данных SQLite, однако в Сети   
вы найдете и другие свободно распространяемые альтернативы. Механизм   
ZODB, например, своими особенностями напоминает модуль shelve, однако   
в ZODB отсутствуют многие ограничения, присущие shelve� он поддержи-  
вает возможность работы с большими базами данных, параллельное изме-  
нение данных, транзакции и автоматическую сквозную запись изменений,   
выполняемых в памяти. Базы данных SQL, такие как MySQL, обеспечива-  
SQL, такие как MySQL, обеспечива-  
, такие как MySQL, обеспечива-  
MySQL, обеспечива-  
, обеспечива-  
ют инструментальные средства по организации хранилищ данных уровня   
предприятия и могут напрямую использоваться из сценариев на языке Py-  
Py-  
thon.  
Механизмы объектно-реляционных отображений (ORM)  
При переходе на использование системы управления реляционными ба-  
зами данных нам не придется отказываться от инструментов ООП, имею-  
щихся в языке Python. Механизмы объектно-реляционного отображения   
(object-relational mapping, ORM), такие как SQLObject и SQLAlchemy, могут   
автоматически отображать реляционные таблицы и записи в классы и эк-  
земпляры на языке Python и обратно, благодаря чему мы можем обраба-  
Python и обратно, благодаря чему мы можем обраба-  
 и обратно, благодаря чему мы можем обраба-  
тывать хранимые данные, используя привычный синтаксис классов языка   
Python. Эти механизмы обеспечивают альтернативу использованию ООДБ,   
таких как shelve и ZODB, и позволяют объединить сильные стороны реля-  
ZODB, и позволяют объединить сильные стороны реля-  
, и позволяют объединить сильные стороны реля-  
ционных баз данных и модели классов в языке Python.  
Я надеюсь, что это введение подтолкнет вас к дальнейшим исследованиям, но,   
к сожалению, обсуждение всех этих тем выходит далеко за рамки данной гла-  
вы и книги в целом. Если у вас появится желание заняться самостоятельными   
исследованиями, дополнительную информацию вы сможете найти в Сети, в ру-  
ководствах по стандартной библиотеке Python и в книгах, посвященных при-  
Python и в книгах, посвященных при-  
 и в книгах, посвященных при-  
кладным аспектам программирования, – в таких как «Программирование на   
Python». В ней я продолжаю этот пример с того места, где я остановился в этой   
книге, и показываю, как можно реализовать графический и веб-интерфейс   
к базе данных, обеспечивающий возможность просмотра и изменения записей.   
Я надеюсь встретиться с вами в той книге, а пока вернемся к основам классов   
и продолжим изучение основ языка программирования Python.  
В заключение  
В этой главе мы на практических примерах исследовали все основные особен-  
ности классов и ООП в языке Python. В несколько этапов мы создали простой   
действующий пример. Добавили конструкторы, методы, реализовали пере-  
 пример. Добавили конструкторы, методы, реализовали пере-  
пример. Добавили конструкторы, методы, реализовали пере-  
. Добавили конструкторы, методы, реализовали пере-  
Добавили конструкторы, методы, реализовали пере-

766   
Глава 27. Более реалистичный пример   
грузку операторов, адаптировали базовые классы с помощью подклассов и ис-  
пользовали инструменты интроспекции. Мы также познакомились с другими   
концепциями (такими как составные объекты, делегирование и полимор-  
физм).  
В заключение мы реализовали сохранение объектов, созданных из наших клас-  
сов, в объектно-ориентированной базе данных с применением модуля shelve –   
простой в использовании системы, обеспечивающей возможность сохранения   
и извлечения объектов по ключу. Исследуя основы использования классов, мы   
также столкнулись с различными способами структуризации программного   
кода, позволяющими уменьшить его избыточность и минимизировать усилия   
по его сопровождению в будущем. Наконец, мы кратко обрисовали направле-  
ния расширения нашего примера с помощью инструментов прикладного про-  
граммирования, такие как создание графического интерфейса и применение   
баз данных, о которых подробнее рассказывается в следующих моих книгах.  
В следующих главах этой части книги мы вернемся к изучению тонкостей   
устройства модели классов в языке Python и узнаем, как они применяются   
в некоторых концепциях проектирования, используемых для объединения   
классов в крупных программах. Однако прежде чем двинуться дальше, ответь-  
те на контрольные вопросы, чтобы освежить в памяти основы, которые были   
рассмотрены здесь. Поскольку в этой главе мы получили неплохой практиче-  
ский опыт, мы закроем ее набором в основном теоретических вопросов, кото-  
рые составлены так, чтобы заставить вас вернуться к программному коду и по-  
размыслить над некоторыми основными идеями, стоящими за ним.  
Закрепление пройденного  
Контрольные вопросы  
1. Когда мы извлекаем объект класса Manager из хранилища и выводим его,   
как интерпретатор определяет, какую логику форматирования и отображе-  
ния следует применить?  
2. Когда объект класса Person извлекается из хранилища без импортирования   
модуля, в котором он определен, как интерпретатор узнает, что объект об-  
ладает методом giveRaise, который мы можем вызвать?  
3. Почему так важно перенести обработку атрибутов в методы, а не выполнять   
ее за пределами класса?  
4. Почему лучше использовать прием адаптации с помощью подкласса, чем   
копировать и изменять оригинальный программный код?  
5. Почему лучше вызывать метод суперкласса для выполнения действий по   
умолчанию, чем копировать и изменять его программный код в подклассе?  
6. Почему лучше использовать такие инструменты, как \_\_dict\_\_, позволяю-  
щие реализовать обобщенную обработку объектов, чем писать специализи-  
рованный программный код для каждого отдельного класса?  
7. Когда вообще может быть желательным вместо наследования использовать   
прием встраивания одних объектов в другие?  
8. Как бы вы изменили классы, представленные в этой главе, чтобы на их   
основе реализовать персональную базу данных с контактной информацией?

Закрепление пройденного   
767  
Ответы  
1. В окончательной версии нашего примера класс Manager наследует метод \_\_  
str\_\_ вывода из класса AttrDisplay, находящегося в модуле classtools. Класс   
Manager не имеет собственного метода \_\_str\_\_, поэтому механизм поиска в де-  
реве наследования просматривает его суперкласс Person. Поскольку в этом   
классе также нет метода \_\_str\_\_, поиск переносится в класс AttrDisplay,   
стоящий еще выше, где и обнаруживается искомый метод. Имена классов,   
перечисленные в круглых скобках в заголовке инструкции class, обеспечи-  
вают связь с вышестоящими суперклассами.  
2. Модуль shelve (в действительности – модуль pickle, который используется   
модулем shelve) автоматически связывает экземпляр с его классом, когда   
загружает его из хранилища в память. Интерпретатор автоматически им-  
портирует класс из его модуля, создает экземпляр со всеми его сохранен-  
ными значениями атрибутов и записывает в атрибут \_\_class\_\_ экземпляра   
ссылку на оригинальный класс. Благодаря этому загруженные экземпля-  
ры автоматически обретают оригинальные методы (такие как lastName,   
giveRaise и \_\_str\_\_), даже если мы не выполняли импортирование класса   
в текущую область видимости.  
3. Переносить обработку атрибутов в методы важно потому, что в этом случае   
остается единственная копия программного кода, которую может потребо-  
ваться изменить в будущем, а также потому, что методы могут вызываться   
относительно любого экземпляра. Этот прием называется инкапсуляцией –   
упаковывание логики в интерфейсы с целью упростить поддержку про-  
граммного кода в будущем. Если этого не сделать, программный код будет   
получаться избыточным, что повлечет за собой многократное увеличение   
усилий, которые потребуется затратить на его сопровождение и дальней-  
шее развитие в будущем.   
4. Адаптация с помощью подклассов уменьшает усилия, затрачиваемые на   
разработку. При использовании объектно-ориентированного стиля про-  
граммирования мы адаптируем уже имеющийся программный код, а не   
копируем и изменяем его. Это по-настоящему «самый основной» принцип   
ООП – мы легко можем расширять возможности программного кода, на-  
писанного ранее, создавая новые подклассы, и можем использовать то, что   
уже имеется. Это намного лучше, чем каждый раз начинать все с самого   
начала или создавать множество избыточных копий программного кода,   
которые, вполне возможно, придется изменять в будущем.  
5. Копирование и изменение существующего программного кода удваивает   
объем работы, которую придется проделывать в будущем, независимо от   
ситуации. Если в подклассе требуется выполнить какие-либо действия,   
предусмотренные по умолчанию и реализованные в методе суперкласса,   
будет намного лучше произвести вызов оригинального метода по имени су-  
перкласса, чем копировать его программный код. То же справедливо и по   
отношению к конструкторам суперклассов. Повторюсь, что копирование   
программного кода создает избыточность, что может стать источником про-  
блем при дальнейшем его развитии.  
6. Обобщенные инструменты помогут избежать жестко запрограммирован-  
ных решений, которые постоянно придется изменять по мере дальнейшего   
развития класса. Обобщенный метод \_\_str\_\_ вывода, например, не придется   
изменять каждый раз при добавлении нового атрибута экземпляра в кон-

768   
Глава 27. Более реалистичный пример   
структоре \_\_init\_\_. Кроме того, обобщенный метод вывода будет унаследо-  
ван всеми классами, а в случае необходимости изменения придется вносить   
только в одном месте – изменения в обобщенной версии автоматически бу-  
дут отражаться на всех других классах, наследующих обобщенный класс.   
Повторюсь еще раз, устранение избыточности в программном коде умень-  
шает усилия, которые потребуется затратить на его развитие в будущем, –   
это одно из основных преимуществ, которые несут в себе классы.  
7. Наследование лучше использовать при разработке расширений с опорой на   
непосредственную адаптацию (подобно тому, как класс Manager адаптирует   
класс Person). Встраивание хорошо подходит в случаях, когда необходимо   
объединить несколько объектов в единое целое и управлять ими с помощью   
промежуточного класса. Механизм наследования позволяет многократно   
использовать методы суперклассов, а метод встраивания – делегировать   
вызовы методов другим классам. Наследование и встраивание не являются   
взаимоисключающими методиками – нередко встречаются ситуации, ког-  
да объекты встраиваются в контроллер и при этом сами могут адаптиро-  
ваться с помощью механизма наследования.  
8. Классы из этой главы можно было бы использовать как «заготовки» для реа-  
лизации баз данных различных типов. Фактически вы можете изменить их   
назначение, модифицировав конструкторы, определив в них другие атрибу-  
ты и создав методы, подходящие для целевого применения. Например, для   
создания базы данных с контактной информацией можно было бы создать   
такие атрибуты, как name, address, birthday, phone, email и так далее, и мето-  
ды, соответствующие этой цели. Метод с именем sendmail, например, мог   
бы с помощью модуля smptlib из стандартной библиотеки автоматически   
отправлять электронные сообщения (за более подробной информацией о по-  
добных инструментах обращайтесь к руководствам по языку Python или   
к книгам прикладного характера). Для вывода контактной информации,   
хранящейся в объектах, можно было бы использовать класс AttrDisplay, ко-  
торый мы написали в этой главе, потому что именно для этого он и задумы-  
вался. Для организации долговременного хранения объектов также можно   
было бы использовать большую часть программного кода, представленного   
здесь и использующего модуль shelve, с незначительными изменениями.

Глава 28.  
   
Подробнее   
о программировании классов  
Если что-то относительно ООП в языке Python до сих пор осталось для вас не-  
Python до сих пор осталось для вас не-  
 до сих пор осталось для вас не-  
понятным, не волнуйтесь – теперь, совершив краткий тур, мы начнем копать   
немного глубже и подробно изучим понятия, представленные ранее. В этой и в   
следующей главе мы с другой стороны посмотрим на классы. В данной главе   
мы рассмотрим классы, методы и наследование. Формализуем и дополним   
некоторые идеи программирования классов, представленные в главе 26. По-  
скольку классы являются нашим последним инструментом пространств имен,   
то здесь мы также сведем вместе концепции пространств имен в языке Python.   
В следующей главе мы продолжим эту тему и предпримем вторую попытку   
изучения механики классов, рассмотрев один важный аспект: перегрузку опе-  
раторов. Кроме того, в этой и в следующей главе будут представлены примеры   
более крупных классов, чем те, что мы видели до сих пор.   
Инструкция class  
Несмотря на то что на первый взгляд инструкция class в языке Python напо-  
Python напо-  
 напо-  
минает похожие инструменты в других языках программирования, при более   
близком рассмотрении видно, что она существенно отличается от того, к чему   
привыкли некоторые программисты. Например, как и инструкция class в язы-  
ке C++, инструкция class в языке Python является основным инструментом   
ООП, но в отличие от инструкции в C++, в языке Python она не является объ-  
C++, в языке Python она не является объ-  
++, в языке Python она не является объ-  
Python она не является объ-  
 она не является объ-  
явлением. Подобно инструкции def, инструкция class создает объект и явля-  
ется неявной инструкцией присваивания – когда она выполняется, создается   
объект класса, ссылка на который сохраняется в имени, использованном в за-  
головке инструкции. Кроме того, как и инструкция def, инструкция class яв-  
ляется настоящим выполняемым программным кодом – класс не существует,   
пока поток выполнения не достигнет инструкции class, которая определяет его   
(обычно при импортировании модуля, в котором она находится, но не ранее).

770   
Глава 28. Подробнее о программировании классов   
Общая форма  
Инструкция class – это составная инструкция, с блоком операторов, обычно   
под строкой заголовка. В заголовке после имени в круглых скобках через запя-  
тую перечисляются суперклассы. Наличие более одного суперкласса в списке   
означает множественное наследование (которое более формально будет обсуж-  
даться в главе 30). Ниже показана общая форма инструкции:  
class <name>(superclass,...): # Присваивание имени  
 data = value # Совместно используемые данные класса  
 def method(self,...): # Методы  
 self.member = value # Данные экземпляров  
Внутри инструкции class любая операция присваивания создает атрибут клас-  
са, а методы со специальными именами перегружают операторы. Например,   
функция с именем \_\_init\_\_, если она определена, вызывается во время созда-  
ния объекта экземпляра.  
Пример  
Как мы уже видели, классы – это всего лишь пространства имен, то есть ин-  
струменты, определяющие имена (атрибуты), с помощью которых клиентам   
экспортируются данные и логика. Так как же инструкция class порождает   
пространство имен?  
А вот как. Так же как и в модулях, инструкции, вложенные в тело инструкции   
class, создают атрибуты класса. Когда интерпретатор достигает инструкции   
class (а не тогда, когда происходит вызов класса), он выполняет все инструкции   
в ее теле от начала и до конца. Все присваивания, которые производятся в ходе   
этого процесса, создают имена в локальной области видимости класса, которые   
становятся атрибутами объекта класса. Благодаря этому классы напоминают   
модули и функции:  
 •  
Подобно функциям, инструкции class являются локальными областями   
видимости, где располагаются имена, созданные вложенными операциями   
присваивания.  
 •  
Подобно именам в модуле, имена, созданные внутри инструкции class, ста-  
новятся атрибутами объекта класса.  
Основное отличие классов состоит в том, что их пространства имен также со-  
ставляют основу механизма наследования в языке Python, – ссылки на атри-  
Python, – ссылки на атри-  
, – ссылки на атри-  
буты, отсутствующие в классе или в объекте экземпляра, будут получены из   
других классов.  
Поскольку инструкция class – это составная инструкция, в ее тело могут быть   
вложены любые инструкции – print, =, if, def и так далее. Все инструкции   
внутри инструкции class выполняются, когда выполняется сама инструкция   
class (а не когда позднее класс вызывается для создания экземпляра). Опера-  
ции присваивания именам внутри инструкции class создают атрибуты класса,   
а вложенные инструкции def создают методы класса� кроме этого, атрибуты   
класса создаются и другими инструкциями, выполняющими присваивание.  
Например, присваивание объекта, не являющегося функцией, атрибутам соз-  
дает атрибуты данных, совместно используемых всеми экземплярами:  
>>> class SharedData:  
... spam = 42 # Создает атрибут данных класса

Инструкция class   
771  
...  
>>> x = SharedData() # Создать два экземпляра  
>>> y = SharedData()  
>>> x.spam, y.spam # Они наследуют и совместно используют атрибут spam  
(42, 42)  
В данном случае из-за того, что имя spam создается на верхнем уровне в ин-  
струкции class, оно присоединяется к классу и поэтому совместно использу-  
ется всеми экземплярами. Мы можем изменять значение атрибута, выполняя   
присваивание через имя класса, и обращаться к нему через имена экземпляров   
или класса.1  
>>> SharedData.spam = 99  
>>> x.spam, y.spam, SharedData.spam  
(99, 99, 99)  
Такие атрибуты класса могут использоваться для хранения информации, до-  
ступной всем экземплярам, например для хранения счетчика количества соз-  
данных экземпляров (эту идею мы рассмотрим в главе 31). Теперь посмотрим,   
что произойдет, если присвоить значение атрибуту spam не через имя класса,   
а через имя экземпляра:  
>>> x.spam = 88  
>>> x.spam, y.spam, SharedData.spam  
(88, 99, 99)  
Операция присваивания, применяемая к атрибуту экземпляра, создает или   
изменяет имя в экземпляре, а не в классе. Вообще говоря, поиск в дереве на-  
следования производится только при попытке чтения атрибута, но не при при-  
сваивании: операция присваивания атрибуту объекта всегда изменяет сам объ-  
ект, а не что-то другое.2 Например, атрибут y.spam будет найден в наследуемом   
классе, а операция присваивания атрибуту x.spam присоединит имя непосред-  
ственно к объекту x.  
Ниже приводится более понятный пример этого поведения, где одно и то же имя   
создается в двух местах. Предположим, что мы используем следующий класс:  
class MixedNames: # Определение класса  
 data = ‘spam’ # Присваивание атрибуту класса  
 def \_\_init\_\_(self, value): # Присваивание имени метода  
 self.data = value # Присваивание атрибуту экземпляра  
 def display(self):  
 print(self.data, MixedNames.data) # Атрибут экземпляра, атрибут класса  
1   
Если у вас есть опыт работы с языком C++, вы можете заметить в этом некоторое   
сходство со «статическими» членами данных в языке С++ – членами, которые хра-  
нятся в классе независимо от экземпляров. В языке Python в этом нет ничего осо-  
бенного: все атрибуты класса – это всего лишь имена, созданные в инструкции class,   
независимо от того, ссылаются они на методы («методы» в C++) или на что-то дру-  
гое («члены» в C++). В главе 31 мы также познакомимся со статическими методами   
(родственными статическим методам в языке C++), которые являются обычными   
функциями, которым не передается аргумент self, и которые обычно используются   
для работы с атрибутами класса.  
2   
При условии, что класс не переопределил операцию присваивания с помощью метода   
перегрузки оператора \_\_setattr\_\_ (обсуждается в главе 29) с целью выполнять какие-  
то особые действия.

772   
Глава 28. Подробнее о программировании классов   
Этот класс содержит две инструкции def, которые связывают атрибуты класса   
с методами. Здесь также присутствует инструкция присваивания =. Так как эта   
инструкция выполняет присваивание имени data внутри инструкции class, оно   
создается в локальной области видимости класса и становится атрибутом объ-  
екта класса. Как и все атрибуты класса, атрибут data наследуется и использует-  
ся всеми экземплярами класса, которые не имеют собственного атрибута data.  
Когда создаются экземпляры этого класса, имя data присоединяется к этим эк-  
земплярам через присваивание атрибуту self.data в конструкторе:  
>>> x = MixedNames(1) # Создаются два объекта экземпляров,  
>>> y = MixedNames(2) # каждый из которых имеет свой атрибут data  
>>> x.display(); y.display() # self.data - это другие атрибуты,   
1 spam # а MixedNames.data - тот же самый  
2 spam  
Суть этого примера состоит в том, что атрибут data находится в двух разных   
местах: в объектах экземпляров (создаются присваиванием атрибуту self.data   
в методе \_\_init\_\_) и в классе, от которого они наследуют имена (создается при-  
сваиванием имени data в инструкции class). Метод класса display выводит обе   
версии – сначала атрибут экземпляра self, а затем атрибут класса.  
Используя этот прием сохранения атрибутов в различных объектах, мы опре-  
деляем их области видимости. Атрибуты классов совместно используются   
всеми экземплярами, а атрибуты экземпляров уникальны для каждого экзем-  
пляра – ни данные, ни поведение экземпляра недоступны для совместного ис-  
пользования. Несмотря на то что операция поиска в дереве наследования по-  
зволяет отыскивать имена, мы всегда можем получить доступ к ним в любой   
точке дерева, обратившись непосредственно к нужному объекту.  
В предыдущем примере, например, выражения x.data и self.data возвращают   
атрибут экземпляра, которые переопределяют то же самое имя в классе. Одна-  
ко выражение MixedNames.data явно обращается к атрибуту класса. Позднее мы   
еще встретим подобные шаблоны программирования, например следующий   
раздел описывает один из наиболее часто используемых.  
Методы  
Вы уже знакомы с функциями и знаете о методах в классах. Методы – это обыч-  
ные объекты функций, которые создаются инструкциями def в теле инструк-  
ции class. Говоря вкратце, методы реализуют поведение, наследуемое объек-  
тами экземпляров. С точки зрения программирования методы работают точно   
так же, как и обычные функции, с одним важным исключением: в первом ар-  
гументе методам всегда передается подразумеваемый объект экземпляра.  
Другими словами, интерпретатор автоматически отображает вызов метода эк-  
земпляра на метод класса следующим образом. Вызов метода экземпляра:  
instance.method(args...)  
автоматически преобразуется в вызов метода класса:  
class.method(instance, args...)  
где класс определяется в результате поиска имени метода по дереву наследо-  
вания. Фактически в языке Python обе формы вызова метода являются допу-  
Python обе формы вызова метода являются допу-  
 обе формы вызова метода являются допу-  
стимыми.

Методы   
773  
Помимо обычного наследования имен методов, первый специальный аргу-  
мент – это единственная необычная особенность методов. Первый аргумент   
в методах классов обычно называется self, в соответствии с общепринятыми   
соглашениями (с технической точки зрения само имя не играет никакой роли,   
значение имеет позиция аргумента). Этот аргумент обеспечивает доступ к эк-  
земпляру, то есть к субъекту вызова, – поскольку из классов может создавать-  
ся множество объектов экземпляров, этот аргумент необходим для доступа   
к данным конкретного экземпляра.  
Программисты, знакомые с языком C++, сочтут, что аргумент self в языке   
Python напоминает указатель this в языке C++. Однако в языке Python имя   
self всегда явно используется в программном коде: методы всегда должны ис-  
пользовать имя self для получения или изменения атрибутов экземпляра, об-  
рабатываемого текущим вызовом метода. Такая явная природа аргумента self   
предусмотрена намеренно – присутствие этого имени делает очевидным ис-  
пользование имен атрибутов экземпляра.  
Пример метода  
Чтобы пояснить эти концепции, обратимся к примеру. Предположим, что име-  
ется следующее определение класса:  
class NextClass: # Определение класса  
 def printer(self, text): # Определение метода  
 self.message = text # Изменение экземпляра  
 print(self.message) # Обращение к экземпляру  
Имя printer ссылается на объект функции, а так как оно создается в области   
видимости инструкции class, оно становится атрибутом объекта класса и будет   
унаследовано всеми экземплярами, которые будут созданы из класса. Обычно   
методы, такие как printer, предназначены для обработки экземпляров, поэто-  
му мы вызываем их через экземпляры:  
>>> x = NextClass() # Создать экземпляр  
   
>>> x.printer(‘instance call’) # Вызвать его метод  
instance call  
   
>>> x.message # Экземпляр изменился  
‘instance call’  
Когда метод вызывается с использованием квалифицированного имени экзем-  
пляра, как в данном случае, то сначала определяется местонахождение метода   
printer, а затем его аргументу self автоматически присваивается объект экзем-  
пляра (x). В аргумент text записывается строка, переданная в вызов метода   
(‘instance call’). Обратите внимание, что Python автоматически передает в пер-  
Python автоматически передает в пер-  
 автоматически передает в пер-  
вом аргументе self ссылку на сам экземпляр, поэтому нам достаточно передать   
методу только один аргумент. Внутри метода printer имя self используется для   
доступа к данным конкретного экземпляра, потому что оно ссылается на теку-  
щий обрабатываемый экземпляр.  
Методы могут вызываться любым из двух способов – через экземпляр или через   
сам класс. Например, метод printer может быть вызван с использованием имени   
класса, при этом ему явно требуется передать экземпляр в аргументе self:   
>>> NextClass.printer(x, ‘class call’) # Прямой вызов метода класса  
class call

774   
Глава 28. Подробнее о программировании классов   
>>> x.message # Экземпляр снова изменился  
‘class call’  
Вызов метода, который производится через экземпляр и через имя класса, ока-  
зывает одинаковое воздействие при условии, что при вызове через имя класса   
передается тот же самый экземпляр. По умолчанию, если попытаться вызвать   
метод без указания экземпляра, будет выведено сообщение об ошибке:  
>>> NextClass.printer(‘bad call’)  
TypeError: unbound method printer() must be called with NextClass instance...  
(TypeError: несвязанный метод printer() должен вызываться с экземпляром NextClass...)  
Вызов конструкторов суперклассов  
Обычно методы вызываются через экземпляры. Тем не менее вызовы методов   
через имя класса могут играть особую роль. Одна из таких ролей связана с вы-  
зовом конструктора. Метод \_\_init\_\_ наследуется точно так же, как и любые   
другие атрибуты. Это означает, что во время создания экземпляра интерпрета-  
тор отыскивает только один метод \_\_init\_\_. Если в конструкторе подкласса не-  
обходимо гарантировать выполнение действий, предусматриваемых конструк-  
тором суперкласса, необходимо явно вызвать метод \_\_init\_\_ через имя класса:  
class Super:  
 def \_\_init\_\_(self, x):  
 ...программный код по умолчанию...  
   
class Sub(Super):  
 def \_\_init\_\_(self, x, y):  
 Super.\_\_init\_\_(self, x) # Вызов метода \_\_init\_\_ суперкласса  
 ...адаптированный код... # Выполнить дополнительные действия  
   
I = Sub(1, 2)  
Это один из немногих случаев, когда вашему программному коду потребуется   
явно вызывать метод перегрузки оператора. Естественно, вызывать конструк-  
тор суперкласса таким способом следует, только если это действительно необ-  
ходимо, – без этого вызова подкласс полностью переопределяет данный метод.   
Более реалистичный случай применения этого приема приводится в классе   
Manager, в предыдущей главе.1  
Другие возможности методов  
Такой способ вызова методов через имя класса представляет собой основу для   
расширения (без полной замены) поведения унаследованных методов. В гла-  
ве 31 мы познакомимся с еще одной возможностью, добавленной в Python 2.2,   
статическими  методами, которые не предполагают наличие объекта эк-  
земпляра в первом аргументе. Такие методы могут действовать как обычные   
функции, имена которых являются локальными по отношению к классам, где   
они были определены, и использоваться для манипулирования данными клас-  
са. Родственные им методы класса принимают в первом аргументе сам класс,   
1   
Небольшое замечание: внутри одного и того же класса можно определить несколько   
методов с именем \_\_init\_\_, но использоваться будет только последнее определение.   
Дополнительные подробности приводятся в главе 30.

Наследование   
775  
а не экземпляр, и могут использоваться для манипулирования данными, при-  
надлежащими конкретному классу. Однако это дополнительное расширение   
не является обязательным – обычно нам всегда бывает необходимо передавать   
экземпляр методам, вызываемым либо через сам экземпляр, либо через имя   
класса.  
Наследование  
Основное назначение такого инструмента пространств имен, как инструкция   
class, заключается в обеспечении поддержки наследования имен. В этом раз-  
деле мы подробно остановимся на вопросах, связанных с механизмами насле-  
дования атрибутов в языке Python.  
В языке Python наследование вступает в игру после того, как объект будет ква-  
Python наследование вступает в игру после того, как объект будет ква-  
 наследование вступает в игру после того, как объект будет ква-  
лифицирован, и его действие заключается в операции поиска в дереве опреде-  
лений атрибутов (в одном или более пространствах имен). Каждый раз, когда   
используется выражение вида object.attr (где object – это объект экземпляра   
или класса), интерпретатор приступает к поиску первого вхождения атрибу-  
та attr в дереве пространств имен снизу вверх, начиная с объекта object. Сюда   
относятся и ссылки на атрибуты аргумента self внутри методов. Поскольку   
самые нижние определения в дереве наследования переопределяют те, что на-  
ходятся выше, механизм наследования составляет основу специализации про-  
граммного кода.  
Создание дерева атрибутов  
На рис. 28.1 приводятся способы, которыми создаются и заполняются имена-  
ми деревья пространств имен. Вообще:  
 •  
Атрибуты экземпляров создаются посредством присваивания атрибутам   
аргумента self в методах.  
 •  
Атрибуты классов создаются инструкциями (присваивания), расположен-  
ными внутри инструкции class.  
 •  
Ссылки на суперклассы создаются путем перечисления классов в круглых   
скобках в заголовке инструкции class.  
Результатом является дерево пространств имен с атрибутами, которое ведет   
в направлении от экземпляров к классам, из которых они были созданы, и ко   
всем суперклассам, перечисленным в заголовке инструкции class. Интерпре-  
татор выполняет поиск в дереве в направлении снизу вверх, от экземпляров   
к суперклассам всякий раз, когда используемое имя подразумевает атрибут   
объекта экземпляра.1  
1   
Это описание далеко не полное, потому что точно так же возможно создавать атри-  
буты экземпляров и классов с помощью инструкций присваивания за пределами ин-  
струкций class – но этот прием используется существенно реже и зачастую более под-  
вержен ошибкам (изменения не изолированы от инструкций class). В языке Python   
все атрибуты всегда доступны по умолчанию. Более подробно о сокрытии данных мы   
поговорим в главе 29, где мы поближе познакомимся с методом \_\_setattr\_\_, в гла-  
ве 30, где мы будем рассматривать имена вида \_\_X, и еще раз – в главе 38, где мы   
займемся реализацией декоратора класса.

776   
Глава 28. Подробнее о программировании классов   
Экземпляр  
Объекты  
Программа  
Суперкласс  
Суперкласс  
Класс  
class X(S1, S2):  
 def attr(self,...):  
 self.attr = V  
class S1:  
class S2:  
object = X()  
object.attr?  
Рис. 28.1. Программный код создает дерево наследования объектов в памяти,   
в котором будет выполняться поиск атрибутов. Вызов класса создает новый   
экземпляр, который помнит, к какому классу он принадлежит. Инструкция   
class, в заголовке которой в круглых скобках перечислены суперклассы, созда-  
, в заголовке которой в круглых скобках перечислены суперклассы, созда-  
ет новый класс. Каждое обращение к атрибуту вызывает новую процедуру   
поиска в дереве наследования снизу вверх – даже обращения к атрибутам ар-  
гумента self в методах класса  
Специализация унаследованных методов  
Только что описанная модель поиска в дереве наследования представляет собой   
прекрасный способ специализации программ. Поскольку механизм наследова-  
ния сначала пытается отыскать имена в подклассах и только потом в их супер-  
классах, подклассы могут изменять поведение по умолчанию, предусматривае-  
мое атрибутами их суперклассов. Фактически можно создавать целые системы   
как иерархии классов, возможности которых расширяются за счет добавления   
новых подклассов, а не за счет изменения существующего программного кода.  
Идея переопределения унаследованных имен приводит к множеству приемов   
специализации. Например, подклассы могут полностью замещать унасле-  
дованные атрибуты, предоставлять атрибуты, которые ожидается отыскать   
в суперклассах, и расширять методы суперкласса за счет их вызова из методов   
подкласса. Мы уже видели прием с замещением в действии. Ниже приводится   
пример, демонстрирующий, как выполняется расширение:  
>>> class Super:  
... def method(self):  
... print(‘in Super.method’)  
...  
>>> class Sub(Super):  
... def method(self): # Переопределить метод  
... print(‘starting Sub.method’)# Дополнительное действие  
... Super.method(self) # Выполнить действие по умолчанию  
... print(‘ending Sub.method’)  
...

Наследование   
777  
Главное здесь – это прямые вызовы методов суперкласса. Класс Sub замещает   
метод method класса Super своей собственной, специализированной версией. Но   
внутри замещающего метода в классе Sub производится вызов версии, экспор-  
тируемой классом Super, чтобы выполнить действия по умолчанию. Другими   
словами, метод Sub.method не замещает полностью метод Super.method, а просто   
расширяет его:   
>>> x = Super() # Создать экземпляр класса Super   
>>> x.method() # Вызвать Super.method  
in Super.method  
   
>>> x = Sub() # Создать экземпляр класса Sub  
>>> x.method() # Вызвать Sub.method, который вызовет Super.method  
starting Sub.method  
in Super.method  
ending Sub.method  
Этот прием расширения также часто используется в конструкторах, за приме-  
рами обращайтесь к предыдущему разделу «Методы».  
Приемы организации взаимодействия классов  
Расширение – это лишь один из способов организации взаимодействий с супер-  
классом. В файле ниже, specialize.py, определяется несколько классов, кото-  
рые иллюстрируют различные приемы использования классов:  
Super  
Определяет метод method и метод delegate, который предполагает наличие   
метода action в подклассе.  
Inheritor  
Не предоставляет никаких новых имен, поэтому он получает все, что опре-  
делено только в классе Super.  
Replacer  
Переопределяет метод method класса Super своей собственной версией.  
Extender  
Адаптирует метод method класса Super, переопределяя и вызывая его, чтобы   
выполнить действия, предусмотренные по умолчанию.  
Provider  
Реализует метод action, который ожидается методом delegate класса Super.  
Рассмотрим каждый из этих классов, чтобы получить представление о спосо-  
бах, которыми они адаптируют свой общий суперкласс. Содержимое самого   
файла приводится ниже:  
class Super:  
 def method(self):  
 print(‘in Super.method’) # Поведение по умолчанию  
 def delegate(self):  
 self.action() # Ожидаемый метод  
   
class Inheritor(Super): # Наследует методы, как они есть  
 pass

778   
Глава 28. Подробнее о программировании классов   
class Replacer(Super): # Полностью замещает method  
 def method(self):  
 print(‘in Replacer.method’)  
   
class Extender(Super): # Расширяет поведение метода method   
 def method(self):  
 print(‘starting Extender.method’)  
 Super.method(self)  
 Print(‘ending Extender.method’)  
   
class Provider(Super): # Определяет необходимый метод  
 def action(self):  
 print(‘in Provider.action’)  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 for klass in (Inheritor, Replacer, Extender):  
 print(‘\n’ + klass.\_\_name\_\_ + ‘...’)  
 klass().method()  
 print(‘\nProvider...’)  
 x = Provider()  
 x.delegate()  
Здесь следует отметить несколько моментов. Программный код тестирования   
модуля в конце примера создает экземпляры трех разных классов в цикле for.   
Поскольку классы – это объекты, можно поместить их в кортеж и создавать   
экземпляры единообразным способом (подробнее об этой идее рассказывается   
ниже). Кроме всего прочего, классы, как и модули, имеют атрибут \_\_name\_\_ – он   
содержит строку с именем класса, указанным в заголовке инструкции class.   
Ниже показано, что произойдет, если запустить файл:  
% python specialize.py  
   
Inheritor...  
in Super.method  
   
Replacer...  
in Replacer.method  
   
Extender...  
starting Extender.method  
in Super.method  
ending Extender.method  
   
Provider...  
in Provider.action  
Абстрактные суперклассы  
Обратите внимание, как работает класс Provider в предыдущем примере. Ког-  
да через экземпляр класса Provider вызывается метод delegate, инициируются   
две независимые процедуры поиска:  
1. При вызове x.delegate интерпретатор отыскивает метод delegate в классе Su-  
per, начиная поиск от экземпляра класса Provider и двигаясь вверх по дереву   
наследования. Экземпляр x передается методу в виде аргумента self, как   
обычно.

Наследование   
779  
2. Внутри метода Super.delegate выражение self.action приводит к запуску но-  
вого, независимого поиска в дереве наследования, начиная от экземпляра   
self и дальше вверх по дереву. Поскольку аргумент self ссылается на экзем-  
пляр класса Provider, метод action будет найден в подклассе Provider.  
Такой способ «восполнения пробелов» в реализации – обычное дело для плат-  
форм ООП. По крайней мере, в терминах метода delegate такие суперклассы,   
как в этом примере, иногда называют абстрактными суперклассами – клас-  
сы, которые предполагают, что часть их функциональности будет реализована   
их подклассами. Если ожидаемый метод не определен в подклассе, интерпре-  
татор возбудит исключение с сообщением о неопределенном имени, когда по-  
иск в дереве наследования завершится неудачей.   
Разработчики классов иногда делают такие требования к подклассам более   
очевидными с помощью инструкций assert или возбуждая встроенное исклю-  
чение NotImplementedError с помощью инструкции raise (более подробно об ин-  
струкциях, которые могут возбуждать исключения, мы поговорим в следую-  
щей части книги). Ниже приводится короткий пример приема, основанного на   
применении инструкции assert:  
class Super:  
 def delegate(self):  
 self.action()  
 def action(self):  
 assert False, ‘action must be defined!’ # При вызове этой версии  
   
>>> X = Super()  
>>> X.delegate()  
AssertionError: action must be defined!  
Мы познакомимся с инструкцией assert в главах 32 и 33, а пока лишь замечу,   
что если выражение возвращает ложь, она возбуждает исключение с указан-  
ным сообщением об ошибке. В данном случае выражение всегда возвращает   
ложь, чтобы вызвать появление об ошибке, если метод не будет переопределен   
и поиск по дереву наследования остановится на этой версии. В некоторых клас-  
сах, напротив, в таких методах-заглушках исключение NotImplementedError воз-  
буждается напрямую.   
class Super:  
 def delegate(self):  
 self.action()  
 def action(self):  
 raise NotImplementedError(‘action must be defined!’)  
   
>>> X = Super()  
>>> X.delegate()  
NotImplementedError: action must be defined!  
При работе с экземплярами подклассов мы так же будем получать исключе-  
ния, если эти подклассы не обеспечат собственную реализацию ожидаемого   
метода, замещающего метод в суперклассе:  
>>> class Sub(Super): pass  
...  
>>> X = Sub()  
>>> X.delegate()

780   
Глава 28. Подробнее о программировании классов   
NotImplementedError: action must be defined!  
>>> class Sub(Super):  
... def action(self): print(‘spam’)  
...  
>>> X = Sub()  
>>> X.delegate()  
spam  
Более реалистичный пример использования концепций, представленных   
в этом разделе, вы найдете в упражнении «Классификация животных в зооло-  
гии» (упражнение 8) в конце главы 31 и в решении этого упражнения в разделе   
«Часть VI, Классы и ООП» (приложение B). Такое частичное наследование яв-  
VI, Классы и ООП» (приложение B). Такое частичное наследование яв-  
, Классы и ООП» (приложение B). Такое частичное наследование яв-  
B). Такое частичное наследование яв-  
). Такое частичное наследование яв-  
ляется традиционным способом введения в ООП, но оно постепенно исчезает из   
арсенала многих разработчиков.  
Абстрактные суперклассы в Python 2.6 и 3.0  
В версиях Python 2.6 и 3.0 абстрактные суперклассы (они же «абстрактные   
базовые классы»), представленные в предыдущем разделе, которые требуют,   
чтобы подклассы переопределяли некоторые методы, могут быть реализованы   
с применением специальной синтаксической конструкции определения клас-  
са. Способ определения абстрактного суперкласса зависит от версии интер-  
претатора. В Python 3.0 для этих целей используется именованный аргумент   
в заголовке инструкции class и специальный декоратор @abstract методов. Обе   
конструкции мы будем подробно рассматривать далее, в этой книге:  
from abc import ABCMeta, abstractmethod  
   
class Super(metaclass=ABCMeta):  
 @abstractmethod  
 def method(self, ...):  
 pass  
В Python 2.6 вместо именованного аргумента в заголовке инструкции class ис-  
пользуется атрибут класса:  
class Super:  
 \_\_metaclass\_\_ = ABCMeta  
 @abstractmethod  
 def method(self, ...):  
 pass  
В любом случае результат получается одним и тем же – мы лишены возмож-  
ности создавать экземпляры, если метод не будет определен ниже в дереве   
классов. Ниже приводится пример абстрактного суперкласса, реализованного   
в версии 3.0, эквивалентный примеру в предыдущем разделе:  
>>> from abc import ABCMeta, abstractmethod  
>>>  
>>> class Super(metaclass=ABCMeta):  
... def delegate(self):  
... self.action()  
... @abstractmethod  
... def action(self):  
... pass

Пространства имен: окончание истории   
781  
...  
>>> X = Super()  
TypeError: Can’t instantiate abstract class Super with abstract methods action  
   
>>> class Sub(Super): pass  
...  
>>> X = Sub()  
TypeError: Can’t instantiate abstract class Sub with abstract methods action  
   
>>> class Sub(Super):  
... def action(self): print(‘spam’)  
...  
>>> X = Sub()  
>>> X.delegate()  
spam  
Реализованный таким способом класс с абстрактным методом не может ис-  
пользоваться для создания экземпляров (то есть нам не удастся создать эк-  
земпляр вызовом этого класса), если все абстрактные методы не будут реали-  
зованы в подклассах. Хотя при такой реализации объем программного кода   
увеличивается, тем не менее она имеет свои преимущества – ошибки из-за от-  
сутствующих методов будут появляться при попытке создать экземпляр клас-  
са, а не позднее, при попытке вызвать отсутствующий метод. Данная возмож-  
ность может использоваться для построения ожидаемого интерфейса, полнота   
реализации которого будет автоматически проверяться в клиентских классах.  
К сожалению, этот прием основан на использовании двух специализирован-  
ных инструментов языка, с которыми мы еще не встречались, – на декора-  
торах функций, которые будут представлены в главе 31 и более полно будут   
рассматриваться в главе 38, и объявлениях метаклассов, которые будут упо-  
минаться в главе 31 и подробно рассматриваться в главе 39. Поэтому здесь мы   
не будем углубляться в обсуждение этих особенностей. Более подробную ин-  
формацию по этой теме можно найти в стандартных руководствах по языку   
Python, а также в исходных текстах предопределенных суперклассов.  
Пространства имен: окончание истории   
Теперь, когда мы уже исследовали объекты классов и экземпляров, повество-  
вание о пространствах имен в языке Python можно считать завершенным. Для   
справки я напомню здесь все правила, используемые при разрешении имен.   
Первое, что вам нужно запомнить: квалифицированные и неквалифицирован-  
ные имена интерпретируются по-разному, и некоторые области видимости слу-  
жат для инициализации пространств имен объектов:  
 •  
Неквалифицированные имена (например, X) располагаются в областях ви-  
димости.  
 •  
Квалифицированные имена атрибутов (например, object.X) принадлежат   
пространствам имен объектов.  
 •  
Некоторые области видимости инициализируют пространства имен объек-  
тов (в модулях и классах).

782   
Глава 28. Подробнее о программировании классов   
Простые имена: глобальные,   
пока не выполняется присваивание  
Поиск неквалифицированных простых имен выполняется в соответствии   
с правилом лексической видимости LEGB, выведенном для функций в главе 17:  
Присваивание (X = value)  
Операция присваивания делает имена локальными: создает или изменяет   
имя X в текущей локальной области видимости, если имя не объявлено гло-  
бальным.  
Ссылка (X)  
Пытается отыскать имя X в текущей локальной области видимости, затем   
в области видимости каждой из вмещающих функций, затем в текущей гло-  
бальной области видимости и, наконец, во встроенной области видимости.  
Имена атрибутов: пространства имен объектов  
Квалифицированные имена атрибутов ссылаются на атрибуты конкретных   
объектов и к ним применяются правила, предназначенные для модулей и клас-  
сов. Для объектов классов и экземпляров эти правила дополняются включени-  
ем процедуры поиска в дереве наследования:  
Присваивание (object.X = value)  
Создает или изменяет атрибут с именем X в пространстве имен объекта ob-  
ject, и ничего больше. Восхождение по дереву наследования происходит   
только при попытке получить ссылку на атрибут, но не при выполнении   
операции присваивания.  
Ссылка (object.X)  
Для объектов, созданных на основе классов, поиск атрибута X производится   
сначала в объекте object, затем во всех классах, расположенных выше в де-  
реве наследования. В случае объектов, которые создаются не из классов, та-  
ких как модули, атрибут X извлекается непосредственно из объекта object.  
«Дзен» пространств имен в Python: классификация   
имен происходит при присваивании  
Из-за различий в процедурах поиска простых и составных имен и нескольких   
уровней поиска в обеих процедурах иногда бывает трудно сказать, где будет   
найдено имя. В языке Python место, где выполняется присваивание, имеет   
крайне важное значение – оно полностью определяет область видимости или   
объект, где будет размещаться имя. Файл manynames.py, ниже, иллюстрирует,   
как эти принципы переводятся в программный код, и обобщает идеи, касаю-  
щиеся пространств имен, с которыми мы встречались на протяжении книги:  
# manynames.py  
X = 11 # Глобальное (в модуле) имя/атрибут (X, или manynames.X)  
   
def f():  
 print(X) # Обращение к глобальному имени X (11)

Пространства имен: окончание истории   
783  
def g():  
 X = 22 # Локальная (в функции) переменная (X, скрывает имя X в модуле)  
 Print(X)  
   
class C:  
 X = 33 # Атрибут класса (C.X)  
 def m(self):  
 X = 44 # Локальная переменная в методе (X)  
 self.X = 55 # Атрибут экземпляра (instance.X)  
В этом файле пять раз выполняется присваивание одному и тому же имени X.   
Однако, так как присваивание выполняется в пяти разных местах, все пять   
имен X в этой программе представляют совершенно разные переменные. Сверху   
вниз присваивание имени X приводит к созданию: атрибута модуля (11), локаль-  
ной переменной в функции (22), атрибута класса (33), локальной переменной   
в методе (44) и атрибута экземпляра (55). Все пять переменных имеют одина-  
ковые имена, однако они создаются в разных местах программного кода или   
в разных объектах, что делает их уникальными переменными.  
Не следует торопиться тщательно изучать этот пример, потому что в нем со-  
браны идеи, которые мы исследовали на протяжении последних нескольких   
частей этой книги. Когда до вас дойдет его смысл, вы достигнете своего рода   
нирваны пространств имен в языке Python. Конечно, существует и другой путь   
к нирване – просто запустите программу и посмотрите, что произойдет. Ниже   
приводится остаток этого файла, где создается экземпляр и выводятся значе-  
ния всех имеющихся переменных X:  
# manynames.py, продолжение  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 print(X) # 11: модуль (за пределами файла manynames.X)  
 f() # 11: глобальная  
 g() # 22: локальная  
 print(X) # 11: переменная модуля не изменилась  
   
 obj = C() # Создать экземпляр  
 print(obj.X) # 33: переменная класса, унаследованная экземпляром  
   
 obj.m() # Присоединить атрибут X к экземпляру  
 print(obj.X) # 55: экземпляр  
 print(C.X) # 33: класс (она же obj.X, если в экземпляре нет X)  
   
 #print(C.m.X) # ОШИБКА: видима только в методе  
 #print(g.X) # ОШИБКА: видима только в функции  
В комментариях отмечено, что будет выведено на экран после запуска этого   
файла, – прочитайте их, чтобы увидеть, к какой переменной X выполняется   
обращение в том или ином случае. Обратите также внимание, что мы можем   
добраться до атрибута класса (C.X), но мы никогда не сможем получить доступ   
к локальным переменным в функциях или методах, находясь за пределами со-  
ответствующих инструкций def. Локальные переменные видимы только про-  
граммному коду внутри инструкции def и существуют в памяти только во вре-  
мя выполнения функции или метода.  
Некоторые из имен, определяемых этим файлом, видимы и за пределами фай-  
ла, в других модулях, но не забывайте, чтобы получить доступ к именам в дру-

784   
Глава 28. Подробнее о программировании классов   
гом файле, мы всегда должны сначала выполнить операцию импортирования –   
в конце концов, в этом заключается главная особенность модулей.  
# otherfile.py  
   
import manynames  
   
X = 66  
print(X) # 66: здешняя глобальная переменная  
print(manynames.X) # 11: глобальная, ставшая атрибутом в результате импорта  
   
manynames.f() # 11: X в manynames, не здешняя глобальная!  
manynames.g() # 22: локальная в функции, в другом файле  
   
print(manynames.C.X) # 33: атрибут класса в другом модуле  
I = manynames.C()  
print(I.X) # 33: все еще атрибут класса  
I.m()  
print(I.X) # 55: а теперь атрибут экземпляра!  
Обратите внимание, что manynames.f() выводит значение переменной X из модуля   
manynames, а не переменной из текущего модуля – область видимости всегда опре-  
деляется местоположением инструкции присваивания в программном коде (то   
есть лексически) и не зависит от того, что импортируется и куда импортиру-  
ется. Кроме того, обратите внимание, что собственный атрибут X в экземпляре   
отсутствовал, пока не был вызван метод I.m(), – атрибуты, как и любые другие   
переменные, появляются на свет во время операции присваивания, а не до нее.   
Обычно атрибуты экземпляра создаются за счет присваивания им начальных   
значений в конструкторе \_\_init\_\_, но это не единственная возможность.  
Наконец, как мы узнали в главе 17, с помощью инструкций global и (в Py-  
Py-  
thon 3.0) nonlocal функции могут изменять переменные, находящиеся за их   
пределами, – эти инструкции не только обеспечивают доступ к переменным   
для записи, но и изменяют правила привязки инструкций присваивания   
к пространствам имен:  
X = 11 # Глобальная в модуле  
   
def g1():  
 print(X) # Ссылка на глобальную переменную в модуле  
   
def g2():  
 global X  
 X = 22 # Изменит глобальную переменную в модуле  
   
def h1():  
 X = 33 # Локальная в функции  
 def nested():  
 print(X) # Ссылка на локальную переменную в объемлющей функции  
   
def h2():  
 X = 33 # Локальная в функции  
 def nested():  
 nonlocal X # Инструкция из Python 3.0  
 X = 44 # Изменит локальную переменную в объемлющей функции  
Конечно, вы не должны использовать одно и то же имя для обозначения всех   
переменных в своем сценарии! Но этот пример демонстрирует, что если даже

Пространства имен: окончание истории   
785  
вы поступите так, пространства имен в языке Python предотвратят случайный   
конфликт имен, используемых в одном контексте, с именами, используемыми   
в другом контексте.  
Словари пространств имен  
В главе 22 мы узнали, что пространства имен модулей фактически реализова-  
ны как словари и доступны в виде встроенного атрибута \_\_dict\_\_. То же отно-  
сится к объектам классов и экземпляров: обращение к квалифицированному   
имени атрибута фактически является операцией доступа к элементу словаря,   
а механизм наследования атрибута работает лишь как поиск в связанных сло-  
варях. Фактически объекты экземпляра и класса – это в значительной сте-  
пени просто словари со ссылками, ведущими вглубь интерпретатора. Интер-  
претатор Python обеспечивает возможность доступа к этим словарям, а также   
к ссылкам между ними для использования в особых случаях (например, при   
создании инструментальных средств).  
Чтобы понять внутреннее устройство атрибутов, давайте с помощью инте-  
рактивной оболочки проследим, как растут словари пространств имен, когда   
в игру вступают классы. Более простой вариант этого примера мы уже видели   
в главе 26, но теперь мы знаем гораздо больше о методах и суперклассах, поэто-  
му расширим его немного. Сначала определим суперкласс и подкласс с метода-  
ми, которые сохраняют данные в своих экземплярах:  
>>> class super:  
... def hello(self):  
... self.data1 = ‘spam’  
...  
>>> class sub(super):  
... def hola(self):  
... self.data2 = ‘eggs’  
...  
Когда мы создаем экземпляр подкласса, он начинает свое существование с пу-  
стым словарем пространства имен, но имеет ссылку на класс, стоящий выше   
в дереве наследования. Фактически дерево наследования доступно в виде спе-  
циальных атрибутов, которые вы можете проверить. Экземпляры обладают   
атрибутом \_\_class\_\_, который ссылается на класс, а классы имеют атрибут   
\_\_bases\_\_, который является кортежем, содержащим ссылки на суперклассы   
выше в дереве наследования (я выполнял этот пример в Python 3.0 – в вер-  
Python 3.0 – в вер-  
 3.0 – в вер-  
сии 2.6 формат вывода и имена некоторых внутренних атрибутов немного от-  
личаются):  
>>> X = sub()  
>>> X.\_\_dict\_\_ # Словарь пространства имен экземпляра  
{}  
   
>>> X.\_\_class\_\_ # Класс экземпляра  
<class ‘\_\_main\_\_.sub’>  
   
>>> sub.\_\_bases\_\_ # Суперклассы данного класса  
(<class ‘\_\_main\_\_.super’>,)  
   
>>> super.\_\_bases\_\_ # В Python 2.6 возвращает пустой кортеж ()  
(<class ‘object’>,)

786   
Глава 28. Подробнее о программировании классов   
Так как в классах выполняется присваивание атрибутам аргумента self, тем   
самым они заполняют объекты экземпляров, то есть атрибуты включаются   
в словари пространств имен экземпляров, а не классов. В пространство имен   
объекта экземпляра записываются данные, которые могут отличаться для раз-  
ных экземпляров, и аргумент self является точкой входа в это пространство   
имен:  
>>> Y = sub()  
   
>>> X.hello()  
>>> X.\_\_dict\_\_  
{‘data1’: ‘spam’}  
   
>>> X.hola()  
>>> X.\_\_dict\_\_  
{‘data1’: ‘spam’, ‘data2’: ‘eggs’}  
   
>>> sub.\_\_dict\_\_.keys()  
[‘\_\_module\_\_’, ‘\_\_doc\_\_’, ‘hola’]  
   
>>> super.\_\_dict\_\_.keys()  
[‘\_\_dict\_\_’, ‘\_\_module\_\_’, ‘\_\_weakref\_\_’, ‘hello’, ‘\_\_doc\_\_’>]  
   
>>> Y.\_\_dict\_\_  
{}  
Обратите внимание на имена в словарях классов, содержащие символы подчер-  
 внимание на имена в словарях классов, содержащие символы подчер-  
внимание на имена в словарях классов, содержащие символы подчер-  
 на имена в словарях классов, содержащие символы подчер-  
на имена в словарях классов, содержащие символы подчер-  
 имена в словарях классов, содержащие символы подчер-  
имена в словарях классов, содержащие символы подчер-  
 в словарях классов, содержащие символы подчер-  
в словарях классов, содержащие символы подчер-  
 словарях классов, содержащие символы подчер-  
словарях классов, содержащие символы подчер-  
 классов, содержащие символы подчер-  
классов, содержащие символы подчер-  
кивания, – эти имена определяются интерпретатором автоматически. Боль-  
шинство из них обычно не используются в программах, но существуют такие   
инструменты, которые используют некоторые из этих имен (например, \_\_doc\_\_   
хранит строки документирования, обсуждавшиеся в главе 15).  
Обратите также внимание, что второй экземпляр Y, созданный в начале сеан-  
са, по-прежнему имеет пустой словарь пространства имен, несмотря на то, что   
словарь экземпляра X заполнялся инструкциями присваивания в методах. На-  
помню еще раз, что у каждого экземпляра имеется свой, независимый словарь,   
который изначально пуст и может быть заполнен совершенно другими атрибу-  
тами, чем пространства имен других экземпляров того же самого класса.  
Так как атрибуты фактически являются ключами словаря, существует два   
способа получать и изменять их значения – по квалифицированным именам   
или индексированием по ключу:  
>>> X.data1, X.\_\_dict\_\_[‘data1’]  
(‘spam’, ‘spam’)  
   
>>> X.data3 = ‘toast’  
>>> X.\_\_dict\_\_  
{‘data1’: ‘spam’, ‘data3’: ‘toast’, ‘data2’: ‘eggs’}  
   
>>> X.\_\_dict\_\_[‘data3’] = ‘ham’  
>>> X.data3  
‘ham’  
Однако такая эквивалентность применяется только к атрибутам, фактически   
присоединенным к экземпляру. Так как обращение по квалифицированному   
имени также вызывает запуск процедуры поиска в дереве наследования, такой   
способ может обеспечить доступ к атрибутам, которые нельзя получить индек-

Пространства имен: окончание истории   
787  
сированием словаря. Например, унаследованный атрибут X.hello недоступен   
через выражение X.\_\_dict\_\_[‘hello’].  
Наконец, ниже показано, что дает применение функции dir, с которой мы   
встречались в главах 4 и 15, к объектам классов и экземпляров. Эта функция   
применяется к объектам, имеющим атрибуты: dir(object) напоминает вызов   
object.\_\_dict\_\_.keys(). Однако обратите внимание, что функция dir сортирует   
свой список и включает в него некоторые системные атрибуты� начиная с вер-  
сии Python 2.2, функция dir также автоматически собирает унаследованные   
атрибуты, а в версии 3.0 она добавляет в перечень имена, унаследованные от   
класса object, который является суперклассом для всех классов:1  
>>> X.\_\_dict\_\_, Y.\_\_dict\_\_  
{ ({‘data1’: ‘spam’, ‘data3’: ‘ham’, ‘data2’: ‘eggs’}, {})  
>>> list(X.\_\_dict\_\_.keys()) # Необходимо в Python 3.0  
[‘data1’, ‘data3’, ‘data2’]  
   
# В Python 2.6  
   
>>> dir(X)  
[‘\_\_doc\_\_’, ‘\_\_module\_\_’, ‘data1’, ‘data2’, ‘data3’, ‘hello’, ‘hola’]  
>>> dir(sub)  
[‘\_\_doc\_\_’, ‘\_\_module\_\_’, ‘hello’, ‘hola’]  
>>> dir(super)  
[‘\_\_doc\_\_’, ‘\_\_module\_\_’, ‘hello’]  
   
# В Python 3.0:  
   
>>> dir(X)  
[‘\_\_class\_\_’, ‘\_\_delattr\_\_’, ‘\_\_dict\_\_’, ‘\_\_doc\_\_’, ‘\_\_eq\_\_’, ‘\_\_format\_\_’,  
...часть строк опущена...  
‘data1’, ‘data2’, ‘data3’, ‘hello’, ‘hola’]  
   
>>> dir(sub)  
[‘\_\_class\_\_’, ‘\_\_delattr\_\_’, ‘\_\_dict\_\_’, ‘\_\_doc\_\_’, ‘\_\_eq\_\_’, ‘\_\_format\_\_’,  
...часть строк опущена...  
‘hello’, ‘hola’]  
   
>>> dir(super)  
[‘\_\_class\_\_’, ‘\_\_delattr\_\_’, ‘\_\_dict\_\_’, ‘\_\_doc\_\_’, ‘\_\_eq\_\_’, ‘\_\_format\_\_’,  
...часть строк опущена...  
‘hello’  
]  
Поэкспериментируйте самостоятельно с этими специальными атрибутами,   
чтобы получить представление о том, как в действительности ведется работа   
с атрибутами. Даже если вы никогда не будете использовать их в своих про-  
граммах, понимание того, что пространства имен – это всего лишь обычные   
1   
Содержимое словарей атрибутов и результаты вызова функции dir могут отличать-  
ся. Например, т.к. теперь интерпретатор позволяет встроенным типам классифици-  
ровать себя как классы, для встроенных типов функция dir включает информацию   
о методах перегрузки операторов, точно так же, как и для классов, определяемых   
пользователем, в Python 3.0. Вообще, имена атрибутов, начинающиеся и завершаю-  
щиеся двумя символами подчеркивания, являются особенностью интерпретатора.   
Подклассы типов будут рассматриваться в главе 31.

788   
Глава 28. Подробнее о программировании классов   
словари, поможет лишить покрова таинственности само понятие пространств   
имен.  
Ссылки на пространства имен  
В предыдущем разделе были представлены специальные атрибуты экземпляра   
и класса \_\_class\_\_ и \_\_bases\_\_, но не объяснялось, зачем они могут понадобить-  
ся. В двух словах, эти атрибуты позволяют осматривать иерархии наследова-  
ния в вашем программном коде. Например, их можно использовать для ото-  
бражения дерева классов на экране, как в следующем примере:  
# classtree.py  
   
“””  
Выполняет обход дерева наследования снизу вверх, используя ссылки на пространства   
имен, и отображает суперклассы с отступами  
“””  
   
def classtree(cls, indent):  
 print(‘.’ \* indent + cls.\_\_name\_\_) # Вывести имя класса  
 for supercls in cls.\_\_bases\_\_: # Рекурсивный обход всех суперклассов  
 classtree(supercls, indent+3) # Каждый суперкласс может быть посещен   
 # более одного раза  
def instancetree(inst):  
 print(‘Tree of’, inst) # Показать экземпляр  
 classtree(inst.\_\_class\_\_, 3) # Взойти к его классу  
   
def selftest():  
 class A: pass  
 class B(A): pass  
 class C(A): pass  
 class D(B,C): pass  
 class E: pass  
 class F(D,E): pass  
   
 instancetree(B())  
 instancetree(F())  
   
if \_\_name\_\_ == ‘\_\_main\_\_’: selftest()  
Функция classtree в этом сценарии является рекурсивной – она выводит имя   
класса, используя атрибут \_\_name\_\_, и затем начинает подъем к суперклассам,   
вызывая саму себя. Это позволяет функции выполнять обход деревьев клас-  
сов произвольной формы – в процессе рекурсии выполняется подъем по дереву   
и заканчивается по достижении корневых суперклассов, у которых атрибут   
\_\_bases\_\_ пуст.   
Большую часть этого файла занимает программный код самотестирования –   
если запустить файл как самостоятельный сценарий, он построит пустое дере-  
во классов, создаст в нем два экземпляра и выведет структуры классов, соот-  
ветствующие им:  
C:\misk> c:\python26\python classtree.py  
Tree of <\_\_main\_\_.B instance at 0x02557328>  
...B  
......A  
Tree of <\_\_main\_\_.F instance at 0x02557328>

Пространства имен: окончание истории   
789  
...F  
......D  
.........B  
............A  
.........C  
............A  
......E  
При работе под управлением Python 3.0 в дерево классов будет включен су-  
Python 3.0 в дерево классов будет включен су-  
 3.0 в дерево классов будет включен су-  
перкласс всех объектов object, который автоматически добавляется в список   
суперклассов, когда он пуст, потому что все классы в Python 3.0 относятся   
к классам «нового стиля» (подробнее об этом рассказывается в главе 31):  
C:\misc> c:\python30\python classtree.py  
Tree of <\_\_main\_\_.B object at 0x02810650>  
...B  
......A  
.........object  
Tree of <\_\_main\_\_.F object at 0x02810650>  
...F  
......D  
.........B  
............A  
...............object  
.........C  
............A  
...............object  
......E  
.........object  
Отступы, отмеченные точками, обозначают высоту в дереве классов. Конечно,   
мы могли бы улучшить формат вывода и даже отобразить дерево в графиче-  
ском интерфейсе. Мы можем импортировать эти функции везде, где нам может   
потребоваться быстро отобразить дерево классов:  
C:\misc> c:\python30\python  
>>> class Emp: pass  
...  
>>> class Person(Emp): pass  
...  
>>> bob = Person()  
   
>>> import classtree  
>>> classtree.instancetree(bob)  
Tree of <\_\_main\_\_.Person instance at 0x028203B0>  
...Person  
......Emp  
.........object  
Независимо от того, будете вы создавать и использовать нечто подобное в своей   
практике или нет, этот пример демонстрирует один из многих способов исполь-  
зования специальных атрибутов, которые создаются внутренними механизма-  
ми интерпретатора. Еще один пример вы увидите в разделе «Множественное   
наследование: классы-смеси», в главе 30, где мы с помощью этого же приема   
реализуем вывод атрибутов всех объектов в дереве классов. В последней части   
книги мы снова вернемся к этой теме, когда будем рассматривать способы соз-

790   
Глава 28. Подробнее о программировании классов   
дания частных атрибутов, проверку аргументов и многое другое. Доступность   
внутренних особенностей реализации является мощным оружием в руках про-  
граммиста, однако оно требуется далеко не всем.  
Еще раз о строках документирования  
Пример модуля в предыдущем разделе содержит строку документирования,   
описывающую этот модуль, но точно так же они могут использоваться для   
описания компонентов классов. Строки документирования, которые мы под-  
робно рассматривали в главе 15, – это литералы строк, которые присутствуют   
на верхнем уровне различных структур и автоматически сохраняются интер-  
претатором в атрибутах \_\_doc\_\_ соответствующих им объектов. Строки доку-  
ментирования могут присутствовать в модулях, в инструкциях def, а также   
в определениях классов и методов.   
Теперь, когда мы ближе познакомились с классами и методами, можно изу-  
чить короткий, но емкий пример docstr.py – здесь демонстрируются места   
в программном коде, где могут появляться строки документирования. Все они   
могут представлять собой блоки в тройных кавычках:  
“I am: docstr.\_\_doc\_\_”  
   
def func(args):  
 “I am: docstr.func.\_\_doc\_\_”  
 pass  
   
class spam:  
 “I am: spam.\_\_doc\_\_ or docstr.spam.\_\_doc\_\_”  
 def method(self, arg):  
 “I am: spam.method.\_\_doc\_\_ or self.method.\_\_doc\_\_”  
 pass  
Основное преимущество строк документирования состоит в том, что их содер-  
жимое доступно во время выполнения. То есть, если текст был оформлен в виде   
строки документирования, можно будет обратиться к атрибуту \_\_doc\_\_ объек-  
та, чтобы получить его описание:  
>>> import docstr  
>>> docstr.\_\_doc\_\_  
‘I am: docstr.\_\_doc\_\_’  
   
>>> docstr.func.\_\_doc\_\_  
‘I am: docstr.func.\_\_doc\_\_’  
   
>>> docstr.spam.\_\_doc\_\_  
‘I am: spam.\_\_doc\_\_ or docstr.spam.\_\_doc\_\_’  
   
>>> docstr.spam.method.\_\_doc\_\_  
‘I am: spam.method.\_\_doc\_\_ or self.method.\_\_doc\_\_’  
В главе 15 также обсуждается PyDoc – инструмент, который позволяет форми-  
ровать отчеты из всех этих строк. Ниже приводится пример интерактивного   
сеанса в Python 2.6 (в версии Python 3.0 выводятся дополнительные атрибу-  
Python 2.6 (в версии Python 3.0 выводятся дополнительные атрибу-  
 2.6 (в версии Python 3.0 выводятся дополнительные атрибу-  
Python 3.0 выводятся дополнительные атрибу-  
 3.0 выводятся дополнительные атрибу-  
ты, унаследованные от класса object, который в модели классов «нового стиля»   
является суперклассом всех классов. Запустите его у себя в версии 3.0, чтобы   
увидеть дополнительные атрибуты, а дополнительную информацию об этих   
различиях вы найдете в главе 31):

Классы и модули   
791  
>>> help(docstr)  
Help on module docstr:  
   
NAME  
 docstr - I am: docstr.\_\_doc\_\_  
   
FILE  
 c:\misc\docstr.py  
   
CLASSES  
 spam  
   
 class spam  
 | I am: spam.\_\_doc\_\_ or docstr.spam.\_\_doc\_\_  
 |  
 | Methods defined here:  
 |  
 | method(self, arg)  
 | I am: spam.method.\_\_doc\_\_ or self.method.\_\_doc\_\_  
   
FUNCTIONS  
 func(args)  
 I am: docstr.func.\_\_doc\_\_  
Строки документирования доступны во время выполнения, но синтаксически   
они менее гибки, чем комментарии # (которые могут находиться в любом ме-  
сте программы). Обе формы – полезные инструменты, и любая документация   
к программе – это хорошо (при условии, что она точная). Вообще говоря, строки   
документирования лучше использовать для функционального описания (что   
делают объекты), а комментарии # – для небольших пояснений (описывающих,   
как действуют выражения).  
Классы и модули  
Мы завершаем эту главу кратким сравнением предметов обсуждения двух   
последних частей книги: модулей и классов. Так как оба представляют со-  
бой пространства имен, различия между ними бывает трудно заметить сразу.   
В двух словах:  
 •  
Модули  
 •  
Это пакеты данных и исполняемого кода.  
 •  
Создаются как файлы с программным кодом на языке Python или как   
расширения на языке C.  
 •  
Задействуются операцией импортирования.  
 •  
Классы  
 •  
Реализуют новые объекты.  
 •  
Создаются с помощью инструкции class.  
 •  
Задействуются операцией вызова.  
 •  
Всегда располагаются внутри модуля.  
Кроме того, классы поддерживают дополнительные возможности, недоступ-  
ные в модулях, такие как перегрузка операторов, создание множества экзем-  
пляров и наследование. Несмотря на то что и классы, и модули являются про-

792   
Глава 28. Подробнее о программировании классов   
странствами имен, к настоящему времени вы должны четко понимать, что   
между ними имеются существенные различия.  
В заключение  
В этой главе был предпринят второй, более глубокий тур по механизмам ООП   
в языке Python. Мы узнали еще больше о классах и методах, о наследовании   
и методах перегрузки операторов. Мы также закончили повествование о про-  
странствах имен в языке Python, расширив это понятие, чтобы охватить его   
применение к классам. По пути мы рассмотрели еще несколько дополнитель-  
ных концепций, таких как абстрактные суперклассы, атрибуты данных клас-  
са, словари пространств имен и ссылки на них, и вызов методов и конструкто-  
ров суперкласса вручную.   
Теперь, когда мы знаем все о программировании классов в языке Python, в сле-  
дующей главе мы обратимся к такому специфическому аспекту, как перегрузка   
операторов. После этого мы исследуем некоторые распространенные шаблоны   
проектирования, рассмотрим некоторые способы использования и комбиниро-  
вания классов для оптимизации многократного использования программного   
кода. Однако, прежде чем двинуться дальше, ответьте на обычные контроль-  
ные вопросы, чтобы освежить в памяти все, о чем говорилось в этой главе.  
Закрепление пройденного  
Контрольные вопросы  
1. Что такое абстрактный суперкласс?  
2. Что произойдет, когда простая инструкция присваивания появится на   
верхнем уровне в инструкции class?  
3. Зачем может потребоваться в классе вручную вызывать метод \_\_init\_\_ су-  
перкласса?  
4. Как можно расширить унаследованный метод вместо полного его замеще-  
ния?  
5. Назовите… столицу Ассирии.  
Ответы  
1. Абстрактный суперкласс – это класс, который вызывает методы, но не на-  
следует и не определяет их. Он ожидает, что методы будут реализованы   
в подклассах. Часто такой прием используется для обобщения классов, ког-  
да поведение будущих подклассов трудно предсказать заранее. Фреймвор-  
ки ООП также используют этот прием для выполнения операций, опреде-  
ляемых клиентом.  
2. Когда простой оператор присваивания (X = Y) появляется на верхнем уровне   
в инструкции class, он присоединяет к классу атрибут данных (Class.X). Как   
и все атрибуты класса, этот атрибут будет совместно использоваться всеми   
экземплярами. При этом атрибуты данных не являются вызываемыми ме-  
тодами.

Закрепление пройденного   
793  
3. Вручную вызывать метод \_\_init\_\_ суперкласса может потребоваться, ког-  
да класс определяет свой собственный конструктор \_\_init\_\_ и при этом не-  
обходимо, чтобы выполнялись действия, предусмотренные конструктором   
суперкласса. Интерпретатор Python автоматически вызывает только один   
конструктор – самый нижний в дереве наследования. Конструктор супер-  
класса вызывается через имя класса, и ему вручную передается аргумент   
self: Superclass.\_\_init\_\_(self, ...).  
4. Чтобы расширить унаследованный метод вместо полного его замещения,   
нужно переопределить его в подклассе и при этом вручную вызвать версию   
метода суперкласса из нового метода в подклассе. То есть вручную передать   
версии метода суперкласса аргумент self: Superclass.method(self, ...).  
5. Ашшур (или Калат-Шеркат), Калах (или Нимруд), короткое время был сто-  
лицей Дур-Шаррукин (или Хорсабад) и, наконец, Ниневия.

Глава 29.  
   
Перегрузка операторов  
Эта глава продолжает детальное исследование классов, фокусируясь на пере-  
грузке операторов. Перегрузку операторов мы коротко рассмотрели в предыду-  
щей главе, а здесь мы обсудим все более детально и рассмотрим несколько наи-  
более часто используемых методов перегрузки. Мы не будем демонстрировать   
все доступные методы перегрузки операторов, тем не менее те примеры, что бу-  
дут показаны, можно считать представительной выборкой, достаточно полно   
раскрывающей эту особенность классов в языке Python.  
Основы  
В действительности термин «перегрузка операторов» означает всего лишь пере-  
хватывание встроенных операций с помощью методов классов – интерпрета-  
тор автоматически вызывает эти методы при выполнении встроенных опера-  
ций над экземплярами классов, а методы должны возвращать значения, ко-  
торые будут интерпретироваться как результаты соответствующих операций.   
Ниже приводится краткий обзор ключевых идей, лежащих в основе механиз-  
ма перегрузки:  
 •  
Перегрузка операторов в языке Python позволяет классам участвовать   
в обычных операциях.  
 •  
Классы в языке Python могут перегружать все операторы выражений.  
 •  
Классы могут также перегружать такие операции, как вывод, вызов функ-  
ций, обращение к атрибутам и так далее.  
 •  
Перегрузка делает экземпляры классов более похожими на встроенные   
типы.  
 •  
Перегрузка заключается в реализации в классах методов со специальными   
именами.  
Другими словами, если в классе определен метод со специальным именем, ин-  
терпретатор автоматически будет вызывать его при выполнении соответствую-  
щей методу операции над экземплярами этого класса. Как мы уже знаем, мето-  
ды перегрузки операторов никогда не являются обязательными, и обычно для   
них не предусматривается реализация по умолчанию – если метод не реализо-  
ван в классе и не унаследован, это всего лишь означает, что класс не поддержи-

Закрепление пройденного   
795  
вает соответствующую операцию. Однако если эти методы используются, то   
они позволяют классам имитировать интерфейсы встроенных объектов и обе-  
спечивают их единообразие.  
Конструкторы и выражения: \_\_init\_\_ and \_\_sub\_\_  
Рассмотрим простой пример: класс Number в файле number.py, реализующий   
метод перегрузки операции создания экземпляра (\_\_init\_\_), а также метод реа-  
лизации операции вычитания (\_\_sub\_\_). Специальные методы, такие как эти,   
позволяют перехватывать и выполнять встроенные операции:  
class Number:  
 def \_\_init\_\_(self, start): # Вызов Number(start)  
 self.data = start  
 def \_\_sub\_\_(self, other): # Выражение: экземпляр - other  
 return Number(self.data - other) # Результат – новый экземпляр  
   
>>> from number import Number # Извлечь класс из модуля  
>>> X = Number(5) # Number.\_\_init\_\_(X, 5)  
>>> Y = X - 2 # Number.\_\_sub\_\_(X, 2)  
>>> Y.data # Y - новый экземпляр класса Number  
3  
Как уже обсуждалось ранее, конструктор \_\_init\_\_, присутствующий в этом   
примере, – это наиболее часто используемый метод перегрузки операторов   
в языке Python, потому что он присутствует в большинстве классов. В этой гла-  
Python, потому что он присутствует в большинстве классов. В этой гла-  
, потому что он присутствует в большинстве классов. В этой гла-  
ве мы изучим некоторые другие инструменты, связанные с перегрузкой, и рас-  
смотрим наиболее типичные примеры их использования.  
Общие методы перегрузки операторов  
Почти все, что можно делать с объектами встроенных типов, такими как целые   
числа и списки, можно реализовать и в классах – с помощью специальных ме-  
тодов перегрузки операторов. В табл. 29.1 перечислены наиболее часто исполь-  
зуемые, но на самом деле их намного больше. В действительности многие мето-  
ды перегрузки существуют в нескольких версиях (например, \_\_add\_\_, \_\_radd\_\_   
и \_\_iadd\_\_ для операции сложения), и в этом заключается основная причина   
такого большого их количества. Исчерпывающий список имен специальных   
методов вы найдете в других книгах, посвященных языку Python, и в справоч-  
Python, и в справоч-  
, и в справоч-  
ных руководствах.  
Таблица 29.1. Общие методы перегрузки операторов  
Метод  
Перегружает  
Вызывается  
\_\_init\_\_  
Конструктор  
При создании объекта:   
X = Class(args)  
\_\_del\_\_  
Деструктор  
При уничтожении объекта  
\_\_add\_\_  
Оператор +  
X + Y, X += Y,   
если отсутствует метод \_\_iadd\_\_  
\_\_or\_\_  
Оператор |   
(побитовое ИЛИ)  
X | Y, X |= Y,   
если отсутствует метод \_\_ior\_\_

796   
Глава 29. Перегрузка операторов   
Метод  
Перегружает  
Вызывается  
\_\_repr\_\_, \_\_  
str\_\_  
Вывод, преобразова-  
ние  
print(X), repr(X), str(X)  
\_\_call\_\_  
Вызовы функции  
X(\*args, \*\*kargs)  
\_\_getattr\_\_  
Обращение к атрибуту X.undefined  
\_\_setattr\_\_  
Присваивание   
атрибуту  
X.any = value  
\_\_delattr\_\_  
Удаление атрибута  
del X.any  
\_\_getattrib-  
ute\_\_  
Обращение к атрибуту X.any  
\_\_getitem\_\_  
Доступ к элементу по   
индексу, извлечение   
среза, итерации  
X[key], X[i:j], циклы for и другие   
конструкции итерации, при отсут-  
ствии метода \_\_iter\_\_  
\_\_setitem\_\_  
Присваивание эле-  
менту по индексу или   
срезу  
X[key] = value, X[i:j] = sequence  
\_\_delitem\_\_  
Удаление элемента   
по индексу или среза  
del X[key], del X[i:j]  
\_\_len\_\_  
Длина  
len(X), проверка истинности, если   
отсутствует метод \_\_bool\_\_  
\_\_bool\_\_  
Проверка логического   
значения  
bool(X) , проверка истинности (в вер-  
сии 2.6 называется \_\_nonzero\_\_)  
\_\_lt\_\_, \_\_gt\_\_,   
\_\_le\_\_, \_\_ge\_\_,  
\_\_eq\_\_, \_\_ne\_\_  
Сравнивание  
X < Y, X > Y, X <= Y, X >= Y, X == Y, X   
!= Y (или \_\_cmp\_\_, но только в 2.6)  
\_\_radd\_\_  
Правосторонний   
оператор +  
Не\_экземпляр + X  
\_\_iadd\_\_  
Добавление   
(увеличение)  
X += Y (в ином случае \_\_add\_\_)  
\_\_iter\_\_, \_\_  
next\_\_  
Итерационный   
контекст   
I=iter(X), next(I)� циклы for, опе-  
ратор in (если не определен метод   
\_\_contains\_\_), все типы генераторов,   
map(F, X) и другие (в версии 2.6 ме-  
тод \_\_next\_\_ называется next))  
\_\_contains\_\_  
Проверка   
на вхождение  
item in X (где X – любой итерируе-  
мый объект)  
\_\_index\_\_  
Целое число  
hex(X), bin(X) , oct(X) , O[X] , O[X:]   
(замещает методы \_\_oct\_\_, \_\_hex\_\_   
в Python 2)  
Таблица 29.1 (продолжение)

Доступ к элементам по индексу и извлечение срезов: \_\_getitem\_\_ и \_\_setitem\_\_   
797  
Метод  
Перегружает  
Вызывается  
\_\_enter\_\_,   
\_\_exit\_\_  
Менеджеры контек-  
стов (глава 33)  
with obj as var:  
\_\_get\_\_, \_\_  
set\_\_,  
\_\_delete\_\_  
Дескрипторы атрибу-  
тов (глава 37)  
X.attr, X.attr = value, del X.attr  
\_\_new\_\_  
Создание (глава 39)  
Вызывается при создании объектов,   
перед вызовом метода \_\_init\_\_  
Все методы перегрузки имеют имена, начинающиеся и заканчивающиеся дву-  
мя символами подчеркивания, что отличает их от других имен, которые вы   
обычно определяете в своих классах. Отображение операторов выражений или   
операций на методы со специальными именами предопределяется языком Py-  
Py-  
thon (и описывается в стандартном руководстве по языку). Например, по опре-  
 (и описывается в стандартном руководстве по языку). Например, по опре-  
делению языка оператор + всегда отображается на имя \_\_add\_\_ независимо от   
того, что в действительности делает метод \_\_add\_\_.  
Методы перегрузки операторов могут наследоваться от суперклассов, если они   
отсутствуют в самом классе, как и любые другие методы. Кроме того, методы   
перегрузки операторов являются необязательными – если какой-то метод не   
реализован, это лишь означает, что соответствующая ему операция не поддер-  
живается классом, а при попытке применить такую операцию возбуждается   
исключение. Некоторые встроенные операции, такие как вывод, имеют реали-  
зацию по умолчанию (в Python 3.0 они наследуются от класса object, являю-  
щегося суперклассом для всех объектов), но большинство операций будут вы-  
зывать исключение, если класс не предусматривает реализацию соответствую-  
щего метода.  
Большинство методов перегрузки операторов используются только при реше-  
нии специальных задач, когда необходимо, чтобы объекты имитировали пове-  
дение встроенных типов, однако конструктор \_\_init\_\_ присутствует в большин-  
стве классов, поэтому мы уделим ему особое внимание. Мы уже познакомились   
с конструктором \_\_init\_\_, который вызывается на этапе инициализации, и с   
несколькими другими, перечисленными в табл. 29.1. Теперь мы исследуем   
примеры использования некоторых других методов из таблицы.  
Доступ к элементам по индексу и извлечение   
срезов: \_\_getitem\_\_ и \_\_setitem\_\_  
Если метод \_\_getitem\_\_ присутствует в определении класса (или наследуется   
им), он автоматически будет вызываться интерпретатором в случае примене-  
ния операций индексирования к экземплярам. Когда экземпляр X появляется   
в выражении извлечения элемента по индексу, таком как X[i], интерпретатор   
Python вызывает метод \_\_getitem\_\_, наследуемый этим экземпляром, переда-  
вая методу объект X в первом аргументе и индекс, указанный в квадратных   
скобках, во втором аргументе. Например, следующий класс возвращает ква-  
драт значения индекса:  
>>> class Indexer:  
... def \_\_getitem\_\_(self, index):

798   
Глава 29. Перегрузка операторов   
... return index \*\* 2  
...  
>>> X = Indexer()  
>>> X[2] # Выражение X[i] вызывает X.\_\_getitem\_\_(i)  
4  
>>> for i in range(5):  
... print(X[i], end=’ ‘) # Вызывает \_\_getitem\_\_(X, i) в каждой итерации  
...  
0 1 4 9 16  
Извлечение срезов  
Интересно отметить, что метод \_\_getitem\_\_ вызывается не только при выпол-  
нении операции обращения к элементу по индексу, но и при извлечении сре-  
зов. Формально, встроенные типы обрабатывают операцию извлечения среза   
одинаково. Ниже приводится пример применения операции извлечения среза   
к списку, при этом используются верхняя и нижняя границы среза, а также   
шаг (подробно операция извлечения среза рассматривается в главе 7):  
>>> L = [5, 6, 7, 8, 9]  
>>> L[2:4] # Извлечение среза с использованием синтаксиса срезов  
[7, 8]  
>>> L[1:]  
[6, 7, 8, 9]  
>>> L[:-1]  
[5, 6, 7, 8]  
>>> L[::2]  
[5, 7, 9]  
Однако в действительности параметры среза определяются с помощью объек-  
та среза, который и передается реализации операции индексирования списка.   
Фактически вы всегда можете передать объект среза вручную – синтаксис сре-  
зов в значительной степени является всего лишь синтаксическим подсластите-  
лем для операции индексирования с применением объекта среза:  
>>> L[slice(2, 4)] # Извлечение среза с помощью объекта среза  
[7, 8]  
>>> L[slice(1, None)]  
[6, 7, 8, 9]  
>>> L[slice(None, -1)]  
[5, 6, 7, 8]  
>>> L[slice(None, None, 2)]  
[5, 7, 9]  
Эта особенность имеет значение для классов, реализующих метод \_\_getitem\_\_ ,–   
этот метод будет вызываться и для выполнения операций обращения к элемен-  
там по индексам (с целочисленным индексом), и для выполнения операций из-  
влечения срезов (с объектом среза). Наш класс в предыдущем примере не спосо-  
бен обрабатывать операцию извлечения среза, потому что его логика принимает   
лишь целочисленные индексы, однако такую возможность поддерживает сле-  
дующий класс. Когда метод вызывается для выполнения операции обращения   
к элементу по индексу, в аргументе передается целое число, как и прежде:  
>>> class Indexer:  
... data = [5, 6, 7, 8, 9]

Доступ к элементам по индексу и извлечение срезов: \_\_getitem\_\_ и \_\_setitem\_\_   
799  
... def \_\_getitem\_\_(self, index): # Вызывается при индексировании или   
... print(‘getitem:’, index) # извлечении среза  
... return self.data[index] # Выполняет индексирование   
... # или извлекает срез  
>>> X = Indexer()  
>>> X[0] # При индексировании \_\_getitem\_\_   
getitem: 0 # получает целое число  
5  
>>> X[1]  
getitem: 1  
6  
>>> X[-1]  
getitem: -1  
9  
Однако, когда метод вызывается для извлечения среза, он получает объект сре-  
за, который просто передается списку, встроенному в класс Indexer, в виде вы-  
ражения обращения по индексу:  
>>> X[2:4] # При извлечении среза \_\_getitem\_\_ получает объект среза  
getitem: slice(2, 4, None)  
[7, 8]  
>>> X[1:]  
getitem: slice(1, None, None)  
[6, 7, 8, 9]  
>>> X[:-1]  
getitem: slice(None, -1, None)  
[5, 6, 7, 8]  
>>> X[::2]  
getitem: slice(None, None, 2)  
[5, 7, 9]  
Метод \_\_setitem\_\_ присваивания элементу по индексу точно так же обслужи-  
присваивания элементу по индексу точно так же обслужи-  
 элементу по индексу точно так же обслужи-  
элементу по индексу точно так же обслужи-  
 по индексу точно так же обслужи-  
по индексу точно так же обслужи-  
 индексу точно так же обслужи-  
индексу точно так же обслужи-  
 точно так же обслужи-  
точно так же обслужи-  
 так же обслужи-  
так же обслужи-  
 же обслужи-  
же обслужи-  
 обслужи-  
обслужи-  
вает обе операции – присваивания элементу по индексу и присваивание срезу.   
В последнем случае он получает объект среза, который может передаваться   
другим операциям присваивания по индексу:  
def \_\_setitem\_\_(self, index, value): # Реализует присваивание   
 ... # по индексу или по срезу  
 self.data[index] = value # Приcваивание по индексу или по срезу  
Фактически метод \_\_getitem\_\_ может автоматически вызываться не только при   
выполнении операций индексирования или извлечения срезов, как описыва-  
ется в следующем разделе.  
Извлечение срезов и элементов по индексу в Python 2.6  
До появления Python 3.0 в классах можно было также определять мето-  
ды \_\_getslice\_\_ и \_\_setslice\_\_, предназначенные для выполнения опера-  
ций извлечения среза и присваивания срезу, – они получали границы   
среза и были предпочтительными способами реализации операций над   
срезами перед \_\_getitem\_\_ и \_\_setitem\_\_.

800   
Глава 29. Перегрузка операторов   
В версии 3.0 эти методы перегрузки операций над срезами были удале-  
ны, поэтому теперь для реализации обоих типов операций, с индексами   
и со срезами, должны использоваться методы \_\_getitem\_\_ и \_\_setitem\_\_,   
которые должны принимать в качестве аргументов не только целочис-  
ленные индексы, но и объекты срезов. В большинстве классов для этого   
не придется прибегать к каким-либо специальным приемам, потому что   
внутри этих методов допускается подставлять объекты срезов в ква-  
дратных скобках внутри других выражений с операцией индексирова-  
ния (как в нашем примере). Еще один пример реализации операций над   
срезами вы найдете в разделе «Проверка на вхождение: \_\_contains\_\_,   
\_\_iter\_\_ и \_\_getitem\_\_».  
Кроме того, не следует считать, что (возможно, неудачно названный) ме-  
тод \_\_index\_\_ в Python 3.0 имеет отношение к операции индексирова-  
ния, – этот метод возвращает целое число, представляющее экземпляр,   
и используется встроенными типами, которые выполняют преобразова-  
ние целых чисел в строку цифр:  
>>> class C:  
... def \_\_index\_\_(self):  
... return 255  
...  
>>> X = C()  
>>> hex(X) # Целочисленное значение  
‘0xff’  
>>> bin(X)  
‘0b11111111’  
>>> oct(X)  
‘0o377’  
Хотя этот метод не имеет отношения к реализации операции индексиро-  
вания, как метод \_\_getitem\_\_, тем не менее он также используется в опе-  
рациях, требующих целое число, включая и операцию индексирования:  
>>> (‘C’ \* 256)[255]  
‘C’  
>>> (‘C’ \* 256)[X] # X используется как индекс (не X[i])  
‘C’  
>>> (‘C’ \* 256)[X:] # X используется как индекс (не X[i:])  
‘C’  
В Python 2.6 этот метод действует точно так же, за исключением того,   
что он не вызывается встроенными функциями hex и oct (вместо этого   
в версии 2.6 данные функции используют методы перегрузки операто-  
ров \_\_hex\_\_ и \_\_oct\_\_).  
Итерации по индексам: \_\_getitem\_\_  
Здесь описывается прием, который не всегда очевиден для начинающих про-  
граммистов, но на практике может оказаться необычайно полезным. Инструк-  
ция for многократно применяет операцию индексирования к последовательно-

Итерации по индексам: \_\_getitem\_\_   
801  
сти, используя индексы от нуля и выше, пока не будет получено исключение   
выхода за границы. Благодаря этому метод \_\_getitem\_\_ представляет собой   
один из способов перегрузки итераций в языке Python – если этот метод реали-  
Python – если этот метод реали-  
 – если этот метод реали-  
зован, инструкции циклов for будут вызывать его на каждом шаге цикла, с по-  
стоянно увеличивающимся значением смещения. Это один из случаев, когда   
«купив один предмет, другой получаешь в подарок», – любой встроенный или   
определяемый пользователем объект, к которому применима операция индек-  
сирования, также может участвовать в итерациях:  
>>> class stepper:  
... def \_\_getitem\_\_(self, i):  
... return self.data[i]  
...  
>>> X = stepper() # X - это экземпляр класса stepper  
>>> X.data = “Spam”  
>>>  
>>> X[1] # Индексирование, вызывается \_\_getitem\_\_  
‘p’  
>>> for item in X: # Циклы for вызывают \_\_getitem\_\_  
... print(item, end=’ ‘) # Инструкция for индексирует элементы 0..N  
...  
S p a m  
Фактически это случай, когда, «купив один предмет, в подарок получаешь це-  
лую связку». Любой класс, поддерживающий циклы for, автоматически под-  
держивает все итерационные контексты, имеющиеся в языке Python, многие   
из которых мы видели в более ранних главах (другие итерационные контексты   
описываются в главе 14). Например, оператор проверки на принадлежность in,   
генераторы списков, встроенная функция map, присваивание списков и корте-  
жей и конструкторы типов также автоматически вызывают метод \_\_getitem\_\_,   
если он определен:  
>>> ‘p’ in X # Во всех этих случаях вызывается \_\_getitem\_\_  
True  
   
>>> [c for c in X] # Генератор списков  
[‘S’, ‘p’, ‘a’, ‘m’]  
   
>>> list(map(str.upper, X)) # Функция map (в версии 3.0  
[‘S’, ‘P’, ‘A’, ‘M’] # требуется использовать функцию list)  
   
>>> (a, b, c, d) = X # Присваивание последовательностей  
>>> a, c, d  
(‘S’, ‘a’, ‘m’)  
   
>>> list(X), tuple(X), ‘’.join(X)  
([‘S’, ‘p’, ‘a’, ‘m’], (‘S’, ‘p’, ‘a’, ‘m’), ‘Spam’)  
   
>>> X  
<\_\_main\_\_.stepper instance at 0x00A8D5D0>  
На практике этот прием может использоваться для создания объектов, ко-  
торые реализуют интерфейс последовательностей, и для добавления логики   
к операциям над встроенными типами – мы рассмотрим эту идею, когда будем   
расширять встроенные типы в главе 31.

802   
Глава 29. Перегрузка операторов   
Итераторы: \_\_iter\_\_ и \_\_next\_\_  
Прием, основанный на использовании метода \_\_getitem\_\_, представленный   
в предыдущем разделе, действительно работает, однако он используется опе-  
рациями, выполняющими итерации, в самом крайнем случае. В настоящее   
время все итерационные контексты в языке Python пытаются сначала ис-  
Python пытаются сначала ис-  
 пытаются сначала ис-  
пользовать метод \_\_iter\_\_, и только потом – метод \_\_getitem\_\_. То есть при вы-  
полнении обхода элементов объекта предпочтение отдается итерационному   
протоколу, с которым мы познакомились в главе 14, – если итерационный   
протокол не поддерживается объектом, вместо него используется операция   
индексирования. Вообще говоря, вы также должны отдавать предпочтение ме-  
тоду \_\_iter\_\_ – он обеспечивает более оптимальную поддержку итерационных   
контекстов, чем метод \_\_getitem\_\_.  
С технической точки зрения итерационные контексты вызывают встроенную   
функцию iter, чтобы определить наличие метода \_\_iter\_\_, который должен воз-  
вращать объект итератора. Если он предоставляется, то интерпретатор Python   
будет вызывать метод \_\_next\_\_ объекта итератора для получения элементов до   
тех пор, пока не будет возбуждено исключение StopIteration. Если метод \_\_iter\_\_   
отсутствует, интерпретатор переходит на использование схемы с применением   
метода \_\_getitem\_\_ и начинает извлекать элементы по индексам, пока не будет   
возбуждено исключение IndexError. Кроме того, для удобства предоставляется   
встроенная функция next, позволяющая выполнять итерации вручную: вызов   
next(I) – это то же самое, что вызов I.\_\_next\_\_().  
Примечание,  касающееся  различий  между  версиями: Как опи-  
сывалось в главе 14, в версии Python 2.6 метод I.\_\_next\_\_(), кото-  
рый только что был представлен, называется I.next(), а встроен-  
ная функция next(I) обеспечивает переносимый способ вызова   
этого метода: в версии 2.6 она вызовет метод I.next(), а в вер-  
сии 3.0 – метод I.\_\_next\_\_(). Во всех остальных отношениях ите-  
рации в версии 2.6 выполняются точно так же.  
Итераторы, определяемые пользователями  
В схеме с применением метода \_\_iter\_\_ классы реализуют итераторы простой   
реализацией итерационного протокола, представленного в главах 14 и 20 (за   
дополнительной информацией об итераторах возвращайтесь к этим главам).   
Например, в следующем файле iters.py определяется класс итератора, который   
возвращает квадраты чисел:  
class Squares:  
 def \_\_init\_\_(self, start, stop): # Сохранить состояние при создании  
 self.value = start - 1  
 self.stop = stop  
 def \_\_iter\_\_(self): # Возвращает итератор в iter()  
 return self  
 def \_\_next\_\_(self): # Возвращает квадрат в каждой итерации  
 if self.value == self.stop: # Также вызывается функцией next  
 raise StopIteration  
 self.value += 1  
 return self.value \*\* 2

Итераторы: \_\_iter\_\_ и \_\_next\_\_   
803  
% python  
>>> from iters import Squares  
>>> for i in Squares(1, 5): # for вызывает iter(), который вызывает \_\_iter\_\_()  
... print(i, end=’ ‘) # на каждой итерации вызывается \_\_next\_\_()  
...  
1 4 9 16 25  
Здесь объект итератора – это просто экземпляр self, поэтому метод \_\_next\_\_   
является частью этого класса. В более сложных ситуациях объект итератора   
может быть определен как отдельный класс и объект со своей собственной ин-  
формацией о состоянии, с целью поддержки нескольких активных итераций   
на одних и тех же данных (совсем скоро мы рассмотрим это на примере). Об   
окончании итераций интерпретатору сообщается с помощью инструкции raise   
(подробнее о возбуждении исключений рассказывается в следующей части   
книги). Итерации по встроенным типам можно также выполнять вручную:  
>>> X = Squares(1, 5) # Выполнение итераций вручную: эти действия выполняет   
 # инструкция цикла  
>>> I = iter(X) # iter вызовет \_\_iter\_\_  
>>> next(I) # next вызовет \_\_next\_\_  
1  
>>> next(I)  
4  
...часть строк опущена...  
>>> next(I)  
25  
>>> next(I) # Исключение можно перехватить с помощью инструкции try  
StopIteration  
Эквивалентная реализация с использованием \_\_getitem\_\_ может оказаться ме-  
нее естественной, потому что цикл for явно выполняет перебор всех смещений   
от нуля и выше� смещения, передаваемые методу, могут оказаться связаны   
с диапазоном воспроизводимых значений лишь косвенно (диапазон 0..N может   
потребоваться отображать на диапазон start..stop). Поскольку объекты, воз-  
вращаемые методом \_\_iter\_\_, явно манипулируют информацией о своем состо-  
янии и сохраняют ее между вызовами функции next, такая реализация может   
быть более универсальной, чем использование метода \_\_getitem\_\_.  
C другой стороны, итераторы, реализованные на основе метода \_\_iter\_\_, иногда   
могут оказаться более сложными и менее удобными, чем метод \_\_getitem\_\_. Но   
они действительно предназначены для итераций, а не для случайной индек-  
сации, – фактически они вообще не перегружают операцию индексирования:  
>>> X = Squares(1, 5)  
>>> X[1]  
AttributeError: Squares instance has no attribute ‘\_\_getitem\_\_’  
(AttributeError: экземпляр Squares не имеет атрибута ‘\_\_getitem\_\_’)  
Схема на основе метода \_\_iter\_\_ реализована также во всех остальных итераци-  
онных контекстах, к которым применим метод \_\_getitem\_\_ (проверка на вхож-  
дение, конструкторы, присваивание последовательностей и так далее). Одна-  
ко, в отличие от \_\_getitem\_\_, схема на основе метода \_\_iter\_\_ предназначена для   
выполнения обхода элементов один раз, а не несколько. Например, элементы   
класса Squares можно обойти всего один раз – для каждой последующей итера-  
ции необходимо будет создавать новый объект итератора:

804   
Глава 29. Перегрузка операторов   
>>> X = Squares(1, 5)  
>>> [n for n in X] # Получить все элементы  
[1, 4, 9, 16, 25]  
>>> [n for n in X] # Теперь объект пуст  
[]  
>>> [n for n in Squares(1, 5)] # Создать новый объект итератора  
[1, 4, 9, 16, 25]  
>>> list(Squares(1, 3))  
[1, 4, 9]  
Примечательно, что этот пример можно было бы реализовать проще, применив   
функции-генераторы (которые имеют отношение к итераторам и были пред-  
ставлены в главе 20):  
>>> def gsquares(start, stop):  
... for i in range(start, stop+1):  
... yield i \*\* 2  
...  
>>> for i in gsquares(1, 5): # или: (x \*\* 2 for x in range(1, 5))  
... print(i, end=’ ‘)  
...  
1 4 9 16 25  
В отличие от класса, функция автоматически сохраняет информацию о своем   
состоянии между итерациями. Конечно, для реализации такого искусственно-  
го примера можно было бы вообще не использовать ни один из этих приемов,   
а просто использовать цикл for, функцию map или генератор списков, чтобы соз-  
дать сразу весь список. Нередко самый лучший и самый быстрый способ в язы-  
ке Python оказывается еще и самым простым:  
>>> [x \*\* 2 for x in range(1, 6)]  
[1, 4, 9, 16, 25]  
Однако реализация на базе классов может оказаться лучше при моделирова-  
нии более сложных итераций, особенно когда возможность сохранения ин-  
формации о состоянии и наследование могут принести существенную выгоду.   
Один из таких случаев исследуется в следующем разделе.  
Несколько итераторов в одном объекте  
Ранее я упоминал, что объект итератора может быть определен как отдельный   
класс, со своей собственной информацией о состоянии, что обеспечивает под-  
держку протекания нескольких итерационных процессов с одним и тем же на-  
бором данных. Посмотрим, что происходит при выполнении обхода элементов   
встроенных типов, таких как строка:  
>>> S = ‘ace’  
>>> for x in S:  
... for y in S:  
... print(x + y, end=’ ‘)  
...  
aa ac ae ca cc ce ea ec ee  
Здесь внешний цикл получает итератор строки вызовом функции iter и каж-  
дый вложенный цикл делает то же самое, чтобы получить независимый ите-  
ратор. Так как каждый итератор хранит свою собственную информацию о со-

Итераторы: \_\_iter\_\_ и \_\_next\_\_   
805  
стоянии, каждый цикл управляет своим собственным положением в строке,   
независимо от любых других активных циклов.   
В главах 14 и 20 мы видели похожие примеры. Например, функции-генераторы   
и выражения-генераторы, а также встроенные функции, такие как map и zip,   
возвращают итераторы однократного применения. Напротив, встроенная   
функция range и другие встроенные типы, такие как списки, поддерживают   
возможность создания множества независимых итераторов.  
При создании собственных итераторов мы можем выбирать между поддерж-  
кой единственного итератора или множества независимых итераторов. Чтобы   
обеспечить поддержку множества независимых итераторов, метод \_\_iter\_\_ дол-  
жен не просто возвращать аргумент self, а создавать новый объект итератора со   
своей информацией о состоянии.  
Например, в следующем примере определяется класс итератора, который про-  
пускает каждый второй элемент. Поскольку объект итератора создается заново   
для каждой итерации, он обеспечивает поддержку нескольких активных ци-  
клов одновременно:  
class SkipIterator:  
 def \_\_init\_\_(self, wrapped):  
 self.wrapped = wrapped # Информация о состоянии  
 self.offset = 0  
 def next(self):  
 if self.offset >= len(self.wrapped): # Завершить итерации  
 raise StopIteration  
 else:  
 item = self.wrapped[self.offset] # Иначе перешагнуть и вернуть  
 self.offset += 2  
 return item  
   
class SkipObject:  
 def \_\_init\_\_(self, wrapped): # Сохранить используемый элемент  
 self.wrapped = wrapped   
 def \_\_iter\_\_(self):  
 return SkipIterator(self.wrapped) # Каждый раз новый итератор  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 alpha = ‘abcdef’  
 skipper = SkipObject(alpha) # Создать объект-контейнер  
 I = iter(skipper) # Создать итератор для него  
 print(next(I), next(I), next(I)) # Обойти элементы 0, 2, 4  
   
 for x in skipper: # for вызывает \_\_iter\_\_ автоматически  
 for y in skipper: # Вложенные циклы for также вызывают \_\_iter\_\_   
 print(x + y, end=' ') # Каждый итератор помнит свое состояние, смещение  
Этот пример работает подобно вложенным циклам с обычными строками –   
каждый активный цикл запоминает свое положение в строке, потому что каж-  
дый из них получает независимый объект итератора, который хранит свою   
собственную информацию о состоянии:  
% python skipper.py  
a c e  
aa ac ae ca cc ce ea ec ee

806   
Глава 29. Перегрузка операторов   
Наш более ранний пример класса Squares, напротив, поддерживал всего одну   
активную итерацию, нужно было во вложенных циклах вызывать Squares сно-  
ва, чтобы получить новый объект. Здесь у нас имеется единственный объект   
SkipObject, который создает множество объектов итераторов.  
Как и прежде, подобных результатов можно было бы достичь с использовани-  
ем встроенных инструментов, например с помощью операции получения среза   
с третьим граничным значением, чтобы организовать пропуск элементов:  
>>> S = ‘abcdef’  
>>> for x in S[::2]:  
... for y in S[::2]: # Новые объекты в каждой итерации  
... print(x + y, end=’ ‘)  
...  
aa ac ae ca cc ce ea ec ee  
Однако это далеко не то же самое по двум причинам. Во-первых, каждое выра-  
жение извлечения среза физически сохраняет весь список с результатами в па-  
мяти, тогда как итераторы воспроизводят по одному значению за раз, что по-  
зволяет существенно экономить память в случае большого объема результатов.   
Во-вторых, операции извлечения среза создают новые объекты, поэтому в дей-  
ствительности итерации не протекают одновременно в одном и том же объекте.   
Чтобы оказаться ближе к реализации на основе классов, нам необходимо было   
бы создать единственный объект для обхода, заранее выполнив операцию из-  
влечения среза:  
>>> S = ‘abcdef’  
>>> S = S[::2]  
>>> S  
‘ace’  
>>> for x in S:  
... for y in S: # Тот же самый объект, новые итераторы  
... print(x + y, end=’ ‘)  
...  
aa ac ae ca cc ce ea ec ee  
Эта реализация больше похожа на наше решение, выполненное с помощью   
классов, но здесь по-прежнему список с результатами целиком хранится в па-  
мяти (на сегодняшний день не существует генераторов, способных формиро-  
вать срезы), и эта реализация эквивалентна только для данного конкретного   
случая пропуска каждого второго элемента.  
Итераторы могут выполнять любые действия, которые можно реализовать   
в классах, поэтому они обладают более широкими возможностями, чем пред-  
полагается в данном примере. Независимо от того, требуется ли такая широта   
возможностей в наших приложениях, итераторы, определяемые пользовате-  
лем, представляют собой мощный инструмент – они позволяют создавать про-  
извольные объекты, которые выглядят и ведут себя подобно другим последова-  
тельностям и итерируемым объектам, с которыми мы встречались в этой кни-  
ге. Мы могли бы использовать этот механизм, например, для создания объекта   
базы данных, чтобы одновременно выполнять несколько итераций в одном   
и том же наборе данных, извлеченном в результате запроса к базе данных.

Проверка на вхождение: \_\_contains\_\_, \_\_iter\_\_ и \_\_getitem\_\_   
807  
Проверка на вхождение:   
\_\_contains\_\_, \_\_iter\_\_ и \_\_getitem\_\_  
Область применения итераций значительно шире, чем мы могли видеть до сих   
пор. Перегрузка операторов нередко образует многослойную  архитектуру:   
классы могут предоставлять реализацию специфических методов или обоб-  
щенные альтернативы, используемые в крайнем случае. Например:  
 •  
Операции сравнения в Python 2.6 используют специальные методы, такие   
как \_\_lt\_\_, если они присутствуют, или более обобщенный метод \_\_cmp\_\_.   
В Python 3.0 используются только специализированные методы, а метод   
\_\_cmp\_\_ не используется, как уже объяснялось в этой главе.  
 •  
Операция проверки логического значения также сначала пытается вызвать   
специализированный метод \_\_bool\_\_ (возвращающий явное значение True   
или False), а в случае его отсутствия вызывает более обобщенный метод   
\_\_len\_\_ (ненулевое возвращаемое значение интерпретируется как True). Как   
будет показано ниже в этой главе, интерпретатор версии 2.6 действует точ-  
но так же, но вместо метода \_\_bool\_\_ использует метод \_\_nonzero\_\_.  
Что касается итераций, обычно классы реализуют поддержку оператора in про-  
верки на вхождение с помощью итераций – либо в виде метода \_\_iter\_\_, либо   
\_\_getitem\_\_. Однако классы могут обеспечить более специализированную под-  
держку операции проверки на членство, реализовав метод \_\_contains\_\_, – этот   
метод имеет преимущество перед методом \_\_iter\_\_, который в свою очередь   
пользуется преимуществом перед методом \_\_getitem\_\_. В случае отображений   
метод \_\_contains\_\_ должен определять членство, применяя ключи (и может ис-  
пользовать быструю операцию поиска), а в случае последовательностей – про-  
изводить поиск.  
Рассмотрим следующий пример класса, который реализует все три метода,   
и выполним проверку на вхождение, а также попробуем использовать экзем-  
пляр этого класса в различных итерационных контекстах. Его методы выводят   
сообщения при вызове, позволяющие отслеживать ход выполнения операций:  
class Iters:  
 def \_\_init\_\_(self, value):  
 self.data = value  
 def \_\_getitem\_\_(self, i): # Крайний случай для итераций  
 print(‘get[%s]:’ % i, end=’’) # А также для индексирования и срезов  
 return self.data[i]  
 def \_\_iter\_\_(self): # Предпочтительный для итераций  
 print(‘iter=> ‘, end=’’) # Возможен только 1 активный итератор  
 self.ix = 0  
 return self  
 def \_\_next\_\_(self):  
 print(‘next:’, end=’’)  
 if self.ix == len(self.data): raise StopIteration  
 item = self.data[self.ix]  
 self.ix += 1  
 return item  
 def \_\_contains\_\_(self, x): # Предпочтительный для оператора ‘in’  
 print(‘contains: ‘, end=’’)  
 return x in self.data

808   
Глава 29. Перегрузка операторов   
X = Iters([1, 2, 3, 4, 5]) # Создать экземпляр  
print(3 in X) # Проверка на вхождение  
for i in X: # Циклы  
 print(i, end=’ | ‘)  
   
print()  
print([i \*\* 2 for i in X]) # Другие итерационные контексты  
print( list(map(bin, X)) )  
I = iter(X) # Обход вручную (именно так действуют   
while True: # другие итерационные контексты)  
 try:  
 print(next(I), end=’ @ ‘)  
 except StopIteration:  
 break  
Если запустить этот сценарий, произойдет следующее: специализированный   
метод \_\_contains\_\_ будет вызван операцией проверки на вхождение, обобщен-  
ный метод \_\_iter\_\_ будет вызываться в контексте итераций, в ходе которых   
многократно вызывается метод \_\_next\_\_, а метод \_\_getitem\_\_ не будет вызван   
ни разу:  
contains: True  
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:  
iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]  
iter=> next:next:next:next:next:next:[‘0b1’, ‘0b10’, ‘0b11’, ‘0b100’, ‘0b101’]  
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:  
Но взгляните, что произойдет, если мы закомментируем метод \_\_contains\_\_, –   
теперь операция проверки на вхождение будет использовать обобщенный ме-  
тод \_\_iter\_\_:  
iter=> next:next:next:True  
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:  
iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]  
iter=> next:next:next:next:next:next:[‘0b1’, ‘0b10’, ‘0b11’, ‘0b100’, ‘0b101’]  
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:  
И наконец, ниже приводятся результаты работы сценария, когда оба метода,   
\_\_contains\_\_ и \_\_iter\_\_, были закомментированы, – при проверке на вхождение   
и в других итерационных контекстах используется метод \_\_getitem\_\_, которо-  
му последовательно передаются индексы в порядке возрастания:  
get[0]:get[1]:get[2]:True  
get[0]:1 | get[1]:2 | get[2]:3 | get[3]:4 | get[4]:5 | get[5]:  
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:[1, 4, 9, 16, 25]  
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:[‘0b1’, ‘0b10’, ‘0b11’, ‘0b100’,’0b101’]  
get[0]:1 @ get[1]:2 @ get[2]:3 @ get[3]:4 @ get[4]:5 @ get[5]:  
Как видите, метод \_\_getitem\_\_ является еще более обобщенным: помимо итера-  
ций он также используется операциями индексирования и извлечения срезов.   
При выполнении операции извлечения среза методу \_\_getitem\_\_ передается   
объект, содержащий параметры среза, как в случае встроенных типов, так и в   
случае пользовательских классов, благодаря этому наш класс автоматически   
поддерживает операцию извлечения среза:  
>>> X = Iters(‘spam’) # Индексирование  
>>> X[0] # \_\_getitem\_\_(0)

Обращения к атрибутам: \_\_getattr\_\_ и \_\_setattr\_\_   
809  
get[0]:’s’  
   
>>> ‘spam’[1:] # Извлечение среза  
‘pam’  
>>> ‘spam’[slice(1, None)] # Объект среза  
‘pam’  
   
>>> X[1:] # \_\_getitem\_\_(slice(..))  
get[slice(1, None, None)]:’pam’  
>>> X[:-1]  
get[slice(None, -1, None)]:’spa’  
В более реалистичных случаях использования итераций, когда класс не явля-  
ется последовательностью, реализация метода \_\_iter\_\_ может оказаться еще   
проще, потому что в этом случае нет необходимости управлять целочисленны-  
ми индексами, а метод \_\_contains\_\_ позволяет реализовать более оптимальный   
способ проверки на вхождение.  
Обращения к атрибутам: \_\_getattr\_\_ и \_\_setattr\_\_  
Метод \_\_getattr\_\_ выполняет операцию получения ссылки на атрибут. Если го-  
ворить более определенно, он вызывается с именем атрибута в виде строки вся-  
кий раз, когда обнаруживается попытка получить ссылку на неопределенный   
(несуществующий) атрибут. Этот метод не вызывается, если интерпретатор   
может обнаружить атрибут посредством выполнения процедуры поиска в дере-  
ве наследования. Вследствие этого метод \_\_getattr\_\_ удобно использовать для   
обобщенной обработки запросов к атрибутам. Например:  
>>> class empty:  
... def \_\_getattr\_\_(self, attrname):  
... if attrname == “age”:  
... return 40  
... else:  
... raise AttributeError, attrname  
...  
>>> X = empty()  
>>> X.age  
40  
>>> X.name  
...текст сообщения об ошибке опущен...  
AttributeError: name  
В этом примере класс empty и его экземпляр X не имеют своих собственных   
атрибутов, поэтому при обращении к атрибуту X.age вызывается метод \_\_getat-  
getat-  
tr\_\_ – в аргументе self передается экземпляр (X), а в аргументе attrname – строка   
с именем неопределенного атрибута (“age”). Класс выглядит так, как если бы   
он действительно имел атрибут age, возвращая результат обращения к имени   
X.age (40).В результате получается атрибут, вычисляемый динамически.  
Для атрибутов, обработка которых классом не предусматривается, метод \_\_ge-  
ge-  
tattr\_\_ возбуждает встроенное исключение AttributeError, чтобы сообщить ин-  
терпретатору, что это действительно неопределенные имена, – попытка обра-  
щения к имени X.name приводит к появлению ошибки. Вы еще раз встретитесь   
с методом \_\_getattr\_\_, когда мы будем рассматривать делегирование и свойства

810   
Глава 29. Перегрузка операторов   
в действии в следующих двух главах, а об исключениях я подробно буду рас-  
сказывать в седьмой части книги.  
Родственный ему метод перегрузки \_\_setattr\_\_ перехватывает все попытки при-  
сваивания значений атрибутам. Если этот метод определен, выражение self.  
attr = value будет преобразовано в вызов метода self.\_\_setattr\_(‘attr’, value).   
Работать с этим методом немного сложнее, потому что любая попытка выпол-  
нить присваивание любому атрибуту аргумента self приводит к повторному   
вызову метода \_\_setattr\_\_, вызывая бесконечный цикл рекурсивных вызовов   
(и, в конечном итоге, исключение переполнения стека!). Если вам потребуется   
использовать этот метод, все присваивания в нем придется выполнять посред-  
ством словаря атрибутов, как описывается в следующем разделе. Используйте   
self.\_\_dict\_\_[‘name’] = x, а не self.name = x:  
>>> class accesscontrol:  
... def \_\_setattr\_\_(self, attr, value):  
... if attr == ‘age’:  
... self.\_\_dict\_\_[attr] = value  
... else:  
... raise AttributeError, attr + ‘ not allowed’  
...  
>>> X = accesscontrol()  
>>> X.age = 40 # Вызовет метод \_\_setattr\_\_  
>>> X.age  
40  
>>> X.name = ‘mel’  
...текст сообщения об ошибке опущен...  
AttributeError: name not allowed  
Эти два метода перегрузки операций доступа к атрибутам позволяют контро-  
лировать или специализировать доступ к атрибутам в ваших объектах. Они   
могут играть весьма специфические роли, часть из которых мы рассмотрим   
далее в этой книге.  
Другие способы управления атрибутами  
В будущем вам могут также пригодиться другие способы управления доступом   
к атрибутам, имеющиеся в языке Python:  
 •  
Метод \_\_getattribute\_\_ вызывается при обращениях к любым атрибутам, не   
только к неизвестным� но при реализации этого метода следует быть еще   
более осторожным, чем при реализации метода \_\_getattr\_\_, чтобы избежать   
зацикливания.  
 •  
Встроенная функция property позволяет ассоциировать специальные мето-  
ды с операциями чтения и записи над определенными атрибутами класса.  
 •  
Дескрипторы предоставляют возможность ассоциировать методы \_\_get\_\_   
и \_\_set\_\_ класса с операциями доступа к определенным атрибутам класса.  
Эти дополнительные средства управления атрибутами представляют интерес   
далеко не для всех программистов, использующих язык Python, поэтому мы   
отложим их рассмотрение до обсуждения свойств в главе 31 и детального об-  
суждения всех приемов управления атрибутами в главе 37.

Обращения к атрибутам: \_\_getattr\_\_ и \_\_setattr\_\_   
811  
Имитация частных атрибутов экземпляра: часть 1  
Следующий фрагмент является обобщением предыдущего примера и позволя-  
ет каждому подклассу иметь свой перечень частных имен атрибутов, которым   
нельзя присваивать значения в экземплярах:  
class PrivateExc(Exception): pass # Подробнее об исключениях позднее  
   
class Privacy:  
 def \_\_setattr\_\_(self, attrname, value): # Вызывается self.attrname = value  
 if attrname in self.privates:  
 raise PrivateExc(attrname, self)  
 else:  
 self.\_\_dict\_\_[attrname] = value # Self.attrname = value   
 # вызовет зацикливание!  
class Test1(Privacy):  
 privates = [‘age’]  
   
class Test2(Privacy):  
 privates = [‘name’, ‘pay’]  
 def \_\_init\_\_(self):  
 self.\_\_dict\_\_[‘name’] = ‘Tom’  
   
x = Test1()  
y = Test2()  
   
x.name = ‘Bob’  
y.name = ‘Sue’ # <== ошибка  
   
y.age = 30  
x.age = 40 # <== ошибка  
Фактически это лишь первая прикидочная реализация частных атрибутов   
в языке Python (то есть запрет на изменение атрибутов вне класса). Несмотря   
на то что язык Python не поддерживает возможность объявления частных   
атрибутов, такие приемы, как этот, могут их имитировать. Однако это лишь   
половинчатое решение – чтобы сделать его более эффективным, его необходимо   
дополнить возможностью изменять значения частных атрибутов из подклассов   
и использовать метод \_\_getattr\_\_ и класс-обертку (иногда называется прокси-  
классом), чтобы контролировать получение значений частных атрибутов.  
Рассмотрение полной реализации мы отложим до главы 38, где для выполне-  
ния операций над атрибутами и проверки их значений мы будем использовать   
более универсальный способ, основанный на применении декораторов классов.   
Однако, хотя таким способом можно имитировать сокрытие атрибутов, тем не   
менее он почти никогда не используется на практике. Программисты, исполь-  
зующие язык Python, способны писать крупные объектно-ориентированные   
системы без частных объявлений, но существующие интересные решения по   
управлению доступом выходят далеко за рамки нашего обсуждения.  
Перехват операций обращения к атрибутам и присваивания им значений –   
вообще очень полезный прием. Он обеспечивает возможность делегирова-  
ния – способ, позволяющий обертывать встроенные объекты объектами-  
контроллерами, добавлять новое поведение и делегировать выполнение опе-  
раций обернутым объектам (подробнее о делегировании и классах-обертках   
рассказывается в главе 30).

812   
Глава 29. Перегрузка операторов   
Строковое представление объектов:   
\_\_repr\_\_ и \_\_str\_\_  
В следующем примере реализованы конструктор \_\_init\_\_ и метод перегрузки   
\_\_add\_\_, которые мы уже видели, но в нем также реализован метод \_\_repr\_\_,   
который возвращает строковое представление экземпляров. Здесь этот ме-  
тод используется для преобразования объекта self.data в строку. Если метод   
\_\_repr\_\_ (или родственный ему метод \_\_str\_\_) определен, он автоматически бу-  
дет вызываться при попытках вывести экземпляр класса или преобразовать   
его в строку. Эти методы позволяют определить более удобочитаемый формат   
вывода ваших объектов.  
Строковое представление объектов по умолчанию не содержит полезной ин-  
формации и имеет неудобочитаемый внешний вид:  
>>> class adder:  
... def \_\_init\_\_(self, value=0):  
... self.data = value # Инициализировать атрибут data  
... def \_\_add\_\_(self, other):  
... self.data += other # Прибавить другое значение  
...  
>>> x = adder() # Формат отображения по умолчанию  
>>> print(x)  
<\_\_main\_\_.adder object at 0x025D66B0>  
>>> x  
<\_\_main\_\_.adder object at 0x025D66B0>  
Но возможность реализовать или унаследовать методы преобразования экзем-  
пляров в строковое представление позволяет нам обеспечить вывод дополни-  
тельной информации и предусмотреть ее форматирование:  
>>> class addrepr(adder): # Наследует \_\_init\_\_, \_\_add\_\_  
... def \_\_repr\_\_(self): # Добавляет строковое представление  
... return ‘addrepr(%s)’ % self.data # Преобразует в строку   
... # программного кода  
>>> x = addrepr(2) # Вызовет \_\_init\_\_  
>>> x + 1 # Вызовет \_\_add\_\_  
>>> x # Вызовет \_\_repr\_\_  
addrepr(3)  
>>> print x # Вызовет \_\_repr\_\_  
addrepr(3)  
>>> str(x), repr(x) # Вызовет \_\_repr\_\_  
(‘addrepr(3)’, ‘addrepr(3)’)  
Почему имеется два метода вывода? Дело вот в чем:   
 •  
Встроенные функции print и str (а также ее внутренний эквивалент, кото-  
рый используется функцией print) сначала пытаются использовать метод   
\_\_str\_\_. Вообще этот метод должен возвращать строковое представление   
объекта в удобном для пользователя виде.   
 •  
Во всех остальных случаях используется метод \_\_repr\_\_: функцией авто-  
матического вывода в интерактивной оболочке, функцией repr, при выводе   
вложенных объектов, а также функциями print и str, когда в классе отсут-  
ствует метод \_\_str\_\_. Вообще этот метод должен возвращать строку, которая   
могла бы использоваться как программный код для воссоздания объекта   
или содержать информацию, полезную для разработчиков.

Строковое представление объектов: \_\_repr\_\_ и \_\_str\_\_   
813  
Проще говоря, метод \_\_repr\_\_ используется везде, за исключением функций   
print и str, если определен метод \_\_str\_\_. Однако, если метод \_\_str\_\_ отсутствует,   
операции вывода будут использовать метод \_\_repr\_\_, но не наоборот – в осталь-  
ных случаях, например, функцией автоматического вывода в интерактивной   
оболочке всегда используется только метод \_\_repr\_\_� попытка использовать ме-  
тод \_\_str\_\_ даже не предпринимается:  
>>> class addstr(adder):  
... def \_\_str\_\_(self): # есть \_\_str\_\_, но нет \_\_repr\_\_  
... return ‘[Value: %s]’ % self.data # Преобразовать в красивую строку  
...  
>>> x = addstr(3)  
>>> x + 1  
>>> x # По умолчанию вызывается \_\_repr\_\_  
<\_\_main\_\_.addstr instance at 0x00B35EF0>  
>>> print x # Вызовет \_\_str\_\_  
[Value: 4]  
>>> str(x), repr(x)  
(‘[Value: 4]’, ‘<\_\_main\_\_.addstr instance at 0x00B35EF0>’)  
Вследствие этого, если вам необходимо обеспечить единое отображение во всех   
контекстах, лучше использовать метод \_\_repr\_\_. Однако, определив оба ме-  
тода, вы обеспечите поддержку вывода в различных контекстах. Например,   
перед конечным пользователем объект будет отображаться с помощью метода   
\_\_str\_\_, а перед программистом будет выводиться информация более низкого   
уровня с помощью метода \_\_repr\_\_:  
>>> class addboth(adder):  
... def \_\_str\_\_(self):  
... return ‘[Value: %s]’ % self.data # Удобочитаемая строка   
... def \_\_repr\_\_(self):  
... return ‘addboth(%s)’ % self.data # Строка программного кода  
...  
>>> x = addboth(4)  
>>> x + 1  
>>> x # Вызовет \_\_repr\_\_  
addboth(5)  
>>> print x # Вызовет \_\_str\_\_  
[Value: 5]  
>>> str(x), repr(x)  
(‘[Value: 5]’, ‘addboth(5)’)  
Здесь я должен сделать два примечания, касающиеся использования. Во-  
первых, имейте в виду, что оба метода, \_\_str\_\_ и \_\_repr\_\_, должны возвращать   
строки – возвращаемые значения других типов не преобразуются в строки   
и вызывают ошибку, поэтому не забывайте выполнять преобразование в слу-  
чае необходимости. Во-вторых, в зависимости от логики преобразования   
в строковое представление, реализованной в контейнерном объекте, операция   
вывода может вызывать метод \_\_str\_\_ только для объектов верхнего уровня –   
вложенные объекты по-прежнему могут выводиться с применением их мето-  
дов \_\_repr\_\_ или метода по умолчанию. Оба эти примечания иллюстрируются   
в следующем примере:  
>>> class Printer:  
... def \_\_init\_\_(self, val):  
... self.val = val

814   
Глава 29. Перегрузка операторов   
... def \_\_str\_\_(self): # Используется для вывода самого экземпляра  
... return str(self.val) # Преобразует результат в строку  
...  
>>> objs = [Printer(2), Printer(3)]  
>>> for x in objs: print(x) # При выводе экземпляра будет вызван  
... # \_\_str\_\_, но не тогда, когда экземпляр 2  
2 # находится в списке!  
3   
>>> print(objs)  
[<\_\_main\_\_.Printer object at 0x025D06F0>, <\_\_main\_\_.Printer ...опущено...  
>>> objs  
[<\_\_main\_\_.Printer object at 0x025D06F0>, <\_\_main\_\_.Printer ... опущено...  
Чтобы обеспечить вызов адаптированной версии метода во всех случаях, неза-  
висимо от реализации контейнера, реализуйте метод \_\_repr\_\_, а не \_\_str\_\_ – пер-  
вый из них вызывается во всех случаях, где последний не может быть применен:  
>>> class Printer:  
... def \_\_init\_\_(self, val):  
... self.val = val  
... def \_\_repr\_\_(self): # \_\_repr\_\_ используется print, если нет \_\_str\_\_  
... return str(self.val) # \_\_repr\_\_ используется интерактивной   
... # оболочкой и для вывода вложенных объектов  
>>> objs = [Printer(2), Printer(3)]  
>>> for x in objs: print(x) # Нет \_\_str\_\_: вызовет \_\_repr\_\_  
...  
2  
3  
>>> print(objs) # Вызовет \_\_repr\_\_, а не \_\_\_str\_\_  
[2, 3]  
>>> objs  
[2, 3]  
На практике метод \_\_str\_\_ (и его низкоуровневый родственник \_\_repr\_\_) явля-  
ется вторым по частоте использования после \_\_init\_\_ среди методов перегрузки   
операторов в сценариях на языке Python. Всякий раз, когда вам приходится   
видеть адаптированное отображение при выводе объекта, это значит, что ско-  
рее всего был использован один из этих методов.  
Правостороннее сложение и операция   
приращения: \_\_radd\_\_ и \_\_iadd\_\_  
C технической точки зрения метод \_\_add\_\_, который использовался в примерах   
выше, не поддерживает использование объектов экземпляров справа от опе-  
ратора +. Чтобы реализовать поддержку таких выражений и тем самым обе-  
спечить допустимость перестановки  операндов, необходимо реализовать ме-  
тод \_\_radd\_\_. Интерпретатор вызывает метод \_\_radd\_\_, только когда экземпляр   
вашего класса появляется справа от оператора +, а объект слева не является   
экземпляром вашего класса. Во всех других случаях, когда объект появляется   
слева, вызывается метод \_\_add\_\_:  
>>> class Commuter:  
... def \_\_init\_\_(self, val):  
... self.val = val  
... def \_\_add\_\_(self, other):

Правостороннее сложение и операция приращения: \_\_radd\_\_ и \_\_iadd\_\_   
815  
... print(‘add’, self.val, other)  
... return self.val + other  
... def \_\_radd\_\_(self, other):  
... print(‘radd’, self.val, other)  
... return other + self.val  
...  
>>> x = Commuter(88)  
>>> y = Commuter(99)  
>>> x + 1 # \_\_add\_\_: экземпляр + не\_экземпляр  
add 88 1  
89  
>>> 1 + y # \_\_radd\_\_: не\_экземпляр + экземпляр  
radd 99 1  
100  
>>> x + y # \_\_add\_\_: экземпляр + экземпляр  
add 88 <\_\_main\_\_.Commuter instance at 0x02630910>  
radd 99 88  
187  
Обратите внимание на изменение порядка следования операндов в вызове ме-  
тода \_\_radd\_\_: аргумент self в действительности находится справа от оператора   
+, а аргумент other – слева. Кроме того, следует заметить, что здесь x и y – это эк-  
земпляры одного и того же класса, – когда в выражении участвуют экземпля-  
ры разных классов, интерпретатор предпочитает вызывать метод экземпляра,   
расположенного слева. Когда выполняется операция сложения двух экземпля-  
ров, интерпретатор вызывает метод \_\_add\_\_, который в свою очередь вызывает   
метод \_\_radd\_\_, упрощая левый операнд.  
На практике, когда требуется распространить тип класса на результат, реали-  
зация может оказаться сложнее: может оказаться необходимым выполнить   
проверку типа, чтобы убедиться в безопасности операции преобразовании   
и избежать вложенности. Так, если в следующем примере не выполнять про-  
верку типа с помощью функции isinstance, дело может закончиться тем, что   
мы получим экземпляр класса Commuter, значением атрибута val которого яв-  
ляется другой экземпляр класса Commuter, – при сложении двух экземпляров,   
когда метод \_\_add\_\_ вызывает метод \_\_radd\_\_:  
>>> class Commuter: # Тип класса распространяется на результат  
... def \_\_init\_\_(self, val):  
... self.val = val  
... def \_\_add\_\_(self, other):  
... if isinstance(other, Commuter): other = other.val  
... return Commuter(self.val + other)  
... def \_\_radd\_\_(self, other):  
... return Commuter(other + self.val)  
... def \_\_str\_\_(self):  
... return ‘<Commuter: %s>’ % self.val  
...  
>>> x = Commuter(88)  
>>> y = Commuter(99)  
>>> print(x + 10) # Результат – другой экземпляр класса Commuter  
<Commuter: 98>  
>>> print(10 + y)  
<Commuter: 109>  
   
>>> z = x + y # Нет вложения: не происходит рекурсивный вызов \_\_radd\_\_   
>>> print(z)

816   
Глава 29. Перегрузка операторов   
<Commuter: 187>  
>>> print(z + 10)  
<Commuter: 197>  
>>> print(z + z)  
<Commuter: 374>  
Комбинированная операция сложения  
Чтобы обеспечить поддержку комбинированной операции сложения +=, увели-  
чивающей значение экземпляра, необходимо реализовать метод \_\_iadd\_\_ или   
\_\_add\_\_. Последний из них используется в случае отсутствия первого. Фактиче-  
 из них используется в случае отсутствия первого. Фактиче-  
из них используется в случае отсутствия первого. Фактиче-  
 них используется в случае отсутствия первого. Фактиче-  
них используется в случае отсутствия первого. Фактиче-  
 используется в случае отсутствия первого. Фактиче-  
используется в случае отсутствия первого. Фактиче-  
 в случае отсутствия первого. Фактиче-  
в случае отсутствия первого. Фактиче-  
 случае отсутствия первого. Фактиче-  
случае отсутствия первого. Фактиче-  
 отсутствия первого. Фактиче-  
отсутствия первого. Фактиче-  
 первого. Фактиче-  
первого. Фактиче-  
. Фактиче-  
Фактиче-  
ски класс Commuter, представленный в предыдущем разделе, уже поддерживает   
операцию +=, однако с помощью метода \_\_iadd\_\_ можно реализовать более эф-  
фективную операцию изменения самого экземпляра:  
>>> class Number:  
... def \_\_init\_\_(self, val):  
... self.val = val  
... def \_\_iadd\_\_(self, other): # \_\_iadd\_\_ явно реализует операцию x += y  
... self.val += other # Обычно возвращает self  
... return self  
...  
>>> x = Number(5)  
>>> x += 1  
>>> x += 1  
>>> x.val  
7  
>>> class Number:  
... def \_\_init\_\_(self, val):  
... self.val = val  
... def \_\_add\_\_(self, other): # \_\_add\_\_ - как крайнее средство: x=(x + y)  
... return Number(self.val + other) # Распространяет тип класса  
...  
>>> x = Number(5)  
>>> x += 1  
>>> x += 1  
>>> x.val  
7  
Любой двухместный оператор имеет похожий правосторонний метод пере-  
грузки и метод, реализующий комбинированную операцию присваивания (на-  
пример, \_\_mul\_\_, \_\_rmul\_\_ и \_\_imul\_\_). Правосторонние методы – это достаточно   
сложная тема, и на практике они используются очень редко – к ним требуется   
обращаться только в том случае, когда необходимо обеспечить для оператора   
возможность перестановки операндов, и если вообще необходима реализа-  
ция поддержки этого оператора. Например, эти методы могут использоваться   
в классе Vector, но в таких классах, как Employee или Button, скорее всего, они   
не нужны.  
Операция вызова: \_\_call\_\_  
Метод \_\_call\_\_ вызывается при обращении к экземпляру как к функции. Это   
не повторяющееся определение – если метод \_\_call\_\_ присутствует, интерпре-

Операция вызова: \_\_call\_\_   
817  
татор будет вызывать его, когда экземпляр вызывается как функция, переда-  
вая ему любые позиционные и именованные аргументы:  
>>> class Callee:  
... def \_\_call\_\_(self, \*pargs, \*\*kargs): # Реализует вызов экземпляра  
... print(‘Called:’, pargs, kargs) # Принимает любые аргументы  
...  
>>> C = Callee()  
>>> C(1, 2, 3) # C – вызываемый объект  
Called: (1, 2, 3) {}  
>>> C(1, 2, 3, x=4, y=5)  
Called: (1, 2, 3) {‘y’: 5, ‘x’: 4}  
Выражаясь более формальным языком, метод \_\_call\_\_ поддерживает все схе-  
мы передачи аргументов, обсуждавшиеся в главе 18, – все, что передается эк-  
земпляру, передается этому методу наряду с обычным аргументом self, в ко-  
тором передается сам экземпляр. Например, следующие определения метода:  
class C:  
 def \_\_call\_\_(self, a, b, c=5, d=6): ... # Обычные и со значениями   
 # по умолчанию  
class C:  
 def \_\_call\_\_(self, \*pargs, \*\*kargs): ... # Произвольные аргументы  
   
class C:  
 def \_\_call\_\_(self, \*pargs, d=6, \*\*kargs): ... # Аргументы, которые могут   
 # передаваться только по   
 # имени в версии 3.0  
соответствуют следующим вызовам экземпляра:  
X = C()  
X(1, 2) # Аргументы со значениями по умолчанию опущены  
X(1, 2, 3, 4) # Позиционные  
X(a=1, b=2, d=4) # Именованные  
X(\*[1, 2], \*\*dict(c=3, d=4)) # Распаковывание произвольных аргументов  
X(1, \*(2,), c=3, \*\*dict(d=4)) # Смешанные режимы  
Суть состоит в том, что классы и экземпляры, имеющие метод \_\_call\_\_, под-  
держивают тот же синтаксис и семантику передачи аргументов, что и обычные   
функции и методы.  
Реализация операции вызова, как в данном примере, позволяет экземплярам   
классов имитировать поведение функций, а также сохранять информацию   
о состоянии между вызовами (похожий пример мы видели в главе 17, когда ис-  
следовали области видимости, но теперь вы больше знаете о перегрузке опера-  
торов):  
>>> class Prod:  
... def \_\_init\_\_(self, value): # Принимает единственный аргумент  
... self.value = value  
... def \_\_call\_\_(self, other):  
... return self.value \* other  
...  
>>> x = Prod(2) # “Запоминает” 2 в своей области видимости  
>>> x(3) # 3 (передано) \* 2 (сохраненное значение)  
6  
>>> x(4)  
8

818   
Глава 29. Перегрузка операторов   
В этом примере реализация метода \_\_call\_\_ может показаться ненужной. То   
же самое поведение можно реализовать с помощью простого метода:  
>>> class Prod:  
... def \_\_init\_\_(self, value):  
... self.value = value  
... def comp(self, other):  
... return self.value \* other  
...  
>>> x = Prod(3)  
>>> x.comp(3)  
9  
>>> x.comp(4)  
12  
Однако метод \_\_call\_\_ может оказаться удобнее при взаимодействии с при-  
кладными интерфейсами, где ожидается функция, – это позволяет создавать   
объекты, совместимые с интерфейсами, ожидающими получить функцию, ко-  
торые к тому же способны сохранять информацию о своем состоянии между   
вызовами. Фактически этот метод занимает третье место среди наиболее часто   
используемых методов перегрузки операторов – после конструктора \_\_init\_\_   
и методов форматирования \_\_str\_\_ и \_\_repr\_\_.  
Функциональные интерфейсы   
и программный код обратного вызова  
Инструментальный набор для создания графического интерфейса tkinter   
(Tkinter, в Python 2.6) позволяет регистрировать функции как обработчики со-  
бытий (они же – функции обратного вызова)� когда возникают какие-либо со-  
бытия, tkinter вызывает зарегистрированные объекты. Если вам необходимо   
реализовать обработчик событий, способный сохранять свое состояние между   
вызовами, вы можете либо зарегистрировать связанный метод класса, либо эк-  
земпляр класса, который с помощью метода \_\_call\_\_ обеспечивает совмести-  
мость с функциональным интерфейсом. В программном коде этого раздела оба   
варианта – x.comp из второго примера и экземпляр x из первого – могут переда-  
ваться в виде объектов функций.  
В следующей главе я более подробно расскажу о связанных методах, а пока   
разберем гипотетический пример использования метода \_\_call\_\_ для построе-  
ния графического интерфейса. Следующий класс определяет объект, поддер-  
живающий функциональный интерфейс, и, кроме того, имеет информацию   
о состоянии, сохраняя цвет, в который должна окрашиваться нажатая кнопка:  
class Callback:  
 def \_\_init\_\_(self, color): # Функция + информация о состоянии  
 self.color = color  
 def \_\_call\_\_(self): # Поддерживает вызовы без аргументов  
 print(‘turn’, self.color)  
Теперь мы можем зарегистрировать экземпляры этого класса в контексте гра-  
фического интерфейса, как обработчики событий для кнопок, несмотря на то,   
что реализация графического интерфейса предполагает вызывать обработчи-  
ки событий как обычные функции без аргументов:

Функциональные интерфейсы и программный код обратного вызова   
819  
cb1 = Callback(‘blue’) # ‘Запомнить’ голубой цвет  
cb2 = Callback(‘green’)  
   
B1 = Button(command=cb1) # Зарегистрировать обработчик  
B2 = Button(command=cb2) # Зарегистрировать обработчик  
Когда позднее кнопка будет нажата, объект экземпляра будет вызван как про-  
стая функция, точно как в следующих ниже вызовах. А поскольку он сохраня-  
ет информацию о состоянии в атрибутах экземпляра, он помнит, что необходи-  
мо сделать:  
cb1() # По событию: выведет ‘blue’  
cb2() # Выведет ‘green’  
Фактически это один из лучших способов сохранения информации о состоя-  
нии в языке Python – он намного лучше способов, обсуждавшихся ранее и при-  
Python – он намного лучше способов, обсуждавшихся ранее и при-  
 – он намного лучше способов, обсуждавшихся ранее и при-  
менявшихся к функциям (глобальные переменные, ссылки в область видимо-  
сти объемлющей функции и изменяемые аргументы со значениями по умолча-  
нию). Благодаря ООП состояние можно сохранять явно, посредством присваи-  
вания значений атрибутам.  
Прежде чем двинуться дальше, рассмотрим еще два способа, которые исполь-  
зуются программистами для сохранения информации о состоянии в функциях   
обратного вызова. В первом варианте используется lambda-функция с аргумен-  
тами, имеющими значения по умолчанию:  
cb3 = (lambda color=’red’: ‘turn ‘ + color) # Или: по умолчанию  
print(cb3())  
Во втором – используются связанные методы класса. Объект связанного мето-  
да – это объект, который запоминает экземпляр self и ссылку на функцию, так   
что потом можно вызывать простую функцию без использования экземпляра:  
class Callback:  
 def \_\_init\_\_(self, color): # Класс с информацией о состоянии  
 self.color = color  
 def changeColor(self): # Обычный метод  
 print(‘turn’, self.color)  
   
cb1 = Callback(‘blue’)  
cb2 = Callback(‘yellow’)  
   
B1 = Button(command=cb1.changeColor) # Ссылка, не вызов  
B2 = Button(command=cb2.changeColor) # Запоминаются функция+self  
Когда позднее кнопка будет нажата, имитируется поведение графического   
интерфейса и вызывается метод changeColor, который обработает информацию   
о состоянии объекта:  
object = Callback(‘blue’)  
cb = object.changeColor # Регистрация обработчика событий  
cb() # По событию выведет ‘blue’  
Этот прием является более простым, но менее универсальным, чем перегрузка   
операции вызова с помощью метода \_\_call\_\_. Еще раз напомню, что подробнее   
о связанных методах будет рассказываться в следующей главе.

820   
Глава 29. Перегрузка операторов   
Кроме того, в главе 31 будет представлен еще один пример использования ме-  
тода \_\_call\_\_, который будет использоваться для реализации так называемого   
декоратора функции – вызываемого объекта, добавляющего уровень логики   
поверх самой функции. Поскольку метод \_\_call\_\_ позволяет присоединять   
информацию о состоянии к вызываемым объектам, этот прием является есте-  
ственным для реализации функций, которые должны запоминать и вызывать   
другие функции.  
Сравнение: \_\_lt\_\_, \_\_gt\_\_ и другие  
Как следует из табл. 29.1, классы могут определять методы, реализующие все   
шесть операций сравнения: <, >, <=, >=, == и !=. Обычно эти методы достаточно   
просты в реализации, но имейте в виду следующее:  
 •  
В отличие от методов \_\_add\_\_/\_\_radd\_\_, обсуждавшихся выше, методы срав-  
нения не имеют правосторонних версий. Вместо этого, когда операцию   
сравнения поддерживает только один операнд, используются зеркальные   
методы сравнивания (например, методы \_\_lt\_\_ и \_\_gt\_\_ являются зеркаль-  
ными по отношению друг к другу).  
 •  
Среди операторов сравнения нет неявных взаимоотношений. Суть в том,   
что истинность операции == не предполагает ложность операции !=, напри-  
мер, чтобы гарантировать корректное поведение обоих операторов, требует-  
ся реализовать оба метода, \_\_eq\_\_ и \_\_ne\_\_.  
 •  
В Python 2.6 все операции сравнения можно было реализовать в виде одного   
метода \_\_cmp\_\_ – он должен выполнить сравнивание (self с другим операн-  
дом) и вернуть число меньшее, равное или большее нуля, чтобы показать,   
что аргумент self меньше, равен или больше второго аргумента соответ-  
ственно. На практике для получения результата в этом методе часто исполь-  
зуется встроенная функция cmp(x, y). В Python 3.0 метод \_\_cmp\_\_ и встро-  
енная функция cmp были удалены: вместо них следует использовать более   
специализированные методы.  
Из-за экономии места в книге мы не имеем возможности провести полное об-  
суждение всех особенностей методов сравнения, однако, в качестве краткого   
введения, исследуем следующий класс и программный код его проверки:  
class C:  
 data = ‘spam’  
 def \_\_gt\_\_(self, other): # версии 3.0 и 2.6  
 return self.data > other  
 def \_\_lt\_\_(self, other):  
 return self.data < other  
   
X = C()  
print(X > ‘ham’) # Выведет True (вызовет \_\_gt\_\_)  
print(X < ‘ham’) # Выведет False (вызовет \_\_lt\_\_)  
Этот сценарий выведет одинаковые результаты в Python 3.0 и 2.6, инструкции   
print в конце сценария выведут вполне ожидаемые результаты, как отмечено   
в комментариях, потому что класс реализует специализированные методы   
сравнения.

Проверка логического значения: \_\_bool\_\_ и \_\_len\_\_   
821  
Метод \_\_cmp\_\_ в Python 2.6 (удален в 3.0)   
В Python 2.6 имеется возможность реализовать метод \_\_cmp\_\_, который будет   
использоваться в крайнем случае, когда отсутствует более специализирован-  
ный метод сравнения: его целочисленное возвращаемое значение используется   
для вычисления результата, возвращаемого оператором. Следующий сценарий   
выведет те же результаты при выполнении под управлением Python 2.6, но по-  
Python 2.6, но по-  
 2.6, но по-  
терпит неудачу при выполнении под управлением Python 3.0, так как в этой   
версии метод \_\_cmp\_\_ больше не используется:  
class C:  
 data = ‘spam’ # Действует только в версии 2.6  
 def \_\_cmp\_\_(self, other): # \_\_cmp\_\_ не используется в версии 3.0  
 return cmp(self.data, other) # Функция cmp отсутствует в версии 3.0  
   
X = C()  
print(X > ‘ham’) # Выведет True (вызовет \_\_cmp\_\_)  
print(X < ‘ham’) # Выведет False (вызовет \_\_cmp\_\_)  
Обратите внимание, что неудача при запуске сценария под управлением Py-  
Py-  
thon 3.0 обусловлена вовсе не тем, что в этой версии отсутствует функция cmp,   
а тем, что метод \_\_cmp\_\_ в этой версии больше не считается специальным. Если   
изменить предыдущий класс так, чтобы он не вызывал функцию cmp, сценарий   
по-прежнему будет работать в версии 2.6, но не будет работать в версии 3.0:  
class C:  
 data = ‘spam’  
 def \_\_cmp\_\_(self, other):  
 return (self.data > other) - (self.data < other)  
Вы могли бы спросить, зачем я привел пример использования   
метода сравнения, который больше не поддерживается в вер-  
сии 3.0? Было бы проще полностью забыть все, что было раньше,   
однако эта книга охватывает обе версии Python, 2.6 и 3.0. Так   
как вы все еще можете встретить метод \_\_cmp\_\_ в программах,   
написанных для работы под управлением версии 2.6, которые   
вам может потребоваться сопровождать, его описание вполне   
оправданно. Кроме того, удаление метода \_\_cmp\_\_ оказалось еще   
большей неожиданностью, чем удаление метода \_\_getslice\_\_,   
описанного выше, и потому этот метод еще достаточно долго мо-  
жет находиться в употреблении. Однако, если вы используете   
Python 3.0 или для вас важно обеспечить совместимость с этой   
версией, не используйте метод \_\_cmp\_\_: используйте специализи-  
рованные методы сравнения.  
Проверка логического значения:   
\_\_bool\_\_ и \_\_len\_\_  
Как уже упоминалось выше, классы могут также определять методы, выра-  
жающие логическую природу их экземпляров, – в логическом контексте ин-

822   
Глава 29. Перегрузка операторов   
терпретатор сначала пытается напрямую получить логическое значение с по-  
мощью метода \_\_bool\_\_ и только потом, если этот метод не реализован, пыта-  
ется вызвать метод \_\_len\_\_ , чтобы выяснить истинность объекта, исходя из   
его длины. Обычно первый из них возвращает логическое значение, исходя из   
значений атрибутов объекта или другой информации:  
>>> class Truth:  
... def \_\_bool\_\_(self): return True  
...  
>>> X = Truth()  
>>> if X: print(‘yes!’)  
...  
yes!  
>>> class Truth:  
... def \_\_bool\_\_(self): return False  
...  
>>> X = Truth()  
>>> bool(X)  
False  
Если этот метод отсутствует, интерпретатор пытается определить длину объек-  
та, поскольку непустой объект интерпретируется как истинный (то есть, если   
длина не равна нулю, в логическом контексте такому объекту соответствует   
значение True, в противном случае – False):  
>>> class Truth:  
... def \_\_len\_\_(self): return 0  
...  
>>> X = Truth()  
>>> if not X: print(‘no!’)  
...  
no!  
Если в классе реализованы оба метода, предпочтение отдается методу \_\_bool\_\_,   
потому что он является более специализированным:  
>>> class Truth:  
... def \_\_bool\_\_(self): return True # в 3.0 первым опробуется \_\_bool\_\_  
... def \_\_len\_\_(self): return 0 # в 2.6 первым опробуется \_\_len\_\_  
...  
>>> X = Truth()  
>>> if X: print(‘yes!’)  
...  
yes!  
Если ни один из методов не определен, объект просто считается истинным (в   
чем вполне можно усмотреть смысл, особенно если вы склонны к метафизиче-  
ским рассуждениям!):  
>>> class Truth:  
... pass  
...  
>>> X = Truth()  
>>> bool(X)  
True

Проверка логического значения: \_\_bool\_\_ и \_\_len\_\_   
823  
Теперь, после того, как нам пришлось соприкоснуться с областью философии,   
обратим свой взгляд на последний метод перегрузки операторов, который реа-  
лизует уничтожение объектов.  
Логические значения в Python 2.6  
Во всех примерах в разделе «Проверка логического значения: \_\_bool\_\_   
и \_\_len\_\_» пользователи Python 2.6 должны использовать метод \_\_non-  
zero\_\_ вместо \_\_bool\_\_. В Python 3.0 метод \_\_nonzero\_\_ был переименован   
в \_\_bool\_\_, но во всех остальных отношениях проверка значения объекта   
в логическом контексте производится одинаково (в обеих версиях, 3.0   
и 2.6, метод \_\_len\_\_ используется в крайнем случае).  
Если запустить самый первый пример в этом разделе под управлени-  
ем Python 2.6, не изменяя имени метода, он все равно будет работать   
правильно, но только потому, что интерпретатор не распознает метод   
\_\_bool\_\_ как специальный метод и объект по умолчанию будет рассма-  
триваться, как истинный!  
Чтобы сделать различия между версиями более явственными, метод   
следует переопределить так, чтобы он возвращал False:  
C:\misc> c:\python30\python  
>>> class C:  
... def \_\_bool\_\_(self):  
... print(‘in bool’)  
... return False  
...  
>>> X = C()  
>>> bool(X)  
in bool  
False  
>>> if X: print(99)  
...  
in bool  
В версии 3.0 были получены вполне предсказуемые результаты. Однако   
в версии 2.6 метод \_\_bool\_\_ игнорируется и объект всегда интерпретиру-  
ется как истинный:  
C:\misc> c:\python26\python  
>>> class C:  
... def \_\_bool\_\_(self):  
... print(‘in bool’)  
... return False  
...  
>>> X = C()  
>>> bool(X)  
True  
>>> if X: print(99)  
...  
99

824   
Глава 29. Перегрузка операторов   
Чтобы реализовать представление объекта в логическом контексте,   
в версии 2.6 следует определить метод \_\_nonzero\_\_ (или возвращать 0 из   
метода \_\_len\_\_, чтобы объект рассматривался как ложный):  
C:\misc> c:\python26\python  
>>> class C:  
... def \_\_nonzero\_\_(self):  
... print(‘in nonzero’)  
... return False  
...  
>>> X = C()  
>>> bool(X)  
in nonzero  
False  
>>> if X: print(99)  
...  
in nonzero  
Но имейте в виду, что метод \_\_nonzero\_\_ может использоваться только   
в версии 2.6, – в версии 3.0 он просто будет игнорироваться, и объект по   
умолчанию будет классифицироваться как истинный – точно так же,   
как при использовании метода \_\_bool\_\_ в версии 2.6!  
Уничтожение объектов: \_\_del\_\_  
Конструктор \_\_init\_\_ вызывается во время создания экземпляра. Противопо-  
ложный ему метод \_\_del\_\_ вызывается автоматически, когда освобождается   
память, занятая объектом (то есть во время «сборки мусора»):  
>>> class Life:  
... def \_\_init\_\_(self, name=’unknown’):  
... print ‘Hello’, name  
... self.name = name  
... def \_\_del\_\_(self):  
... print(‘Goodbye’, self.name)  
...  
>>> brian = Life(‘Brian’)  
Hello Brian  
>>> brian = ‘loretta’  
Goodbye Brian  
Здесь, когда переменной brian присваивается строка, теряется последняя   
ссылка на экземпляр класса Life, что приводит к вызову деструктора. Этот   
метод удобно использовать для реализации некоторых заключительных дей-  
ствий (таких как завершение соединения с сервером). Однако в языке Python   
по целому ряду причин деструкторы используются не так часто, как в других   
объектно-ориентированных языках программирования.

В заключение   
825  
С одной стороны, интерпретатор автоматически освобождает память, занятую   
экземпляром, поэтому нет нужды выполнять очистку памяти в деструкторах.1   
С другой стороны, не всегда бывает возможным предсказать, когда произойдет   
уничтожение экземпляра, поэтому часто лучше выполнять завершающие дей-  
ствия в явно вызываемом методе (или в инструкции try/finally, которая описы-  
вается в следующей части книги) – в некоторых случаях в системных таблицах   
могут сохраняться ссылки на ваши объекты, что будет препятствовать вызову   
деструктора.  
Фактически использование метода \_\_del\_\_ может осложняться   
еще целым рядом причин. Например, исключения, возникшие   
внутри этого метода, просто выводят сообщения в поток sys.  
stderr (поток стандартного вывода сообщений об ошибках), а не   
вызывают событие исключения, что вызвано непредсказуемо-  
стью контекста, в котором метод запускается сборщиком мусора.   
Кроме того, перекрестные (они же – циклические) ссылки между   
объектами могут препятствовать сборке мусора, когда вы ожи-  
даете ее, – механизм определения циклических ссылок, который   
по умолчанию включен, способен автоматически собирать такие   
объекты, но только если они не имеют методов \_\_del\_\_. Посколь-  
ку в этом вопросе возникает слишком неясностей, мы не будем   
дальше погружаться в детали – полный охват особенностей мето-  
да \_\_del\_\_ и модуля сборки мусора gc (garbage collector) вы найде-  
те в стандартном руководстве по языку Python.  
В заключение  
Мы поместили в этой главе столько примеров перегрузки операторов, сколько   
позволило пространство, отведенное для главы. Большая часть других методов   
перегрузки работают похожим образом, и все они – всего лишь ловушки для   
перехвата встроенных операций. Некоторые методы перегрузки, например,   
имеют уникальные списки аргументов или возвращаемые значения. С некото-  
рыми из них мы еще встретимся далее в этой книге:  
 •  
В главе 33 будут использоваться методы \_\_enter\_\_ и \_\_exit\_\_ менеджеров   
контекста, к которым может применяться инструкция with.  
 •  
В главе 37 будут использоваться методы \_\_get\_\_ и \_\_set\_\_ дескрипторов для   
методов чтения/записи значений атрибутов класса.  
 •  
В главе 39 будет использоваться метод \_\_new\_\_, который вызывается на эта-  
пе создания объекта, в контексте метаклассов.  
Кроме того, мы еще не раз встретимся с некоторыми изученными здесь метода-  
ми, такими как \_\_call\_\_ и \_\_str\_\_, в последующих примерах в этой книге. Тем   
1   
В текущей реализации Python на языке C, кроме всего прочего, нет необходимости   
закрывать файлы в деструкторах, потому что они автоматически закрываются при   
уничтожении объектов файлов. Однако, как упоминалось в главе 9, лучше все-таки   
явно закрывать файлы, потому что «автоматическое закрытие при уничтожении   
объекта» – это особенность реализации, а не самого языка (в Jython это поведение   
может отличаться).

826   
Глава 29. Перегрузка операторов   
не менее полный охват этой темы я оставляю за другими источниками инфор-  
мации – за дополнительными подробностями о методах перегрузки операторов   
обращайтесь к стандартному руководству по языку Python или к печатным   
справочным изданиям.  
В следующей главе мы покинем область исследования механики классов и зай-  
мемся исследованием некоторых распространенных шаблонов проектирова-  
ния – способам использования и комбинирования классов для оптимизации   
многократного использования программного кода. Однако, прежде чем про-  
должить чтение, ответьте на обычные контрольные вопросы, чтобы освежить   
в памяти все, о чем говорилось в этой главе.  
Закрепление пройденного  
Контрольные вопросы  
1. Какие два метода перегрузки операторов можно использовать для поддерж-  
ки итераций в классах?  
2. Какие два метода перегрузки операторов можно использовать для вывода   
и в каких случаях?  
3. Как реализовать в классе операции над срезами?  
4. Как реализовать в классе операцию приращения значения самого объекта?  
5. Когда следует использовать методы перегрузки операторов?  
Ответы  
1. Классы могут обеспечить поддержку итераций, определив (или унаследо-  
вав) метод \_\_getitem\_\_ или \_\_iter\_\_. Во всех итерационных контекстах ин-  
терпретатор Python сначала пытается использовать метод \_\_iter\_\_ (кото-  
рый возвращает объект, поддерживающий итерационный протокол в виде   
метода \_\_next\_\_): если метод \_\_iter\_\_ не будет найден в результате поиска   
по дереву наследования, интерпретатор возвращается к использованию ме-  
тода извлечения элемента по его индексу \_\_getitem\_\_ (который вызывается   
многократно и при каждом вызове получает постоянно увеличивающиеся   
значения индексов).  
2. Вывод объекта реализуют методы \_\_str\_\_ и \_\_repr\_\_. Первый из них вызы-  
вается встроенными функциями print и str� последний также вызывается   
функциями print и str, если в классе отсутствует метод \_\_str\_\_, и всегда –   
встроенной функцией repr, функцией автоматического вывода интерактив-  
ной оболочки и при выводе вложенных экземпляров. То есть метод \_\_repr\_\_   
используется везде, за исключением функций print и str, если определен   
метод \_\_str\_\_. Метод \_\_str\_\_ обычно используется для вывода объектов   
в удобочитаемом представлении� метод \_\_repr\_\_ выводит дополнительные   
подробности об объекте или представляет объект в виде программного кода.  
3. Операцию извлечения среза можно перехватить с помощью метода \_\_get-  
item\_\_: в этом случае ему передается не простой числовой индекс, а объект   
среза. В Python 2.6 точно так же можно использовать метод \_\_getslice\_\_   
(в версии 3.0 он уже не используется).

Закрепление пройденного   
827  
4. Операция приращения пытается сначала вызвать метод \_\_iadd\_\_, а затем   
метод \_\_add\_\_ с последующим присваиванием. Тот же самый прием может   
применяться для всех двухместных операторов. Кроме того, правосторон-  
нее сложение можно реализовать с помощью метода \_\_radd\_\_.  
5. Когда класс естественным образом соответствует встроенным типам или   
должен подражать их поведению. Например, классы коллекций могут ими-  
тировать поведение последовательностей или отображений. Как правило,   
не следует реализовать методы перегрузки операторов, если они не явля-  
ются естественными для ваших объектов, – лучше использовать методы   
с обычными именами.

Глава 30.  
   
Шаблоны проектирования с классами  
До сих пор в этой части книги мы все свое внимание уделяли использованию   
объектно-ориентированных инструментов языка Python – классов. Но ООП –   
это еще и задача проектирования: как использовать классы для моделирова-  
ния полезных объектов. В этой главе мы коснемся некоторых базовых идей   
ООП и рассмотрим несколько дополнительных примеров, более реалистичных,   
чем преведенные до сих пор.   
Попутно мы реализуем некоторые наиболее распространенные шаблоны про-  
ектирования ООП в языке Python, такие как наследование, композиция, деле-  
Python, такие как наследование, композиция, деле-  
, такие как наследование, композиция, деле-  
гирование и фабрики. Кроме того, мы исследуем некоторые концепции, связан-  
ные с проектированием, такие как псевдочастные атрибуты, множественное   
наследование и связанные методы. Многие термины, упоминающиеся здесь,   
требуют более подробного пояснения, чем я приводил ранее в этой книге. Если   
эта тема вызывает у вас интерес, я предлагаю в качестве следующего шага   
взяться за изучение книг, посвященных шаблонам проектирования в ООП.   
Python и ООП  
Начнем с краткого обзора. Реализацию ООП в языке Python можно свести   
к трем следующим идеям:  
Наследование  
Наследование основано на механизме поиска атрибутов в языке Python   
(в выражении X.name).  
Полиморфизм  
Назначение метода method в выражении X.method зависит от типа (класса) X.  
Инкапсуляция  
Методы и операторы реализуют поведение� сокрытие данных – это соглаше-  
ние по умолчанию.  
К настоящему времени вы уже должны иметь представление о том, что такое   
наследование в языке Python. Кроме того, мы уже несколько раз говорили о по-  
Python. Кроме того, мы уже несколько раз говорили о по-  
. Кроме того, мы уже несколько раз говорили о по-  
лиморфизме в языке Python – он произрастает из отсутствия объявления типов

Python и ООП   
829  
в языке Python. Поскольку разрешение имен атрибутов производится на этапе   
выполнения, объекты, реализующие одинаковые интерфейсы, являются вза-  
имозаменяемыми – клиентам не требуется знать тип объекта, реализующего   
вызываемый метод.  
Инкапсуляция в языке Python означает упаковывание – то есть сокрытие под-  
Python означает упаковывание – то есть сокрытие под-  
 означает упаковывание – то есть сокрытие под-  
робностей реализации за интерфейсом объекта. Это не означает принудитель-  
ное сокрытие, но оно может быть реализовано, о чем будет рассказано в гла-  
ве 38. Инкапсуляция позволяет изменять реализацию интерфейсов объекта,   
не оказывая влияния на пользователей этого объекта.  
Перегрузка посредством сигнатур вызова   
(точнее, ее невозможность)   
В некоторых объектно-ориентированных языках под полиморфизмом также   
понимается возможность перегрузки функций, основанной на сигнатурах ти-  
пов их аргументов. Но так как в языке Python отсутствуют объявления типов,   
эта концепция в действительности здесь неприменима – полиморфизм в языке   
Python основан на интерфейсах объектов, а не на типах.  
Вы можете попробовать выполнить перегрузку методов, изменяя списки их ар-  
гументов, как показано ниже:  
class C:  
 def meth(self, x):  
 ...  
 def meth(self, x, y, z):  
 ...  
Это вполне работоспособный программный код, но так как инструкция def   
просто присваивает объект некоторому имени в области видимости класса, со-  
хранено будет только последнее определение метода (это все равно, что запи-  
сать две инструкции подряд: X = 1, а затем X = 2, в результате чего X будет иметь   
значение 2).  
Выбор на основе типа всегда можно реализовать с помощью идеи проверки   
типа, с которой мы встречались в главах 4 и 9, или с помощью возможности   
передачи списка аргументов, обсуждавшейся в главе 18:  
class C:  
 def meth(self, \*args):  
 if len(args) == 1:  
 ...  
 elif type(arg[0]) == int:  
 ...  
Однако обычно этого следует избегать, потому что, как описывалось в главе 16,   
следует писать такой код, который опирается на интерфейс объекта, а не на   
конкретный тип данных. Такой подход полезнее, так как охватывает более   
широкие категории типов и приложений, как нынешних, так и тех, что по-  
явятся в будущем:  
class C:  
 def meth(self, x):  
 x.operation() # Предполагается, что x работает правильно

830   
Глава 30. Шаблоны проектирования с классами   
Кроме того, считается, что лучше выбирать разные имена для методов, выпол-  
няющих разные операции, и не полагаться на сигнатуры вызова (при этом не-  
важно, какой язык программирования вы используете).  
Объектная модель в языке Python достаточно проста, поэтому основное мастер-  
Python достаточно проста, поэтому основное мастер-  
 достаточно проста, поэтому основное мастер-  
ство владения ООП заключается в умении комбинировать классы в программе   
для достижения поставленных целей. В следующем разделе мы начинаем экс-  
курс по некоторым приемам использования классов в крупных программах.  
ООП и наследование:   
взаимосвязи типа «является»  
Мы уже достаточно подробно исследовали механизм наследования, но мне хо-  
телось бы показать пример того, как может использоваться модель отношений   
реального мира. С точки зрения программиста, наследование вступает в игру   
с момента появления квалифицированного имени атрибута, при разрешении   
которого запускается поиск имен в экземплярах, в их классах и затем в супер-  
классах. С точки зрения проектировщика, наследование – это способ указать   
принадлежность к некоторому набору: класс определяет набор свойств, кото-  
рые могут быть унаследованы и адаптированы более специализированными   
наборами (то есть подклассами).  
Чтобы проиллюстрировать сказанное, давайте вернемся к машине по изготов-  
лению пиццы, о которой мы говорили в начале этой части книги. Предполо-  
жим, что мы исследовали альтернативные варианты развития своей карьеры   
и решили открыть пиццерию. Первое, что нам предстоит сделать, это нанять   
работников для обслуживания клиентов, для приготовления блюд и так да-  
лее. Будучи в глубине души инженерами, мы решили сконструировать робота   
по приготовлению пиццы, но, будучи также политически и кибернетически   
корректными, мы решили сделать нашего робота полноправным служащим,   
которому выплачивается заработная плата.  
Наш коллектив работников пиццерии можно определить четырьмя классами   
из файла примера employees.py. Самый общий класс, Employee, реализует по-  
ведение, общее для всех работников, такое как повышение заработной платы   
(giveRaise) и вывод на экран (\_\_repr\_\_). Существует две категории служащих   
и соответственно, два подкласса, наследующих класс Employee: Chef (повар)   
и Server (официант). Оба подкласса переопределяют унаследованный метод   
work, чтобы обеспечить вывод более специализированных сообщений. Наконец,   
наш робот по приготовлению пиццы моделируется еще более специализиро-  
ванным классом PizzaRobot, наследующим класс Chef, который в свою очередь   
наследует класс Employee. В терминах ООП мы называем такие взаимоотноше-  
ния «является»: робот является поваром, а повар является служащим. Ниже   
приводится содержимое файла employees.py:  
class Employee:  
 def \_\_init\_\_(self, name, salary=0):  
 self.name = name  
 self.salary = salary  
 def giveRaise(self, percent):  
 self.salary = self.salary + (self.salary \* percent)  
 def work(self):  
 print self.name, “does stuff”

ООП и наследование: взаимосвязи типа «является»   
831  
 def \_\_repr\_\_(self):  
 return “<Employee: name=%s, salary=%s>” % (self.name, self.salary)  
   
class Chef(Employee):  
 def \_\_init\_\_(self, name):  
 Employee.\_\_init\_\_(self, name, 50000)  
 def work(self):  
 print(self.name, “makes food”)  
   
class Server(Employee):  
 def \_\_init\_\_(self, name):  
 Employee.\_\_init\_\_(self, name, 40000)  
 def work(self):  
 print(self.name, “interfaces with customer”)  
   
class PizzaRobot(Chef):  
 def \_\_init\_\_(self, name):  
 Chef.\_\_init\_\_(self, name)  
 def work(self):  
 print(self.name, “makes pizza”)  
   
if \_\_name\_\_ == “\_\_main\_\_”:  
 bob = PizzaRobot(‘bob’) # Создать робота с именем bob  
 print(bob) # Вызвать унаследованный метод \_\_repr\_\_  
 bob.work() # Выполнить действие, зависящее от типа  
 bob.giveRaise(0.20) # Увеличить роботу зарплату на 20%  
 print(bob); print()  
   
 for klass in Employee, Chef, Server, PizzaRobot:  
 obj = klass(klass.\_\_name\_\_)  
 obj.work()  
Когда выполняется программный код самопроверки, включенный в состав мо-  
дуля, создается робот по приготовлению пиццы с именем bob, который насле-  
дует атрибуты трех классов: PizzaRobot, Chef и Employee. Например, при попытке   
вывести экземпляр bob вызывается метод Employee.\_\_repr\_\_, а прибавка зарпла-  
ты производится методом Employee.giveRaise, потому что этот метод обнаружи-  
вается в процессе поиска в дереве наследования именно в этом классе:  
C:\python\examples> python employees.py  
<Employee: name=bob, salary=50000>  
bob makes pizza  
<Employee: name=bob, salary=60000.0>  
   
Employee does stuff  
Chef makes food  
Server interfaces with customer  
PizzaRobot makes pizza  
В иерархиях классов, подобных этой, обычно можно создавать экземпляры лю-  
бого класса, а не только того, что находится в самом низу. Например, в коде   
самопроверки этого модуля в цикле for создаются экземпляры всех четырех   
классов, каждый из которых работает по-разному, потому что все они имеют   
различные методы work. В действительности эти классы пока лишь имитируют   
объекты реального мира – в текущей реализации метод work просто выводит   
сообщение, но позднее его можно расширить так, что он будет выполнять на-  
стоящую работу.

832   
Глава 30. Шаблоны проектирования с классами   
ООП и композиция: взаимосвязи типа «имеет»  
Понятие композиции в этой книге было введено в главе 25. С точки зрения про-  
граммиста, композиция – это прием встраивания других объектов в объект-  
контейнер и использование их для реализации методов контейнера. Для про-  
ектировщика композиция – это один из способов представить взаимоотноше-  
ния в прикладной области. Но вместо того, чтобы определять принадлежность   
к множеству, при композиционном подходе все части объединяются в единое   
целое.  
Кроме того, композиция отражает взаимоотношения между частями, которые   
обычно называются отношениями типа «имеет». В некоторых книгах, посвя-  
щенных объектно-ориентированному проектированию, композиция называ-  
ется агрегированием (и различие между терминами состоит в том, что термин   
«агрегирование» используется для описания более слабой зависимости между   
контейнером и его содержимым)� в этой книге термин «композиция» исполь-  
зуется лишь для обозначения коллекции встраиваемых объектов. Вообще со-  
ставные классы реализуют все свои интерфейсы, управляя работой встраивае-  
мых объектов.  
Теперь, когда у нас имеются реализации классов работников, объединим их   
в коллектив пиццерии и позволим им приступить к работе. Наша пиццерия –   
это составной объект: в нем имеется печь и работники, такие как официанты   
и повара. Когда приходит клиент и делает заказ, все компоненты пиццерии на-  
чинают действовать – официант принимает заказ, повар готовит пиццу и так   
далее. Следующий пример (файл pizzashop.py) имитирует все объекты и взаи-  
моотношения между ними:  
from employees import PizzaRobot, Server  
   
class Customer:  
 def \_\_init\_\_(self, name):  
 self.name = name  
 def order(self, server):  
 print(self.name, “orders from”, server)  
 def pay(self, server):  
 print(self.name, “pays for item to”, server)  
   
class Oven:  
 def bake(self):  
 print(“oven bakes”)  
   
class PizzaShop:  
 def \_\_init\_\_(self):  
 self.server = Server(‘Pat’) # Встроить другие объекты  
 self.chef = PizzaRobot(‘Bob’) # Робот по имени Bob  
 self.oven = Oven()  
   
 def order(self, name):  
 customer = Customer(name) # Активизировать другие объекты  
 customer.order(self.server) # Клиент делает заказ официанту  
 self.chef.work()  
 self.oven.bake()  
 customer.pay(self.server)  
   
if \_\_name\_\_ == “\_\_main\_\_”:  
 scene = PizzaShop() # Создать составной объект

ООП и композиция: взаимосвязи типа «имеет»   
833  
 scene.order(‘Homer’) # Имитировать заказ клиента Homer  
 print(‘...’)  
 scene.order(‘Shaggy’) # Имитировать заказ клиента Shaggy  
Класс PizzaShop – это контейнер и контроллер – это конструктор, который соз-  
дает и встраивает экземпляры классов работников, написанные нами в преды-  
дущем разделе, а также экземпляры класса Oven, который определен здесь.   
Когда программный код самопроверки этого модуля вызывает метод order   
класса PizzaShop, встроенным объектам предлагается приступить к выполне-  
нию своих обязанностей. Обратите внимание, что для каждого клиента мы   
создаем новый экземпляр класса Customer и передаем встроенный объект Server   
(официант) методам класса Customer (клиент) – клиенты приходят и уходят,   
а официант остается частью коллектива пиццерии. Кроме того, обратите вни-  
мание, что работники по-прежнему вовлечены во взаимосвязи наследования –   
композиция и наследование – это взаимодополняющие инструменты.   
Если запустить этот модуль, наша пиццерия обслужит два заказа – один от   
Гомера (Homer) и другой от Шагги (Shaggy):  
C:\python\examples> python pizzashop.py  
Homer orders from <Employee: name=Pat, salary=40000>  
Bob makes pizza  
oven bakes  
Homer pays for item to <Employee: name=Pat, salary=40000>  
...  
Shaggy orders from <Employee: name=Pat, salary=40000>  
Bob makes pizza  
oven bakes  
Shaggy pays for item to <Employee: name=Pat, salary=40000>  
Это всего лишь игрушечная имитация, но объекты и взаимодействия между   
ними наглядно демонстрируют составные объекты в действии. Классы могут   
представлять практически любые объекты и взаимоотношения между ними,   
которые можно выразить словами� для этого просто замените имена существи-  
тельные классами, глаголы – методами, и вы получите первый черновой на-  
бросок проекта.  
Еще раз об обработке потоков  
Рассмотрим более реалистичный пример использования приема композиции.   
Вспомните универсальную функцию обработки потоков данных, которая ча-  
стично была реализована во введении в ООП в главе 25:  
def processor(reader, converter, writer):  
 while 1:  
 data = reader.read()  
 if not data: break  
 data = converter(data)  
 writer.write(data)  
Однако вместо простой функции мы могли бы реализовать обработку в виде   
класса, который использует прием композиции, чтобы обеспечить поддержку   
наследования и более удобную конструкцию программного кода. Одна из воз-  
можных реализаций этого класса содержится в файле streams.py и приводится   
в ниже:

834   
Глава 30. Шаблоны проектирования с классами   
class Processor:  
 def \_\_init\_\_(self, reader, writer):  
 self.reader = reader  
 self.writer = writer  
 def process(self):  
 while 1:  
 data = self.reader.readline()  
 if not data: break  
 data = self.converter(data)  
 self.writer.write(data)  
 def converter(self, data):  
 assert False, ‘converter must be defined’ # Или возбудить исключение  
Этот класс определяет метод converter, который, как ожидается, будет пере-  
определен в подклассах. Это пример использования абстрактных суперклас-  
сов, с которыми мы познакомились в главе 28 (подробнее об инструкции assert   
рассказывается в седьмой части книги). При таком подходе объекты чтения   
(reader) и записи (writer) встраиваются в экземпляр класса (композиция), а ло-  
гика преобразования поставляется в виде подкласса (наследование), а не в виде   
отдельной функции. Ниже приводится содержимое файла converters.py:  
from streams import Processor  
   
class Uppercase(Processor):  
 def converter(self, data):  
 return data.upper()  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 import sys  
 obj = Uppercase(open(‘spam.txt’), sys.stdout)  
 obj.process()  
Здесь класс Uppercase наследует логику цикла обработки потока данных (и все   
остальное, что может присутствовать в суперклассах). В нем необходимо опре-  
делить лишь то, что будет уникальным для него, – логику преобразования дан-  
ных. Если запустить этот файл, он создаст и запустит экземпляр класса Upper-  
case, который прочитает содержимое файла spam.txt, преобразует все символы   
в верхний регистр и выведет их в поток stdout:  
C:\lp4e> type spam.txt  
spam  
Spam  
SPAM!  
   
C:\lp4e> python converters.py  
SPAM  
SPAM  
SPAM!  
Для обработки потоков различных видов достаточно передать конструктору   
класса объекты требуемых типов. Ниже приводится пример реализации выво-  
да в файл вместо стандартного потока вывода:  
C:\lp4e> python  
>>> import converters  
>>> prog = converters.Uppercase(open(‘spam.txt’), open(‘spamup.txt’, ‘w’))  
>>> prog.process()

ООП и композиция: взаимосвязи типа «имеет»   
835  
C:\lp4e> type spamup.txt  
SPAM  
SPAM  
SPAM!  
Но, как предлагалось ранее, мы могли бы также реализовать объекты, оберну-  
тые в классы, которые определяют необходимые интерфейсные методы ввода   
и вывода. Ниже приводится простой пример, где вывод осуществляется через   
класс, который обертывает выводимый текст в теги HTML:  
C:\lp4e> python  
>>> from converters import Uppercase  
>>>  
>>> class HTMLize:  
... def write(self, line):  
... print ‘<PRE>%s</PRE>’ % line.rstrip())  
...  
>>> Uppercase(open(‘spam.txt’), HTMLize()).process()  
<PRE>SPAM</PRE>  
<PRE>SPAM</PRE>  
<PRE>SPAM!</PRE>  
Если проследить порядок выполнения этого примера, можно заметить, что   
было получено два варианта преобразований – приведение символов к верхне-  
му регистру (наследованием) и преобразование в формат HTML (композицией),   
хотя основная логика обработки в оригинальном суперклассе Processor ничего   
не знает ни об одном из них. Программному коду, выполняющему обработку,   
нужны только метод write – в классах, выполняющих запись, и метод convert.   
Его совершенно не интересует, что делают эти методы. Такой полиморфизм   
и инкапсуляция логики составляют основу такой мощи классов.  
В этом примере суперкласс Processor реализует только цикл сканирования   
файла. Для выполнения более существенных действий его можно было бы рас-  
ширить, чтобы обеспечить поддержку дополнительных инструментов в его   
подклассах и постепенно превратить все это в полноценный фреймворк. Соз-  
дав такой инструмент один раз, вы сможете многократно использовать его во   
всех своих программах. Даже в этом простом примере благодаря тому, что с по-  
мощью классов можно упаковать и унаследовать так много, все, что нам при-  
шлось сделать, – это реализовать этап преобразования в формат HTML, а все   
остальное у нас уже и так имелось.  
Еще один пример композиции в действии приводится в упражнении 9 в конце   
главы 31, а его решение – в приложении B. Он напоминает пример с пиццери-  
ей. В этой книге мы сосредоточились на наследовании, потому что это основной   
инструмент, который обеспечивает объектно-ориентированные возможности   
в языке Python. Однако на практике прием композиции используется ничуть   
не реже, чем наследование, в качестве способа организации классов, особенно   
в крупных системах. Как мы видели, наследование и композиция – часто взаи-  
модополняющие (а иногда и альтернативные) приемы. Композиция – это про-  
блема проектирования, которая далеко выходит за рамки языка Python и этой   
книги, поэтому полный охват этой темы я оставляю за другими источниками   
информации.

836   
Глава 30. Шаблоны проектирования с классами   
Придется держать в уме: классы и их хранение  
В этой части книги я уже несколько раз упоминал о возможности со-  
хранения объектов с помощью модулей pickle и shelve, потому что этот   
метод особенно хорошо работает с экземплярами классов. Эти инстру-  
менты достаточно привлекательны, чтобы послужить мотвацией к ис-  
пользованию классов, – возможность сохранения экземпляров классов   
позволяет организовать хранилища, содержащие в себе данные и логи-  
ку ее обработки.  
Например, помимо возможности имитировать взаимодействия в реаль-  
ном мире, классы, разработанные для пиццерии, могли бы также ис-  
пользоваться как основа базы данных пиццерии. Экземпляры классов   
могут сохраняться на диск за одно действие – с помощью модулей pickle   
или shelve. В главе 27 мы уже использовали модуль shelve для сохра-  
нения экземпляров классов, однако интерфейс модуля pickle ничуть не   
сложнее в использовании:  
import pickle  
object = someClass()  
file = open(filename, ‘wb’) # Создать внешний файл  
pickle.dump(object, file) # Сохранить объект в файле  
   
import pickle  
file = open(filename, ‘rb’)  
object = pickle.load(file) # Позднее извлечь обратно  
Модуль pickle преобразует объекты, находящиеся в памяти, в последо-  
вательности байтов (в действительности – в строки), которые можно   
сохранять в файлах, передавать по сети и так далее. При извлечении   
объектов происходит обратное преобразование: из последовательности   
байтов в идентичные объекты в памяти. Модуль shelve реализует похо-  
жую возможность, но он автоматически сохраняет объекты в базе дан-  
ных с доступом по ключу, которая предоставляет интерфейс, похожий   
на интерфейс словаря:  
import shelve  
object = someClass()  
dbase = shelve.open(‘filename’)  
dbase[‘key’] = object # Сохранить под ключом key  
   
import shelve  
dbase = shelve.open(‘filename’)  
object = dbase[‘key’] # Позднее извлечь обратно  
В нашем примере использование классов для моделирования работни-  
ков означает, что можно достаточно легко создать простую базу данных   
сотрудников и пиццерий: записывая экземпляры объектов в файл, мы   
сможем сохранять их между запусками программы:  
>>> from pizzashop import PizzaShop  
>>> shop = PizzaShop()  
>>> shop.server, shop.chef

ООП и делегирование: объекты-обертки   
837  
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)  
>>> import pickle  
>>> pickle.dump(shop, open(‘shopfile.dat’, ‘wb’))  
Мы можем сохранить в файле весь составной объект, представляющий   
пиццерию, одной инструкцией. Чтобы восстановить его в следующем   
сеансе или при очередном запуске программы, также достаточно един-  
ственной инструкции. При этом после восстановления таким способом   
объекты получают обратно и свои данные, и свою логику работы:  
>>> import pickle  
>>> obj = pickle.load(open(‘shopfile.dat’, ‘rb’))  
>>> obj.server, obj.chef  
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)  
>>> obj.order(‘Sue’)  
Sue orders from <Employee: name=Pat, salary=40000>  
Bob makes pizza  
oven bakes  
Sue pays for item to <Employee: name=Pat, salary=40000>  
Более подробную информацию о сохранении ищите в руководстве к стан-  
дартной библиотеке.  
ООП и делегирование: объекты-обертки  
Кроме терминов «наследование» и «композиция» в ООП также часто исполь-  
зуется термин делегирование, под которым обычно подразумевается наличие   
объекта-контроллера, куда встраиваются другие объекты, получающие за-  
просы на выполнение операций. Контроллеры могут решать административ-  
ные задачи, такие как слежение за попытками доступа и так далее. В языке   
Python делегирование часто реализуется с помощью метода \_\_getattr\_\_, потому   
что он перехватывает попытки доступа к несуществующим атрибутам. Класс-  
обертка (иногда называется прокси-классом) может использовать метод \_\_ge-  
ge-  
tattr\_\_ для перенаправления обращений к обернутому объекту. Класс-обертка   
повторяет интерфейс обернутого объекта и может добавлять дополнительные   
операции.  
В качестве примера рассмотрим файл trace.py:  
class wrapper:  
 def \_\_init\_\_(self, object):  
 self.wrapped = object # Сохранить объект  
 def \_\_getattr\_\_(self, attrname):  
 print(‘Trace:’, attrname) # Отметить факт извлечения  
 return getattr(self.wrapped, attrname) # Делегировать извлечение  
В главе 29 говорилось, что метод \_\_getattr\_\_ получает имя атрибута в виде стро-  
ки. В этом примере для извлечения из обернутого объекта атрибута, имя кото-  
рого представлено в виде строки, используется встроенная функция getattr –   
вызов getattr(X, N) аналогичен выражению X.N за исключением того, что N – это   
выражение, которое во время выполнения представлено строкой, а не именем   
переменной. Фактически вызов getattr(X, N) по его действию можно сравнить

838   
Глава 30. Шаблоны проектирования с классами   
с выражением X.\_\_dict\_\_[N], только в первом случае дополнительно выполняет-  
ся поиск в дереве наследования, как в выражении X.N, а во втором – нет (подроб-  
нее об атрибуте \_\_dict\_\_ рассказывается в разделе «Словари пространств имен»   
в главе 29).  
Такой прием, реализованный в этом классе-обертке, можно использовать для   
управления доступом к любому объекту с атрибутами – спискам, словарям   
и даже к классам и экземплярам. Ниже приводится класс wrapper, который   
просто выводит сообщение при каждом обращении к атрибуту и делегирует   
этот запрос обернутому объекту wrapped:  
>>> from trace import wrapper  
>>> x = wrapper([1,2,3]) # Обернуть список  
>>> x.append(4) # Делегировать операцию методу списка  
Trace: append  
>>> x.wrapped # Вывести обернутый объект  
[1, 2, 3, 4]  
   
>>> x = wrapper({“a”: 1, “b”: 2}) # Обернуть словарь  
>>> x.keys() # Делегировать операцию методу словаря  
Trace: keys  
[‘a’, ‘b’]  
В результате интерфейс обернутого объекта расширяется за счет методов   
класса-обертки. Этот подход может использоваться для регистрации вызовов   
методов, перенаправления вызовов методов дополнительному или адаптиро-  
ванному программному коду и так далее.  
В главе 31 мы еще вернемся к обернутым объектам и делегированию операций,   
как к одному из способов расширения встроенных типов. Если шаблон проек-  
тирования с делегированием заинтересовал вас, тогда смотрите обсуждение   
декораторов функций в главах 31 и 38, очень близкой концепции расширения   
отдельных функций и методов, а не всего интерфейса объекта, и декораторов   
классов, которые обеспечивают возможность автоматического добавления обер-  
ток, реализующих прием делегирования, ко всем экземплярам классов.  
Примечание,  касающееся  различий  между  версиями: В версии   
Python 2.6 методы перегрузки операторов, которые вызываются   
встроенными операциями, выбираются с помощью методов, пе-  
рехватывающих обращения к атрибутам, таких как \_\_getattr\_\_.   
Операция непосредственного вывода обернутого объекта, на-  
пример, вызывает этот метод, чтобы выбрать \_\_repr\_\_ или \_\_  
str\_\_, и затем производит вызов соответствующего метода обер-  
нутого объекта. В Python 3.0 эта схема больше не действует: опе-  
рация вывода не обращается к методу \_\_getattr\_\_ и использует   
процедуру вывода по умолчанию. В версии 3.0, где все классы   
относятся к классам нового стиля, интерпретатор пытается оты-  
скать метод перегрузки оператора в классах, пропуская этап по-  
иска в экземпляре. Мы еще вернемся к этой проблеме в главе 37,   
когда будем изучать вопросы управления атрибутами, а пока   
имейте в виду, что вам может потребоваться переопределить ме-  
тоды перегрузки операторов в классах-обертках (вручную, с по-  
мощью других инструментов или с помощью суперклассов),   
если вы хотите, чтобы они действовали в Python 3.0.

Псевдочастные атрибуты класса   
839  
Псевдочастные атрибуты класса  
Помимо основной задачи, связанной со структурированием программного   
кода, в классах часто приходится решать проблемы использования имен. В пя-  
той части книги мы узнали, что все имена, для которых выполняется присваи-  
вание на верхнем уровне модуля, становятся глобальными для этого модуля.   
То же по умолчанию относится и к классам – сокрытие данных регулируется   
соглашениями, и клиенты могут получать и изменять любые атрибуты класса   
или экземпляра по своему усмотрению. Фактически все атрибуты являются   
«общедоступными» (public) и «виртуальными» (virtual), если говорить в тер-  
минах языка C++, – они доступны отовсюду и динамически отыскиваются во   
время выполнения.1  
Тем не менее Python поддерживает такое понятие, как «искажение» («man-  
man-  
gling») имен (то есть расширение) с целью придать им черты локальных имен   
для класса. Искаженные имена иногда ошибочно называют «частными атри-  
бутами», но в действительности это всего лишь способ ограничить доступ   
к именам в классе – искажение имен не предотвращает доступ из программно-  
го кода, находящегося за пределами класса. Эта особенность в основном пред-  
назначена, чтобы избежать конфликтов имен в экземплярах, а не для ограни-  
чения доступа к именам, – поэтому искаженные имена лучше называть «псев-  
дочастными», чем «частными».  
Псевдочастные имена – это дополнительная и совершенно не обязательная   
возможность, и вы, скорее всего, не сочтете ее полезной, пока не столкнетесь   
с необходимостью создания инструментов общего назначения и многоуровне-  
вых иерархий классов в проектах, создаваемых командами программистов.   
Фактически псевдочастные имена не всегда используются даже тогда, когда   
их следовало бы использовать. Гораздо чаще программисты дают внутренним   
атрибутам имена, начинающиеся с одного символа подчеркивания (напри-  
мер, \_X), – согласно неофициальным соглашениям, атрибуты с такими имена-  
ми не должны изменяться за пределами класса (для самого интерпретатора   
такие имена не имеют специального значения).  
Так как использование этой возможности может встретиться в программном   
коде других программистов, вам следует знать о ней, даже если вы сами ее не   
используете.  
Об искажении имен в общих чертах  
Здесь описывается, как действует механизм искажения имен: имена внутри   
инструкции class, которые начинаются с двух символов подчеркивания, но не   
заканчиваются двумя символами подчеркивания, автоматически расширяют-  
1   
Это обычно пугает программистов, работающих с языком C++. В языке Python воз-  
можно даже изменить или полностью удалить метод класса во время выполнения   
программы. С другой стороны, почти никто не использует такие возможности на   
практике. Как язык сценариев, Python больше печется о том, чтобы позволить, а не   
запретить. Кроме того, вспомните обсуждение перегрузки операторов в главе 29, где   
говорилось, что методы \_\_getattr\_\_ и \_\_setattr\_\_ могут использоваться для имитации   
поведения частных атрибутов, но на практике эта возможность обычно не использу-  
ется. Подробнее об этом мы поговорим в главе 38, когда будем создавать действую-  
щий декоратор сокрытия атрибутов.

840   
Глава 30. Шаблоны проектирования с классами   
ся за счет включения имени вмещающего класса. Например, такое имя, как   
\_\_X, в классе с именем Spam автоматически изменится на \_Spam\_\_X: к оригиналь-  
ному имени будет добавлен префикс, состоящий из символа подчеркивания   
и имени вмещающего класса, и в результате будет получено достаточно уни-  
кальное имя, которое не будет вступать в конфликт с именами в других клас-  
сах иерархии.  
Искажение имен происходит только внутри инструкций class и только для   
имен, которые начинаются двумя символами подчеркивания. Однако это про-  
исходит со всеми именами, которые начинаются двумя символами подчерки-  
вания, включая имена методов и имена атрибутов экземпляров (например,   
в нашем примере с классом Spam ссылка на атрибут экземпляра self.\_\_X будет   
преобразована в self.\_Spam\_\_X). Поскольку экземпляр может получать атрибу-  
ты более чем из одного класса, такое искажение позволяет избежать конфлик-  
тов, но чтобы понять, как это происходит, нам нужно рассмотреть пример.  
Для чего нужны псевдочастные атрибуты?   
Задача, которую призваны решить псевдочастные атрибуты, состоит в том,   
чтобы обеспечить способ сохранности атрибутов экземпляра. В языке Python   
все атрибуты экземпляра принадлежат единственному объекту экземпляра,   
расположенному внизу дерева наследования. Это существенно отличается от   
модели языка C++, где каждый класс обладает своим собственным набором   
членов данных, которые он определяет.  
В языке Python всякий раз, когда в пределах метода класса выполняется при-  
Python всякий раз, когда в пределах метода класса выполняется при-  
 всякий раз, когда в пределах метода класса выполняется при-  
сваивание атрибуту аргумента self (например, self.attr = value), создается или   
изменяется атрибут экземпляра (поиск в дереве наследования выполняется   
только при попытке получить ссылку, а не присвоить значение). Это верно,   
даже когда несколько классов в иерархии выполняют присваивание одному   
и тому же атрибуту, поэтому конфликты имен вполне возможны.  
Например, предположим, что, когда программист писал класс, он предпола-  
гал, что экземпляры этого класса будут владеть атрибутом X. В методах класса   
выполняется присваивание этому атрибуту и позднее извлекается его значе-  
ние:  
class C1:  
 def meth1(self): self.X = 88 # Предполагается, что X - это мой атрибут  
 def meth2(self): print(self.X)  
Далее предположим, что другой программист, работающий отдельно, исходил   
из того же предположения, когда писал свой класс:  
class C2:  
 def metha(self): self.X = 99 # И мой тоже  
 def methb(self): print(self.X)  
Каждый класс по отдельности работает нормально. Проблема возникает, когда   
оба класса оказываются в одном дереве наследования:  
class C3(C1, C2): ...  
I = C3() # У меня только один атрибут X!  
Теперь значение, которое получит каждый класс из выражения self.X, будет   
зависеть от того, кто из них последним присвоил значение. Все операции при-  
сваивания атрибуту self.X будут воздействовать на один и тот же экземпляр,

Псевдочастные атрибуты класса   
841  
у которого может быть только один атрибут X – I.X, – независимо от того, сколь-  
ко классов используют это имя.  
Чтобы гарантировать принадлежность атрибута тому классу, который его ис-  
пользует, достаточно в начале имени атрибута поставить два символа подчерки-  
вания везде, где оно используется классом, как в следующем файле private.py:  
class C1:  
 def meth1(self): self.\_\_X = 88 # Теперь X - мой атрибут  
 def meth2(self): print(self.\_\_X) # Превратится в \_C1\_\_X  
   
class C2:  
 def metha(self): self.\_\_X = 99 # И мой тоже  
 def methb(self): print(self.\_\_X) # Превратится в \_C2\_\_X  
   
class C3(C1, C2): pass  
I = C3() # В I два имени X  
   
I.meth1(); I.metha()  
print(I.\_\_dict\_\_)  
I.meth2(); I.methb()  
При наличии такой приставки имена атрибутов X будут дополнены именами   
их классов, прежде чем будут добавлены в экземпляр. Если вызвать функцию   
dir, чтобы просмотреть перечень атрибутов экземпляра I, или просмотреть   
содержимое его словаря пространства имен после того, как атрибутам будут   
присвоены значения, вы увидите измененные имена \_C1\_\_X и \_C2\_\_X, но не X.   
Такое дополнение придаст именам уникальность внутри экземпляра, поэтому   
разработчики классов могут рассчитывать на то, что все имена, начинающиеся   
с двух символов подчеркивания, действительно принадлежат их классам:  
% python private.py  
{‘\_C2\_\_X’: 99, ‘\_C1\_\_X’: 88}  
88  
99  
Этот прием помогает избежать конфликтов имен в экземплярах, но заметьте,   
что он не обеспечивает настоящего сокрытия данных. Если вы знаете имя вме-  
щающего класса, вы сможете обратиться к их атрибутам из любой точки про-  
граммы, где имеется ссылка на экземпляр, используя для этого расширенное   
имя (например, I.\_C1\_\_X = 77). С другой стороны, эта особенность делает менее   
вероятными случайные конфликты с существующими именами в классе.  
Псевдочастные атрибуты также удобно использовать в крупных проектах, так   
как они позволяют избежать необходимости выдумывать новые имена мето-  
дов, которые по ошибке могут переопределить методы, уже существующие   
выше в дереве классов, и помогают снизить вероятность, что внутренние ме-  
тоды окажутся переопределенными где-то ниже в дереве классов. Если метод   
предназначен для использования только внутри класса, и этот класс может   
наследовать или наследоваться другими классами, приставка из двух симво-  
лов подчеркивания гарантирует, что имя этого метода не будет конфликтовать   
с другими именами в дереве, особенно, когда используется прием множествен-  
ного наследования:  
class Super:  
 def method(self): ... # Фактический прикладной метод  
   
class Tool:

842   
Глава 30. Шаблоны проектирования с классами   
 def \_\_method(self): ... # Получит имя \_Tool\_\_method  
 def other(self): self.\_\_method() # Используется внутренний метод  
   
class Sub1(Tool, Super): ...  
 def actions(self): self.method() # Вызовет метод Super.method  
   
class Sub2(Tool):  
 def \_\_init\_\_(self): self.method = 99 # Не уничтожит метод Tool.\_\_method  
Мы коротко познакомились с механизмом множественного наследования   
в главе 25 и более подробно будем исследовать его ниже, в этой главе. Напом-  
ню, что в случае множественного наследования поиск атрибутов в этих клас-  
сах производится слева направо, в порядке их следования в заголовке инструк-  
ции class. Для данного примера это означает, что при обращении к атрибутам   
в классе Sub1 поиск унаследованных атрибутов сначала будет производиться   
в классе Tool, и только потом в классе Super. Мы могли бы в этом примере выну-  
дить интерпретатор сначала пытаться выбирать методы класса Super, поменяв   
порядок следования суперклассов в заголовке определения класса Sub1, но это   
никак не повлияло бы на порядок разрешения имен псевдочастных атрибутов.   
Кроме того, псевдочастные имена предотвращают возможность переопределе-  
ния внутренних методов в подклассах, как показано в классе Sub2.  
Еще раз отмечу, что эта особенность более полезна для крупных проектов,   
в которых участвует несколько программистов, и только для отдельных имен.   
Не торопитесь загромождать свой программный код лишними символами без   
нужды – используйте эту особенность, только когда действительно необходи-  
мо обеспечить принадлежность атрибута единственному классу. Для простых   
программ этот прием будет излишеством.  
Дополнительные примеры использования имен вида \_\_X вы найдете в файле   
lister.py, в примесных классах, которые будут представлены ниже в этой главе,   
в разделе, посвященном множественному наследованию, а также в главе 38,   
в обсуждении декоратора классов Private. Если проблема частных атрибутов   
представляет для вас интерес, вернитесь к главе 29, где в разделе «Обращения   
к атрибутам: \_\_getattr\_\_ и \_\_setattr\_\_» коротко описывается прием имитации   
частных атрибутов, и посмотрите на реализацию декоратора классов Private,   
основанную на этом приеме, которая приводится в главе 38. В действительно-  
сти в языке Python имеется возможность по-настоящему управлять доступом   
к атрибутам классов, однако она редко используется на практике, даже в круп-  
ных системах.  
Методы – это объекты:   
связанные и несвязанные методы  
Методы вообще и связанные методы в частности, упрощают решение многих   
задач в языке Python. Мы уже сталкивались со связанными методами в гла-  
Python. Мы уже сталкивались со связанными методами в гла-  
. Мы уже сталкивались со связанными методами в гла-  
ве 29, когда изучали специальный метод \_\_call\_\_ . Однако, как мы узнаем   
в этом разделе, связанные методы обладают большей гибкостью, чем вы могли   
бы ожидать.  
В главе 19 мы узнали, что функции могут обрабатываться как обычные объ-  
екты. Методы – это разновидность объектов, напоминающая функции, – они

Методы – это объекты: связанные и несвязанные методы   
843  
могут присваиваться переменным, передаваться функциям, сохраняться   
в структурах данных и так далее. Доступ к методам класса осуществляется   
через экземпляр класса или через сам класс и, фактически, в языке Python   
имеется две разновидности методов:  
Несвязанные методы класса: без аргумента self  
Попытка обращения к функциональному атрибуту класса через имя клас-  
са возвращает объект несвязанного метода. Чтобы вызвать этот метод, не-  
обходимо явно передать ему объект экземпляра в виде первого аргумента.   
В Python 3.0 несвязанные методы напоминают простые функции и могут   
вызываться через имя класса. В версии 2.6 несвязанные методы – это совер-  
шенно иной тип данных, и они не могут вызываться без передачи им ссылки   
на экземпляр.  
Связанные методы экземпляра: пара self + функция  
Попытка обращения к функциональному атрибуту класса через имя экзем-  
пляра возвращает объект связанного метода. Интерпретатор автоматиче-  
ски упаковывает экземпляр с функцией в объект связанного метода, поэто-  
му вам не требуется передавать экземпляр в вызов такого метода.  
Обе разновидности методов – это полноценные объекты. Они могут передавать-  
ся между программными компонентами как обычные строки или числа. При   
запуске оба требуют наличия экземпляра в первом аргументе (то есть значения   
для аргумента self). Именно по этой причине в предыдущей главе было необ-  
ходимо явно передавать экземпляр при вызове методов суперкласса из методов   
подкласса – с технической точки зрения такие вызовы порождают объекты не-  
связанных методов.  
Вызывая объект связанного метода, интерпретатор автоматически подставля-  
ет экземпляр, который использовался при создании объекта связанного мето-  
да. Это означает, что объекты связанных методов обычно взаимозаменяемы   
с объектами простых функций и создание их особенно полезно в случае интер-  
фейсов, изначально ориентированных на использование функций (реалистич-  
ный пример приводится во врезке «Придется держать в уме: связанные методы   
и функции обратного вызова» ниже).  
Чтобы проиллюстрировать вышесказанное, предположим, что имеется следу-  
ющее определение класса:  
class Spam:  
 def doit(self, message):  
 print(message)  
В обычной ситуации мы создаем экземпляр и сразу же вызываем его метод для   
вывода содержимого аргумента:  
object1 = Spam()  
object1.doit(‘hello world’)  
Однако в действительности попутно создается объект связанного метода – как   
раз перед круглыми скобками в вызове метода. Т.е. мы можем получить свя-  
занный метод и без его вызова. Квалифицированное имя object.name – это вы-  
ражение, которое возвращает объект. В следующем примере это выражение   
возвращает объект связанного метода, в котором упакованы вместе экземпляр   
(object1) и метод (Spam.doit). Мы можем присвоить этот связанный метод дру-  
гому имени и затем использовать это имя для вызова, как простую функцию:

844   
Глава 30. Шаблоны проектирования с классами   
object1 = Spam()  
x = object1.doit # Объект связанного метода: экземпляр+функция  
x(‘hello world’) # То же, что и object1.doit(‘...’)  
С другой стороны, если для получения метода doit использовать имя класса,   
мы получим объект несвязанного метода, который просто ссылается на объект   
функции. Чтобы вызвать метод этого типа, необходимо явно передавать экзем-  
пляр класса в первом аргументе:  
object1 = Spam()  
t = Spam.doit # Объект несвязанного метода  
t(object1, ‘howdy’) # Передать экземпляр  
Те же самые правила действуют внутри методов класса, когда используются   
атрибуты аргумента self, которые ссылаются на функции в классе. Выражение   
self.method возвращает объект связанного метода, потому что self – это объект   
экземпляра:  
class Eggs:  
 def m1(self, n):  
 print(n)  
 def m2(self):  
 x = self.m1 # Еще один объект связанного метода  
 x(42) # Выглядит как обычная функция  
   
Eggs().m2() # Выведет 42  
В большинстве случаев вы будете вызывать методы немедленно, сразу же после   
указания квалифицированного имени, поэтому вы не всегда будете замечать,   
что попутно создается объект метода. Но как только вы начнете писать про-  
граммный код, который вызывает объекты единообразным способом, вам по-  
требуется проявить внимание к несвязанным методам, потому что обычно они   
требуют явной передачи экземпляра в первом аргументе.1  
В Python 3.0 несвязанные методы являются функциями  
В Python 3.0 было ликвидировано понятие несвязанных методов. Методы, ко-  
торые в этом разделе описываются как несвязанные методы, в версии 3.0 об-  
рабатываются как обычные функции. В большинстве ситуаций это никак не   
влияет на программный код – в любом случае при вызове метода относительно   
экземпляра в первом аргументе ему будет передан сам экземпляр.  
Однако для программ, где выполняется явная проверка типа, это изменение   
может оказаться существенным – если вывести тип метода, не получающего   
ссылку на экземпляр, в версии 2.6 будет выведено «unbound method» (несвя-  
unbound method» (несвя-  
 method» (несвя-  
method» (несвя-  
» (несвя-  
занный метод), а в версии 3.0 – «function» (функция).  
Кроме того, в версии 3.0 метод может вызываться без передачи ему ссылки на   
экземпляр при условии, что сам метод не ожидает ее получить и вызывается   
исключительно через обращение к имени класса. То есть в Python 3.0 ссыл-  
Python 3.0 ссыл-  
 3.0 ссыл-  
1   
Смотрите обсуждение статических методов класса в главе 31, которые представляют   
собой исключение из этого правила. Подобно связанным методам, они также могут   
выглядеть как обычные функции, потому что они не ожидают получить экземпляр   
в первом аргументе. В языке Python поддерживается три разновидности методов   
классов – методы экземпляров, статические методы и методы класса� в версии 3.0   
также допускается использовать обычные функции в классах.

Методы – это объекты: связанные и несвязанные методы   
845  
ка на экземпляр передается методу, только когда он вызывается относительно   
экземпляра. При вызове метода через имя класса передавать ему экземпляр   
требуется, только если он ожидает получить его:  
C:\misc> c:\python30\python  
>>> class Selfless:  
... def \_\_init\_\_(self, data):  
... self.data = data  
... def selfless(arg1, arg2): # Простая функция в 3.0  
... return arg1 + arg2  
... def normal(self, arg1, arg2): # Ожидает получить экземпляр при вызове  
... return self.data + arg1 + arg2  
...  
>>> X = Selfless(2)  
>>> X.normal(3, 4) # Экземпляр передается автоматически  
9  
>>> Selfless.normal(X, 3, 4) # Метод ожидает получить self:   
9 # передается вручную  
>>> Selfless.selfless(3, 4) # Вызов без экземпляра: работает в 3.0,   
7 # но завершается ошибкой в 2.6!  
В Python 2.6 последний вызов в этом примере завершится ошибкой, потому что   
по умолчанию несвязанные методы требуют, чтобы им передавалась ссылка на   
экземпляр, а в Python 3.0 ошибки не возникнет, потому что в этой версии та-  
Python 3.0 ошибки не возникнет, потому что в этой версии та-  
 3.0 ошибки не возникнет, потому что в этой версии та-  
кие методы интерпретируются, как простые функции, не требующие передачи   
экземпляра. Эта особенность версии 3.0 повышает вероятность появления слу-  
чайных ошибок (что если программист просто забудет передать экземпляр по   
невнимательности?), но, с другой стороны, она позволяет использовать методы   
как простые функции, при условии, что им не передается, и они не ожидают   
получить аргумент «self» со ссылкой на экземпляр.  
Следующие два вызова завершатся ошибкой в обеих версиях Python, 2.6   
и 3.0, – в первом случае (вызов относительно экземпляра) методу автоматиче-  
ски будет передан экземпляр, которого он не ожидает, а во втором (вызов через   
обращение к имени класса) метод не получит ожидаемый экземпляр:  
>>> X.selfless(3, 4)  
TypeError: selfless() takes exactly 2 positional arguments (3 given)  
   
>>> Selfless.normal(3, 4)  
TypeError: normal() takes exactly 3 positional arguments (2 given)  
Благодаря этому изменению в версии 3.0 отпала необходимость использовать   
декоратор staticmethod, описываемый в следующей главе, для оформления ме-  
тодов, которые не принимают аргумент self, вызываются только через имя   
класса и никогда не вызываются относительно экземпляра, – такие методы   
действуют как обычные функции, не получая аргумент с экземпляром. В вер-  
сии 2.6 вызовы таких методов будут приводить к ошибкам, если экземпляр не   
будет передаваться им вручную (подробнее о статических методах рассказыва-  
(подробнее о статических методах рассказыва-  
подробнее о статических методах рассказыва-  
 о статических методах рассказыва-  
о статических методах рассказыва-  
 статических методах рассказыва-  
статических методах рассказыва-  
 методах рассказыва-  
методах рассказыва-  
 рассказыва-  
рассказыва-  
ется в следующей главе).  
Важно помнить об этих различиях в поведении несвязанных методов в вер-  
сии 3.0, но с практической точки зрения связанные методы важнее. Связан-  
ные методы представляют собой объекты, объединяющие в себе экземпляры   
и функции, поэтому их можно интерпретировать, как обычные вызываемые   
объекты. Что это означает с точки зрения программирования, демонстрирует-  
ся в следующем разделе.

846   
Глава 30. Шаблоны проектирования с классами   
Более наглядный пример использования несвязанного метода   
в Python 3.0 и 2.6 приводится в файле lister.py, в разделе, посвя-  
щенном множественному наследованию, ниже, в этой главе.   
Классы в этом примере выводят значения методов, полученных   
относительно экземпляров и классов, в обеих версиях Python.  
Связанные методы и другие вызываемые объекты  
Как упоминалось выше, связанные методы могут интерпретироваться как   
обычные вызываемые объекты, то есть как обычные функции, – они могут   
произвольно передаваться между компонентами программы. Кроме того, так   
как связанные методы объединяют в себе функцию и экземпляр, они могут ис-  
пользоваться как любые другие вызываемые объекты и не требуют примене-  
ния специальных синтаксических конструкций для вызова. Ниже демонстри-  
руется возможность сохранения четырех объектов связанных методов в списке   
и их вызов как обычных функций:  
>>> class Number:  
... def \_\_init\_\_(self, base):  
... self.base = base  
... def double(self):  
... return self.base \* 2  
... def triple(self):  
... return self.base \* 3  
...  
>>> x = Number(2) # Объекты экземпляров класса  
>>> y = Number(3) # Атрибуты + методы  
>>> z = Number(4)  
>>> x.double() # Обычный непосредственный вызов  
4  
   
>>> acts = [x.double, y.double, y.triple, z.double] # Список связанных методов  
>>> for act in acts: # Вызовы откладываются  
... print(act()) # Вызов как функции  
...  
4  
6  
9  
8  
Как и простые функции, объекты связанных методов обладают информацией,   
позволяющей провести интроспекцию, включая атрибуты, обеспечивающие   
доступ к объекту экземпляра и к методу. Вызов связанного метода просто за-  
действует эту пару:  
>>> bound = x.double  
>>> bound.\_\_self\_\_, bound.\_\_func\_\_  
(<\_\_main\_\_.Number object at 0x0278F610>, <function double at 0x027A4ED0>)  
>>> bound.\_\_self\_\_.base  
2  
>>> bound() # Вызовет bound.\_\_func\_\_(bound.\_\_self\_\_, ...)  
4  
Фактически связанные методы – это лишь одна из разновидностей вызывае-  
мых объектов в языке Python. Как демонстрирует следующий пример, про-  
Python. Как демонстрирует следующий пример, про-  
. Как демонстрирует следующий пример, про-

Методы – это объекты: связанные и несвязанные методы   
847  
стые функции, определенные с помощью инструкции def или lambda, экзем-  
пляры, наследующие метод \_\_call\_\_, и связанные методы экземпляров могут   
обрабатываться и вызываться одинаковыми способами:  
>>> def square(arg):  
... return arg \*\* 2 # Простые функции (def или lambda)  
...  
>>> class Sum:  
... def \_\_init\_\_(self, val): # Вызываемые экземпляры  
... self.val = val  
... def \_\_call\_\_(self, arg):  
... return self.val + arg  
...  
>>> class Product:  
... def \_\_init\_\_(self, val): # Связанные методы  
... self.val = val  
... def method(self, arg):  
... return self.val \* arg  
...  
>>> sobject = Sum(2)  
>>> pobject = Product(3)  
>>> actions = [square, sobject, pobject.method] # Функция, экземпляр, метод  
   
>>> for act in actions: # Все 3 вызываются одинаково  
... print(act(5)) # Вызов любого вызываемого  
... # объекта с 1 аргументом  
25  
7  
15  
>>> actions[-1](5) # Индексы, генераторы, отображения  
15  
>>> [act(5) for act in actions]  
[25, 7, 15]  
>>> list(map(lambda act: act(5), actions))  
[25, 7, 15]  
Технически классы также принадлежат к категории вызываемых объектов,   
но обычно они вызываются для создания экземпляров, а не для выполнения   
какой-либо фактической работы, как показано ниже:  
>>> class Negate:  
... def \_\_init\_\_(self, val): # Классы - тоже вызываемые объекты  
... self.val = -val # Но вызываются для создания объектов  
... def \_\_repr\_\_(self): # Реализует вывод экземпляра  
... return str(self.val)  
...  
>>> actions = [square, sobject, pobject.method, Negate] # Вызвать класс тоже  
>>> for act in actions:  
... print(act(5))  
...  
25  
7  
15  
-5  
>>> [act(5) for act in actions] # Вызовет \_\_repr\_\_, а не \_\_str\_\_!  
[25, 7, 15, -5]

848   
Глава 30. Шаблоны проектирования с классами   
>>> table = {act(5): act for act in actions} # генератор словарей в 2.6/3.0  
>>> for (key, value) in table.items():  
... print(‘{0:2} => {1}’.format(key, value)) # метод str.format в 2.6/3.0  
...  
-5 => <class ‘\_\_main\_\_.Negate’>  
25 => <function square at 0x025D4978>  
15 => <bound method Product.method of <\_\_main\_\_.Product object at 0x025D0F90>>  
7 => <\_\_main\_\_.Sum object at 0x025D0F70>  
Как видите, связанные методы и модель вызываемых объектов вообще – это   
лишь некоторые из множества особенностей, обеспечивающие языку Python   
невероятную гибкость.  
Теперь, когда вы понимаете суть объектной модели методов, ознакомьтесь   
с примерами применения связанных методов во врезке «Придется держать   
в уме: связанные методы и функции обратного вызова» и еще раз прочитайте   
раздел предыдущей главы «\_\_call\_\_ обрабатывает вызовы», где обсуждаются   
функции обратного вызова.  
Придется держать в уме:   
связанные методы и функции обратного вызова  
В объектах связанных методов вместе с функцией автоматически сохра-  
няется экземпляр класса, поэтому они могут использоваться везде, где   
используются обычные функции. Одно из обычных мест, где можно уви-  
деть эту идею в действии, – это программный код, регистрирующий ме-  
тоды как обработчики событий в интерфейсе tkinter GUI (В Python 2.6   
он называется Tkinter). Ниже приводится простейший случай:  
def handler():  
 ...сохраняет информацию о состоянии в глобальных переменных...  
...  
widget = Button(text=’spam’, command=handler)  
Чтобы зарегистрировать обработчик события щелчка на кнопке, мы   
обычно передаем в аргументе с именем command вызываемый объект, ко-  
торый не имеет входных аргументов. Здесь часто используются имена   
простых функций (и lambda-выражения), но можно также передавать   
и методы классов – при условии, что они будут связанными методами:  
class MyWidget:  
 def handler(self):  
 ...сохраняет информацию о состоянии в self.attr...  
 def makewidgets(self):  
 b = Button(text=’spam’, command=self.handler)  
Здесь обработчик события self.handler – это объект связанного метода,   
в котором сохраняются self и MyWidget.handler. Так как аргумент self   
ссылается на оригинальный экземпляр, позднее, когда метод handler   
будет вызван для обработки события, он получит доступ к атрибутам   
экземпляра, где может сохраняться информация о состоянии между со-  
бытиями.

Множественное наследование: примесные классы   
849  
При использовании обычных функций для этих целей, как правило, ис-  
пользуются глобальные переменные. Другой способ обеспечения совме-  
стимости классов с прикладным интерфейсом, основанным на примене-  
нии функций, приводится в главе 29, где обсуждается метод перегрузки   
операторов \_\_call\_\_.  
Множественное наследование:   
примесные классы  
Многие объектно-ориентированные шаблоны проектирования основаны на   
объединении различных наборов методов. В строке заголовка инструкции class   
в круглых скобках может быть перечислено более одного суперкласса. В этом   
случае в игру вступает механизм множественного наследования – класс и его   
экземпляры наследуют имена из всех перечисленных суперклассов.  
При поиске атрибутов интерпретатор Python выполняет обход суперклассов,   
указанных в строке заголовка класса, слева направо, пока не будет найдено   
первое совпадение. С технической точки зрения из-за того, что любой супер-  
класс может иметь собственные суперклассы, процедура поиска может ока-  
заться достаточно сложной для крупных деревьев классов:  
 •  
В случае классических классов (которые использовались по умолчанию   
в версиях, предшествовавших Python 3.0) поиск атрибутов сначала продол-  
Python 3.0) поиск атрибутов сначала продол-  
 3.0) поиск атрибутов сначала продол-  
жается по направлению снизу вверх всеми возможными путями, вплоть до   
вершины дерева наследования, а затем слева направо.  
 •  
В случае классов нового стиля (для всех классов в 3.0) поиск атрибутов сна-  
чала выполняется по уровням дерева наследования, то есть в ширину – сле-  
ва направо (смотрите обсуждение классов нового стиля в следующей главе).  
Однако, независимо от модели, когда класс наследует несколько суперклассов,   
эти суперклассы просматриваются слева направо – в том порядке, в каком они   
следуют в заголовке инструкции class.  
Вообще множественное наследование хорошо использовать для моделирования   
объектов, принадлежащих более чем одной группе. Например, человек может   
быть инженером, писателем, музыкантом и так далее и наследовать свойства   
всех этих групп. В случае множественного наследования объекты приобретают   
совокупность черт, присущих всем его суперклассам.  
Пожалуй, самый распространенный случай, где используется множественное   
наследование, – это «смешивание» методов общего назначения из нескольких   
суперклассов. Обычно такие суперклассы называются примесными класса-  
ми – они предоставляют методы, которые добавляются в прикладные классы   
наследованием. В некотором смысле примесные классы напоминают модули:   
они предоставляют пакеты методов для использования в клиентских подклас-  
сах. Однако, в отличие от простых функций в модулях, методы в примесных   
классах обладают доступом к экземпляру self, что позволяет им использовать   
информацию, хранящуюся в экземпляре, и вызывать другие его методы. В сле-  
дующем разделе демонстрируется один распространенный пример использова-  
ния подобных инструментов.

850   
Глава 30. Шаблоны проектирования с классами   
Создание примесных классов, реализующих вывод  
Как мы уже видели, способ вывода экземпляра класса, используемый в Python   
по умолчанию, не отличается информативностью:  
>>> class Spam:  
... def \_\_init\_\_(self): # Нет метода \_\_repr\_\_ или \_\_str\_\_  
... self.data1 = “food”  
...  
>>> X = Spam()  
>>> print(X) # По умолчанию: класс, адрес  
<\_\_main\_\_.Spam instance at 0x00864818> # Вывод экземпляра в Python 2.6  
Как мы видели в предыдущей главе, когда рассматривали перегрузку опера-  
торов, существует возможность с помощью метода \_\_repr\_\_ или \_\_str\_\_ реали-  
зовать свою собственную операцию вывода. Но вместо того, чтобы воспроиз-  
водить метод \_\_repr\_\_ в каждом классе, который предполагается выводить на   
экран, почему бы не написать его всего один раз в классе инструментов общего   
назначения и не наследовать его во всех ваших классах?  
Для этого и используются примесные классы. Определив методы вывода в су-  
перклассе один раз, мы сможем повторно использовать его везде, где потребу-  
ется задействовать форматированный вывод. Мы уже видели инструменты,   
выполняющие действия, которые нам необходимы:  
 •  
Класс AttrDisplay в главе 27 обеспечивает вывод атрибутов экземпляра с по-  
мощью обобщенной реализации метода \_\_str\_\_, но он не предусматривает   
возможности подъема по дереву классов и использовался только в ситуации   
с простым наследованием.  
 •  
Модуль classtree.py в главе 28 содержит определения функций, позволяю-  
щих выполнять обход деревьев классов и выводить их схематическое пред-  
ставление, но они не выводят атрибуты объектов и не объединены в класс,   
который можно было бы наследовать.  
Здесь мы повторно рассмотрим приемы, использовавшиеся в этих примерах,   
и на их основе создадим три примесных класса, которые могли бы служить   
универсальными инструментами для отображения списков атрибутов экзем-  
пляра, унаследованных атрибутов и атрибутов всех объектов в дереве классов.   
Кроме того, мы попробуем применить наши инструменты в режиме множе-  
ственного наследования и используем приемы программирования, благодаря   
которым наши классы будут лучше подходить на роль универсальных инстру-  
ментов.  
Получение списка атрибутов экземпляра с помощью \_\_dict\_\_  
Начнем с самого простого – с получения списка атрибутов, присоединенных   
к экземпляру. Ниже приводится определение примесного класса ListInstance,   
находящееся в файле lister.py, реализующего метод \_\_str\_\_ для всех классов,   
которые будут включать его в свой список суперклассов в заголовках инструк-  
ций class. Поскольку этот инструмент является классом, реализованная в нем   
логика вывода может использоваться экземплярами любых его подклассов:  
# Файл lister.py  
   
class ListInstance:  
 “””

Множественное наследование: примесные классы   
851  
 Примесный класс, реализующий получение форматированной строки при вызове   
 функций print() и str() с экземпляром в виде аргумента, через наследование   
 метода \_\_str\_\_, реализованного здесь; отображает только атрибуты   
 экземпляра; self – экземпляр самого нижнего класса в дереве наследования;   
 во избежание конфликтов с именами атрибутов клиентских классов использует   
 имена вида \_\_X   
 “””  
 def \_\_str\_\_(self):  
 return ‘<Instance of %s, address %s:\n%s>’ % (  
 self.\_\_class\_\_.\_\_name\_\_, # Имя клиентского класса  
 id(self), # Адрес экземпляра  
 self.\_\_attrnames()) # Список пар name=value  
 def \_\_attrnames(self):  
 result = ‘’  
 for attr in sorted(self.\_\_dict\_\_): # Словарь атрибутов  
 result += ‘\tname %s=%s\n’ % (attr, self.\_\_dict\_\_ [attr])  
 return result  
Для извлечения имени класса экземпляра и списка атрибутов в классе ListIn-  
stance используются уже известные нам приемы:  
 •  
Каждый экземпляр имеет встроенный атрибут \_\_class\_\_, ссылающийся на   
класс, из которого он был создан, а каждый класс имеет атрибут \_\_name\_\_,   
ссылающийся на имя класса. Таким образом, выражение self.\_\_class\_\_.\_\_  
name\_\_ извлекает имя класса экземпляра.  
 •  
Основная работа этого класса заключается в том, чтобы просмотреть сло-  
варь атрибутов экземпляра (который представлен атрибутом \_\_dict\_\_)   
и сконструировать строку, содержащую имена и значения всех атрибутов   
экземпляра. Чтобы обеспечить единообразное представление во всех верси-  
ях Python, ключи словаря сортируются в порядке возрастания.  
В этом отношении класс ListInstance напоминает класс AttrDisplay из гла-  
вы 27 – фактически, он до определенной степени является разновидностью реа-  
лизации AttrDisplay. Однако в нашем классе используются два новых приема:  
 •  
Он отображает адрес экземпляра в памяти, вызывая встроенную функцию   
id, которая возвращает адрес объекта (по определению – уникальный иден-  
тификатор объекта, который может пригодиться при последующих дора-  
ботках этого программного кода).  
 •  
Использует псевдочастные имена для своих методов: \_\_attrnames. Как мы   
узнали выше в этой главе, интерпретатор автоматически искажает любые   
такие имена, дополняя их именем вмещающего класса (в данном случае   
имя \_\_attrnames превращается в имя \_ListInstance\_\_attrnames). Это правило   
также распространяется на атрибуты класса и на атрибуты экземпляра,   
присоединяемые к объекту self. Это полезное свойство для таких универ-  
сальных инструментов, как этот класс, так как оно гарантирует отсутствие   
конфликтов с именами в клиентских подклассах.  
Класс ListInstance определяет метод \_\_str\_\_ перегрузки операторов, поэтому   
при выводе экземпляров, наследующих этот класс, они автоматически будут   
выводить свои атрибуты, давая о себе больше информации, чем просто свой   
адрес. Ниже демонстрируется пример использования этого класса в режиме   
простого наследования (этот пример одинаково работает в Python 3.0 и 2.6):

852   
Глава 30. Шаблоны проектирования с классами   
>>> from lister import ListInstance  
>>> class Spam(ListInstance): # Наследует метод \_\_str\_\_  
... def \_\_init\_\_(self):  
... self.data1 = ‘food’  
...  
>>> x = Spam()  
>>> print(x) # print() и str() вызывают \_\_str\_\_  
<Instance of Spam, address 40240880:  
 name data1=food  
>  
Вы можете также получить эти результаты в виде строки, без вывода на экран,   
с помощью функции str. При этом функция автоматического вывода в инте-  
рактивной оболочке по-прежнему использует формат по умолчанию для пред-  
ставления экземпляра:  
>>> str(x)  
‘<Instance of Spam, address 40240880:\n\tname data1=food\n>’  
>>> x # По умолчанию используется \_\_repr\_\_  
<\_\_main\_\_.Spam object at 0x026606F0>  
Класс ListInstance пригодится в любых классах, которые вам придется соз-  
давать, – даже в классах, уже имеющих один или более суперклассов. Здесь   
в игру вступает механизм множественного  наследования: добавляя ListIn-  
stance в список суперклассов в заголовке инструкции class (то есть, «подме-  
шав» его), вы получаете реализацию метода \_\_str\_\_ «в подарок», что не мешает   
наследовать существующие суперклассы. Эта возможность демонстрируется   
в файле testmixin.py:  
# Файл testmixin.py  
   
from lister import \* # Импортировать инструментальные классы  
   
class Super:  
 def \_\_init\_\_(self): # Метод \_\_init\_\_ суперкласса  
 self.data1 = ‘spam’ # Создать атрибуты экземпляра  
 def ham(self):  
 pass  
   
class Sub(Super, ListInstance): # Подмешать методы ham и \_\_str\_\_  
 def \_\_init\_\_(self): # Инструментальные классы имеют доступ к self  
 Super.\_\_init\_\_(self)  
 self.data2 = ‘eggs’ # Добавить атрибуты экземпляра  
 self.data3 = 42  
 def spam(self): # Определить еще один метод  
 pass  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 X = Sub()  
 print(X) # Вызовет подмешанный метод \_\_str\_\_  
Здесь класс Sub наследует имена из двух классов, Super и ListInstance, – этот   
объект состоит из своих собственных имен и из имен обоих суперклассов. Если   
создать и вывести экземпляр класса Sub, автоматически будет получено адап-  
тированное его представление, воспроизведенное методом \_\_str\_\_ примесного   
класса ListInstance (данный сценарий выведет одинаковые результаты в обеих   
версиях Python, 3.0 и 2.6, за исключением адресов объектов):

Множественное наследование: примесные классы   
853  
C:\misc> C:\python30\python testmixin.py  
<Instance of Sub, address 40962576:  
 name data1=spam  
 name data2=eggs  
 name data3=42  
>  
Реализация класса ListInstance в состоянии работать с любыми классами, по-  
тому что аргумент self ссылается на экземпляр подкласса, который наследует   
ListInstance, каким бы этот подкласс ни был. В некотором смысле, примесные   
классы – это классы, эквивалентные модулям, потому что они упаковывают   
методы, которые будут полезны самым разным клиентам. Ниже демонстриру-  
ется работа класса ListInstance в режиме простого наследования с экземпляра-  
ми различных классов, к которым присоединяются дополнительные атрибуты   
за пределами определения класса:  
>>> import lister  
>>> class C(lister.ListInstance): pass  
...  
>>> x = C()  
>>> x.a = 1; x.b = 2; x.c = 3  
>>> print(x)  
<Instance of C, address 40961776:  
 name a=1  
 name b=2  
 name c=3  
>  
Примесные классы имеют не только практическую ценность, они также позво-  
ляют оптимизировать сопровождение программного кода, подобно любым дру-  
гим классам. Например, если позднее вы решите усовершенствовать класс Lis-  
tInstance, чтобы его метод \_\_str\_\_ выводил также все атрибуты класса, которые   
были унаследованы экземпляром, вы без опаски сможете сделать это – так как   
это наследуемый метод, изменение в реализации \_\_str\_\_ автоматически начнет   
действовать и во всех подклассах, которые импортируют и подмешивают его.   
Так как, собственно, это «позднее» уже наступило, перейдем к следующему   
разделу и посмотрим, как может выглядеть такое усовершенствование.  
Получение списка атрибутов экземпляра   
с помощью функции dir  
В настоящий момент наш класс ListInstance отображает только атрибуты эк-  
земпляра (то есть имена, присоединенные к самому объекту экземпляра). Од-  
нако совсем несложно усовершенствовать реализацию класса так, чтобы ото-  
бражались все атрибуты, доступные экземпляру, – как его собственные, так   
и унаследованные от его классов. Хитрость заключается в том, чтобы вместо   
сканирования словаря \_\_dict\_\_ экземпляра использовать встроенную функ-  
цию dir, – словарь хранит только атрибуты экземпляра, а функция dir, начи-  
ная с версии Python 2.2, возвращает список всех унаследованных атрибутов.  
Ниже приводится усовершенствованная реализация, действующая по этой   
схеме, – я переименовал класс, чтобы упростить тестирование, но если бы эта   
реализация заменила оригинальную версию, она автоматически начала бы   
действовать во всех существующих клиентах:

854   
Глава 30. Шаблоны проектирования с классами   
# Файл lister.py, продолжение  
   
class ListInherited:  
 “””  
 Использует функцию dir() для получения списка атрибутов самого экземпляра   
 и атрибутов, унаследованных экземпляром от его классов; в Python 3.0   
 выводится больше имен атрибутов, чем в 2.6, потому что классы нового стиля   
 в конечном итоге наследуют суперкласс object; метод getattr() позволяет   
 получить значения унаследованных атрибутов, отсутствующих в self.\_\_dict\_\_;   
 реализует метод \_\_str\_\_, а не \_\_repr\_\_, потому что в противном случае   
 данная реализация может попасть в бесконечный цикл при выводе связанных   
 методов!  
 “””  
 def \_\_str\_\_(self):  
 return ‘<Instance of %s, address %s:\n%s>’ % (  
 self.\_\_class\_\_.\_\_name\_\_, # Имя класса экземпляра  
 id(self), # Адрес экземпляра  
 self.\_\_attrnames()) # Список пар name=value  
 def \_\_attrnames(self):  
 result = ‘’  
 for attr in dir(self): # Передать экземпляр функции dir()  
 if attr[:2] == ‘\_\_’ and attr[-2:] == ‘\_\_’: # Пропустить   
 result += ‘\tname %s=<>\n’ % attr # внутренние имена  
 else:  
 result += ‘\tname %s=%s\n’ % (attr, getattr(self, attr))  
 return result  
Обратите внимание, что данная реализация пропускает имена вида \_\_X\_\_, –   
в большинстве случаев эти имена предназначены для внутреннего использова-  
ния и их обычно бывает нежелательно выводить в общем списке. Кроме того,   
в данной версии потребовалось использовать встроенную функцию getattr для   
извлечения значений атрибутов по именам в виде строк и отказаться от исполь-  
зования словаря с атрибутами – некоторые имена, доступные экземпляру, не   
принадлежат самому экземпляру, а функция getattr поддерживает поиск имен   
в дереве наследования.   
Чтобы проверить новую версию, изменим файл testmixin.py так, чтобы задей-  
ствовать новый класс:  
class Sub(Super, ListInherited): # Подмешать \_\_str\_\_  
Результат работы этой версии файла зависит от версии интерпретатора. В Py-  
Py-  
thon 2.6 будет получен следующий список – обратите внимание, как действует   
механизм искажения имен для имени метода \_\_attrnames в классе ListInherit-  
ed (я сократил полное отображаемое значение, чтобы уместить его по ширине   
страницы):  
C:\misc> c:\python26\python testmixin.py  
<Instance of Sub, address 40073136:  
 name \_ListInherited\_\_attrnames=<bound method Sub.\_\_attrnames of <...>>  
 name \_\_doc\_\_=<>  
 name \_\_init\_\_=<>  
 name \_\_module\_\_=<>  
 name \_\_str\_\_=<>  
 name data1=spam  
 name data2=eggs  
 name data3=42

Множественное наследование: примесные классы   
855  
 name ham=<bound method Sub.ham of <\_\_main\_\_.Sub instance at ...>>  
 name spam=<bound method Sub.spam of <\_\_main\_\_.Sub instance at ...>>  
>  
В Python 3.0 список содержит гораздо больше атрибутов, потому что все клас-  
Python 3.0 список содержит гораздо больше атрибутов, потому что все клас-  
 3.0 список содержит гораздо больше атрибутов, потому что все клас-  
сы являются классами «нового стиля» и наследуют атрибуты и методы от су-  
перкласса object (подробнее о нем рассказывается в главе 31). Так как из супер-  
класса по умолчанию наследуется достаточно большое количество имен, я опу-  
стил многие из них – запустите этот пример у себя, чтобы получить полный   
список:  
C:\misc> c:\python30\python testmixin.py  
<Instance of Sub, address 40831792:  
 name \_ListInherited\_\_attrnames=<bound method Sub.\_\_attrnames of <...>>  
 name \_\_class\_\_=<>  
 name \_\_delattr\_\_=<>  
 name \_\_dict\_\_=<>  
 name \_\_doc\_\_=<>  
 name \_\_eq\_\_=<>  
 ...часть имен опущена...  
 name \_\_repr\_\_=<>  
 name \_\_setattr\_\_=<>  
 name \_\_sizeof\_\_=<>  
 name \_\_str\_\_=<>  
 name \_\_subclasshook\_\_=<>  
 name \_\_weakref\_\_=<>  
 name data1=spam  
 name data2=eggs  
 name data3=42  
 name ham=<bound method Sub.ham of <\_\_main\_\_.Sub object at 0x026F0B30>>  
 name spam=<bound method Sub.spam of <\_\_main\_\_.Sub object at ...>>  
>  
Следует также заметить, что теперь, когда мы предусматриваем вывод уна-  
следованных методов, мы должны поместить реализацию перегрузки вывода   
в метод \_\_str\_\_, а не в \_\_repr\_\_. В методе \_\_repr\_\_ эта реализация будет попадать   
в бесконечный цикл – при попытке отобразить значение метода вызывается ме-  
тод \_\_repr\_\_ класса, которому принадлежит отображаемый метод, чтобы вы-  
вести информацию о классе. То есть, если метод \_\_repr\_\_ класса ListInherited   
попытается отобразить метод, тогда при выводе информации о классе, которо-  
му принадлежит отображаемый метод, снова будет вызван метод \_\_repr\_\_ клас-  
са ListInherited. Эту проблему сложно заметить, но она существует! Измените   
имя метода \_\_str\_\_ на \_\_repr\_\_, чтобы убедиться в этом. Если вам необходимо   
использовать метод \_\_repr\_\_ в подобной ситуации, вы можете избежать заци-  
кливания, сравнивая тип атрибута со значением types.MethodType из стандарт-  
ной библиотеки с помощью функции isinstance и пропуская элементы, для ко-  
торых функция вернет значение True.  
Получение списка атрибутов   
с привязкой к объектам в дереве классов  
Теперь добавим последнее усовершенствование. В настоящий момент класс   
ListInherited ничего не сообщает о том, из каких классов были унаследованы   
те или иные имена. Однако, как мы видели в примере classtree.py в конце гла-  
вы 28, реализовать обход дерева классов совсем не сложно. Следующий при-

856   
Глава 30. Шаблоны проектирования с классами   
месный класс использует тот же прием для отображения атрибутов, сгруппи-  
рованных по классам, где они определены, – он выводит схему всего дерева   
классов, попутно отображая атрибуты, присоединенные к каждому объекту.   
Это достигается за счет обхода дерева наследования, от атрибута \_\_class\_\_ эк-  
земпляра к его классу и затем рекурсивно от атрибута \_\_bases\_\_ класса ко всем   
суперклассам, сканируя словари \_\_dicts\_\_ в процессе обхода:  
# Файл lister.py, продолжение  
   
class ListTree:  
 “””  
 Примесный класс, в котором метод \_\_str\_\_ просматривает все дерево классов   
 и составляет список атрибутов всех объектов, находящихся в дереве выше   
 self; вызывается функциями print(), str() и возвращает сконструированную   
 строку со списком; во избежание конфликтов с именами атрибутов клиентских   
 классов использует имена вида \_\_X; для рекурсивного обхода суперклассов   
 использует выражение-генератор; чтобы сделать подстановку значений более   
 очевидной, использует метод str.format()   
 “””  
 def \_\_str\_\_(self):  
 self.\_\_visited = {}  
 return ‘<Instance of {0}, address {1}:\n{2}{3}>’.format(  
 self.\_\_class\_\_.\_\_name\_\_,  
 id(self),  
 self.\_\_attrnames(self, 0),  
 self.\_\_listclass(self.\_\_class\_\_, 4))  
 def \_\_listclass(self, aClass, indent):  
 dots = ‘.’ \* indent  
 if aClass in self.\_\_visited:  
 return ‘\n{0}<Class {1}:, address {2}: (see above)>\n’.format(  
 dots,  
 aClass.\_\_name\_\_,  
 id(aClass))  
 else:  
 self.\_\_visited[aClass] = True  
 genabove = (self.\_\_listclass(c, indent+4)   
 for c in aClass.\_\_bases\_\_)  
 return ‘\n{0}<Class {1}, address {2}:\n{3}{4}{5}>\n’.format(  
 dots,  
 aClass.\_\_name\_\_,  
 id(aClass),  
 self.\_\_attrnames(aClass, indent),  
 ‘’.join(genabove),  
 dots)  
   
 def \_\_attrnames(self, obj, indent):  
 spaces = ‘ ‘ \* (indent + 4)  
 result = ‘’  
 for attr in sorted(obj.\_\_dict\_\_):  
 if attr.startswith(‘\_\_’) and attr.endswith(‘\_\_’):  
 result += spaces + ‘{0}=<>\n’.format(attr)  
 else:  
 result += spaces + ‘{0}={1}\n’.format(attr, getattr(obj,   
 attr))  
 return result

Множественное наследование: примесные классы   
857  
Обратите внимание, как используется выражение-генератор для реализации   
рекурсивного обхода суперклассов, – оно активируется вложенным строковым   
методом join. Обратите также внимание, что в этой версии вместо оператора %   
форматирования используется метод format, доступный в Python 3.0 и 2.6, – это   
позволило сделать подстановку более очевидной. Когда выполняется подста-  
новка множества значений, как в данном случае, явная нумерация аргументов   
может упростить чтение такого программного кода. Проще говоря, в этой вер-  
сии мы заменили первую из следующих строк второй:  
return ‘<Instance of %s, address %s:\n%s%s>’ % (...) # Выражение  
return ‘<Instance of {0}, address {1}:\n{2}{3}>’.format(...) # Метод  
Теперь изменим файл testmixin.py, чтобы испытуемый класс наследовал новый   
класс ListTree:  
class Sub(Super, ListTree): # Подмешать \_\_str\_\_  
В Python 2.6 этот сценарий выведет следующее:  
C:\misc> c:\python26\python testmixin.py  
<Instance of Sub, address 40728496:  
 \_ListTree\_\_visited={}  
 data1=spam  
 data2=eggs  
 data3=42  
   
....<Class Sub, address 40701168:  
 \_\_doc\_\_=<>  
 \_\_init\_\_=<>  
 \_\_module\_\_=<>  
 spam=<unbound method Sub.spam>  
   
........<Class Super, address 40701120:  
 \_\_doc\_\_=<>  
 \_\_init\_\_=<>  
 \_\_module\_\_=<>  
 ham=<unbound method Super.ham>  
........>  
   
........<Class ListTree, address 40700688:  
 \_ListTree\_\_attrnames=<unbound method ListTree.\_\_attrnames>  
 \_ListTree\_\_listclass=<unbound method ListTree.\_\_listclass>  
 \_\_doc\_\_=<>  
 \_\_module\_\_=<>  
 \_\_str\_\_=<>  
........>  
....>  
>  
Обратите внимание, что теперь в версии 2.6 методы определяются как несвя-  
занные (unbound). Это обусловлено тем, что информацию о них мы извлекаем   
теперь непосредственно из классов, а не из экземпляров. Обратите также вни-  
мание на результат искажения имени таблицы \_\_visited в словаре атрибутов   
экземпляра – только у последних неудачников такое имя могло бы совпасть   
с каким-то другим.  
Если запустить этот сценарий под управлением Python 3.0, мы снова полу-  
Python 3.0, мы снова полу-  
 3.0, мы снова полу-  
чим более длинный список атрибутов и суперклассов. Обратите внимание, что

858   
Глава 30. Шаблоны проектирования с классами   
в версии 3.0 несвязанные методы идентифицируются как простые функции,   
о чем уже говорилось выше в этой главе (и снова, чтобы сэкономить простран-  
ство, я удалил большую часть встроенных атрибутов класса object� запустите   
этот сценарий у себя, чтобы получить полный список):  
C:\misc> c:\python30\python testmixin.py  
<Instance of Sub, address 40635216:  
 \_ListTree\_\_visited={}  
 data1=spam  
 data2=eggs  
 data3=42  
   
....<Class Sub, address 40914752:  
 \_\_doc\_\_=<>  
 \_\_init\_\_=<>  
 \_\_module\_\_=<>  
 spam=<function spam at 0x026D53D8>  
   
........<Class Super, address 40829952:  
 \_\_dict\_\_=<>  
 \_\_doc\_\_=<>  
 \_\_init\_\_=<>  
 \_\_module\_\_=<>  
 \_\_weakref\_\_=<>  
 ham=<function ham at 0x026D5228>  
   
............<Class object, address 505114624:  
 \_\_class\_\_=<>  
 \_\_delattr\_\_=<>  
 \_\_doc\_\_=<>  
 \_\_eq\_\_=<>  
 ...часть строк опущена...  
 \_\_repr\_\_=<>  
 \_\_setattr\_\_=<>  
 \_\_sizeof\_\_=<>  
 \_\_str\_\_=<>  
 \_\_subclasshook\_\_=<>  
............>  
........>  
   
........<Class ListTree, address 40829496:  
 \_ListTree\_\_attrnames=<function \_\_attrnames at 0x026D5660>  
 \_ListTree\_\_listclass=<function \_\_listclass at 0x026D56A8>  
 \_\_dict\_\_=<>  
 \_\_doc\_\_=<>  
 \_\_module\_\_=<>  
 \_\_str\_\_=<>  
 \_\_weakref\_\_=<>  
   
............<Class object:, address 505114624: (see above)>  
........>  
....>  
>  
В этой версии исключается возможность многократного перечисления одного   
и того же объекта класса за счет использования таблицы посещенных классов   
(именно поэтому в вывод включены значения, возвращаемые функцией id для   
объектов, – они могут служить ключами отображаемых элементов). Как и в

Множественное наследование: примесные классы   
859  
реализации функции транзитивной перезагрузки модулей, приводившейся   
в главе 24, словарь помогает избежать повторений и зацикливаний, благодаря   
тому, что объекты классов могут использоваться в качестве ключей словаря, –   
то же самое можно было бы реализовать на основе множества.  
Кроме того, в этой версии снова использован прием, реализующий пропуск   
внутренних объектов с именами вида \_\_X\_\_. Если закомментировать провер-  
ку этих имен, они будут отображаться наряду с обычными атрибутами. Ниже   
приводится выдержка из результатов, полученных в Python 2.6, произведен-  
Python 2.6, произведен-  
 2.6, произведен-  
ных сценарием с закомментированной проверкой (полный список получился   
намного длиннее, а в версии 3.0 он еще больше, что может служить одной из   
причин, почему такие имена лучше пропустить!):  
C:\misc> c:\python26\python testmixin.py  
...часть строк опущена...  
   
........<Class ListTree, address 40700688:  
 \_ListTree\_\_attrnames=<unbound method ListTree.\_\_attrnames>  
 \_ListTree\_\_listclass=<unbound method ListTree.\_\_listclass>  
 \_\_doc\_\_=  
 Примесный класс, в котором метод \_\_str\_\_ просматривает все дерево классов   
 и составляет список атрибутов всех объектов, находящихся в дереве выше   
 self; вызывается функциями print(), str() и возвращает сконструированную   
 строку со списком; во избежание конфликтов с именами атрибутов клиентских   
 классов использует имена вида \_\_X; для рекурсивного обхода суперклассов   
 использует выражение-генератор; чтобы сделать подстановку значений более   
 очевидной, использует метод str.format()   
   
 \_\_module\_\_=lister  
 \_\_str\_\_=<unbound method ListTree.\_\_str\_\_>  
........>  
Забавы ради попробуйте смешать этот класс с каким-нибудь более существен-  
ным классом, например с классом Button из модуля tkinter. Вообще, класс List-  
Tree желательно указать первым в списке (крайним слева) суперклассов в за-  
головке инструкции class, чтобы его метод \_\_str\_\_ имел преимущество, – класс   
Button имеет собственный метод \_\_str\_\_, а процедура поиска при множествен-  
ном наследовании в первую очередь просматривает суперкласс, стоящий пер-  
вым в списке. У меня вывод сценария получился очень массивным (более 18000   
символов), поэтому если вам интересно увидеть полный список, запустите его   
у себя (не забудьте, что в Python 2.6 модуль tkinter называется Tkinter):  
>>> from lister import ListTree  
>>> from tkinter import Button # Оба класса имеют метод \_\_str\_\_  
>>> class MyButton(ListTree, Button): pass # ListTree - первый: будет   
... # использоваться его метод \_\_str\_\_  
>>> B = MyButton(text=’spam’)  
>>> open(‘savetree.txt’, ‘w’).write(str(B)) # Сохранить в файл для   
18247 # последующего просмотра  
>>> print(B) # Вывести результаты  
<Instance of MyButton, address 44355632:  
 \_ListTree\_\_visited={}  
 \_name=44355632  
 \_tclCommands=[]  
 ...очень много строк опущено...  
>

860   
Глава 30. Шаблоны проектирования с классами   
Безусловно, можно было бы продолжить дальнейшее усовершенствование   
(вполне естественно было бы на следующем шаге реализовать вывод схемы де-  
рева в графическом интерфейсе), но я оставлю эту работу вам в качестве са-  
мостоятельного упражнения. Кроме того, в упражнениях, в конце этой части   
книги, будет предложено усовершенствовать этот пример так, чтобы он выво-  
дил имена суперклассов в круглых скобках в первых строках с именами экзем-  
пляров и классов.  
ООП тесно связано с повторным использованием программного кода, и при-  
месные классы в этом отношении представляют собой мощный инструмент.   
Как почти все в программировании, множественное наследование может   
быть благом при грамотном применении� но при неаккуратном и чрезмерном   
употреблении эта возможность может осложнить вам жизнь. Мы вернемся   
к этому вопросу как к одной из типичных проблем в конце следующей главы.   
В этой главе мы также познакомимся с возможностью (в классах нового стиля)   
изменять порядок поиска для одного специального случая множественного   
наследования.  
Поддержка  слотов: Поскольку классы ListInstance и ListTree,   
представленные здесь, выполняют сканирование словарей эк-  
земпляров, они не поддерживают атрибуты, хранящиеся в сло-  
тах – новой и относительно редко используемой особенности,   
с которой мы встретимся в следующей главе, где мы увидим,   
как атрибуты экземпляров объявляются в атрибуте \_\_slots\_\_   
класса. Например, если в определение класса Super, в файле   
testmixin.py,   
добавить   
инструкцию   
присваивания   
\_\_  
slots\_\_=[‘data1’], а в определение класса Sub добавить инструк-  
цию \_\_slots\_\_=[‘data3’], классы ListInstance и ListTree обнаружат   
в экземпляре только атрибут data2 – класс ListTree выведет атри-  
буты data1 и data3, но только как атрибуты объектов классов Su-  
per и Sub, и со значениями в специальном формате (технически   
они являются дескрипторами уровня класса).  
Чтобы добавить поддержку слотов, измените цикл, реализую-  
щий сканирование словарей \_\_dict\_\_, так, чтобы он дополни-  
тельно выполнял итерации через списки \_\_slots\_\_, используя   
программный код, который будет представлен в следующей   
главе, а для извлечения значений вместо индексирования сло-  
варя \_\_dict\_\_ используйте встроенную функцию getattr (это уже   
сделано в классе ListTree). Поскольку экземпляры наследуют   
атрибут \_\_slots\_\_ только из ближайшего к ним класса, вам мо-  
жет потребоваться прибегнуть к хитрости для обработки атри-  
бутов \_\_slots\_\_, присутствующих в нескольких суперклассах   
(класс ListTree уже отображает их как атрибуты классов). Класс   
ListInherited лишен этих недостатков, потому что функция dir   
уже объединяет имена из словаря \_\_dict\_\_ и имена из атрибутов   
\_\_slots\_\_ всех классов.  
Однако мы могли бы просто позволить нашей реализации обра-  
батывать атрибуты в слотах так, как они обрабатываются сей-  
час, вместо того, чтобы усложнять программный код, пытаясь   
обеспечить поддержку этой новой и редко используемой осо-  
бенности. Слоты и обычные атрибуты экземпляра – это разные

Классы – это объекты: универсальные фабрики объектов   
861  
типы имен. Со слотами мы познакомимся поближе в следующей   
главе. Я опустил рассмотрение этой особенности в наших приме-  
рах, только чтобы избежать опережающей ссылки (не учитывая   
этого примечания, конечно!), – для действующей реализации   
это не оправданно, но оправданно для книги.  
Классы – это объекты:   
универсальные фабрики объектов  
Иногда бывает необходимо, чтобы объекты создавались в ответ на сложившие-  
ся условия, которые невозможно предсказать на этапе разработки программы.   
Фабричный шаблон проектирования позволяет реализовать такой подход.   
В значительной степени благодаря высокой гибкости языка Python фабрики   
могут принимать самые разнообразные формы, многие из которых вовсе не вы-  
глядят чем-то особенным.  
Классы – это объекты, поэтому их легко можно передавать между компонента-  
ми программы, сохранять в структурах данных и так далее. Можно также пе-  
редавать классы функциям, которые создают объекты произвольных типов, –   
в кругах, связанных с ООП, такие функции иногда называют фабриками.   
В языках со строгой типизацией, таких как C++, реализация таких функций –   
достаточно сложная задача, но в языке Python она становится почти тривиаль-  
ной. Синтаксическая конструкция, с которой мы познакомились в главе 18,   
может вызывать любые классы с любым числом аргументов конструкторов за   
один присест, генерируя экземпляр любого типа:1  
def factory(aClass, \*args): # Кортеж с переменным числом аргументов  
 return aClass(\*args) # Вызов aClass (или apply, только в 2.6)  
   
class Spam:  
 def doit(self, message):  
 print(message)  
   
class Person:  
 def \_\_init\_\_(self, name, job):  
 self.name = name  
 self.job = job  
   
object1 = factory(Spam) # Создать объект Spam  
object2 = factory(Person, “Guido”, “guru”) # Создать объект Person  
В этом фрагменте определена функция-генератор объектов с именем factory.   
Она ожидает получить объект класса (любого) вместе с одним или более аргу-  
ментами конструктора класса. Функция использует специальный синтаксис   
вызова с переменным числом аргументов, чтобы создать и вернуть экземпляр.  
1   
Фактически эта синтаксическая конструкция может вызывать любой вызываемый   
объект, включая функции, классы и методы. Функция factory в этом примере также   
может вызывать любые вызываемые объекты, а не только классы (несмотря на имя   
аргумента). Кроме того, как мы узнали в главе 18, в Python 2.6 можно использовать   
не только конструкцию aClass(\*args), но и альтернативную ей встроенную функцию   
apply(aClass, args), которая была удалена в Python 3.0 из-за ее избыточности и огра-  
ниченных возможностей.

862   
Глава 30. Шаблоны проектирования с классами   
Остальная часть примера просто определяет два класса и генерирует экзем-  
пляры этих классов, передавая классы функции factory. И это единственная   
фабричная функция, которую вам придется написать на языке Python, – она   
работает с любыми классами и с любыми аргументами конструктора.  
Следует заметить, что здесь возможно одно небольшое улучшение, которое за-  
ключается в обеспечении поддержки именованных аргументов конструктора�   
фабричная функция может собрать их в аргумент \*\*args и передать в вызов   
класса в виде третьего аргумента:  
def factory(aClass, \*args, \*\*kwargs): # +kwargs   
 return aClass(\*args, \*\*kwargs) # Вызвать aClass  
К настоящему времени вы должны знать, что в языке Python все сущее явля-  
Python все сущее явля-  
 все сущее явля-  
ется «объектом», включая и сами классы, которые в других языках, таких как   
С++, являются лишь объявлениями для компилятора. Однако, как упомина-  
лось в начале этой части книги, в языке Python только объекты, порожденные   
из классов, являются субъектами ООП.  
Зачем нужны фабрики?  
Итак, чем же хороша функция factory (помимо иллюстрации того, что клас-  
сы являются объектами)? К сожалению, довольно сложно продемонстрировать   
применение этого шаблона проектирования, потому что для этого необходимо   
привести фрагмент программного кода больший, чем позволяет пространство   
книги. Тем не менее такая фабрика могла бы помочь изолировать программ-  
ный код от динамически настраиваемой конструкции объекта.  
Вспомним пример функции processor, представленный в главе 25, и затем при-  
мер применения принципа композиции в этой главе. В обоих случаях при-  
нимаются объекты, выполняющие чтение и запись обрабатываемого потока   
данных. В оригинальной версии этого примера мы вручную передавали эк-  
земпляры специализированных классов, таких как FileWriter и SocketReader,   
для адаптации под обрабатываемые потоки данных – позднее мы передавали   
жестко заданные объекты файла, потока и преобразования. В других случаях   
внешние источники данных могут определяться настройками в конфигураци-  
онных файлах или в элементах управления графического интерфейса.  
В таком динамическом мире не представляется возможным жестко задавать   
в сценарии объекты, реализующие интерфейс к потоку данных, но вполне воз-  
можно создавать их во время выполнения, в соответствии с содержимым кон-  
фигурационных файлов.  
Например, в файле с настройками может определяться имя класса потока,   
который должен быть импортирован из модуля, и дополнительные аргумен-  
ты конструктора. В этой ситуации могла бы пригодиться фабричная функция   
или эквивалентный ей фрагмент программного кода, потому что они могли   
бы позволить нам получить и передать классы, не определяя их заранее в про-  
грамме. В действительности возможно представить себе, что требуемые классы   
даже не существовали в тот момент, когда мы писали свой программный код:  
classname = ...определяется конфигурационным файлом...  
classarg = ...определяется конфигурационным файлом...  
   
import streamtypes # Специализированный программный код  
aclass = getattr(streamtypes, classname) # Извлечь из модуля

Прочие темы, связанные с проектированием   
863  
reader = factory(aclass, classarg) # Получить экземпляр aclass(classarg)  
processor(reader, ...)  
Здесь встроенная функция getattr снова используется для извлечения атрибу-  
та модуля, имя которого задано в виде строки (это все равно, что записать вы-  
ражение obj.attr, где attr – это строка). Так как этот фрагмент предполагает   
наличие у конструктора единственного аргумента, то, строго говоря, здесь не   
требуется ни функция factory, ни функция apply – мы могли бы просто создать   
экземпляр класса обращением aclass(classarg). Эти функции более полезны   
в случаях, когда количество аргументов не известно заранее, то есть когда уни-  
версальная фабричная функция способна повысить гибкость реализации.   
Прочие темы, связанные с проектированием  
В этой главе мы поближе познакомились с наследованием, композицией, деле-  
гированием, множественным наследованием, связанными методами и фабри-  
ками – типичными шаблонами проектирования, которые используются в ком-  
бинации с классами при создании программ на языке Python. В действитель-  
ности мы лишь слегка соприкоснулись с областью шаблонов проектирования.   
В книге обсуждаются и другие темы, связанные с проектированием, например:  
 •  
Абстрактные суперклассы (глава 28)  
 •  
Декораторы (главы 31 и 38)  
 •  
Подклассы встроенных типов (глава31)  
 •  
Статические методы и методы классов (глава 31)  
 •  
Управляемые атрибуты (глава 37)  
 •  
Метаклассы (главы 31 и 39)  
За дополнительной информацией по этой теме обращайтесь к книгам, которые   
посвящены вопросам ООП и шаблонам проектирования. Шаблоны проектиро-  
вания занимают важное положение в ООП и зачастую их реализация в языке   
Python выглядит более естественной, чем в других языках программирования.   
Тем не менее они не являются характерными только для языка Python.  
В заключение  
В этой главе мы рассмотрели подборку типичных способов комбинирования   
классов для получения наибольшей пользы от их повторного использования   
и возможности разбивать крупные задачи на более мелкие части, что обычно   
относится к проблемам проектирования, которые часто рассматриваются вне   
зависимости от конкретного языка программирования (хотя язык Python спо-  
собен облегчить их решение). Мы изучили приемы делегирования (обертыва-  
ние объектов в классы-обертки), композиции (управление встраиваемыми объ-  
ектами), наследования (приобретение поведения от других классов) и некото-  
рые другие не совсем обычные концепции, такие как псевдочастные атрибуты,   
множественное наследование, связанные методы и фабрики.  
Следующая глава завершает изучение классов и ООП рассмотрением более   
сложных тем, связанных с классами. Часть этого материала может быть более   
интересна для тех, кто пишет инструментальные средства, а не прикладные

864   
Глава 30. Шаблоны проектирования с классами   
программы, но эти сведения все же заслуживают того, чтобы с ними ознакоми-  
лись большинство тех, кто занимается ООП на языке Python. Однако сначала   
ответьте на контрольные вопросы.  
Закрепление пройденного  
Контрольные вопросы  
1. Что такое множественное наследование?  
2. Что такое делегирование?  
3. Что такое композиция?  
4. Что такое связанные методы?  
5. Для чего используются псевдочастные атрибуты?  
Ответы  
1. Множественное наследование имеет место, когда класс наследует более   
одного суперкласса, – это удобно для объединения пакетов программного   
кода, оформленных в виде классов. Порядок поиска атрибутов определяет-  
ся порядком следования суперклассов в заголовке инструкции class.  
2. Делегирование подразумевает обертывание объекта классом-оберткой,   
который расширяет функциональные возможности обернутого объекта   
и передает ему выполнение части операций. Класс-обертка сохраняет ин-  
терфейс обернутого объекта.  
3. Композиция – это прием, который подразумевает наличие контроллера,   
куда встраиваются и которым управляются несколько объектов. Класс   
контроллера предоставляет все интерфейсы как свои собственные – это   
один из способов создания крупных структур с помощью классов.  
4. Связанные методы объединяют экземпляр класса и функцию метода – их   
можно вызывать, не передавая объект экземпляра в первом аргументе, по-  
тому что внутри таких методов по-прежнему доступен оригинальный эк-  
земпляр.  
5. Псевдочастные атрибуты (имена, которых начинаются с двух символов под-  
черкивания: \_\_X) используются с целью придать именам черты локальных   
имен для вмещающего класса. Сюда относятся атрибуты класса, такие как   
методы, определенные внутри класса, и атрибуты экземпляра self, которым   
присваиваются значения внутри класса. Такие имена дополняются именем   
класса, что придает им дополнительную уникальность.

Глава 31.  
   
Дополнительные возможности классов  
Эта глава завершает наше изучение ООП на языке Python представлением не-  
скольких более сложных тем, связанных с использованием классов: мы рас-  
смотрим возможность создания подклассов встроенных типов, дополнения   
и расширения в классах «нового стиля», статические методы и методы классов,   
декораторы функций и многое другое.  
Как мы уже знаем, модель ООП в языке Python чрезвычайно проста, а неко-  
торые из приемов, представленных в этой главе, настолько сложные и совер-  
шенно не обязательные к использованию, что вы, возможно, не слишком ча-  
сто будете встречать их в своей карьере прикладного программиста на языке   
Python. Тем не менее, в интересах законченности обсуждения, мы завершим   
наше обсуждение классов кратким обзором этих возможностей.  
Как обычно, т.к. это последняя глава в этой части, она завершается сводкой ти-  
пичных проблем, с которыми сталкиваются программисты при использовании   
классов, и набором упражнений к этой части. Я советую вам обязательно про-  
работать эти упражнения, чтобы прочнее ухватить идеи, которые мы изучали   
в этой части. Я также предлагаю вам самостоятельно познакомиться с круп-  
ными объектно-ориентированными проектами на языке Python в дополнение   
к этой книге. Как и все в программировании, преимущества ООП становятся   
более очевидными с обретением опыта.  
Примечание к содержимому главы: В этой главе обсуждаются до-  
статочно сложные особенности использования классов, при этом   
некоторые из них даже слишком сложны, чтобы их можно было   
полно охватить в одной главе. Такие особенности, как свойства,   
дескрипторы, декораторы и метаклассы, здесь упоминаются   
лишь кратко и более полно будут рассматриваться в заключи-  
тельной части книги. Непременно загляните в эту часть книги,   
где вы найдете более полные примеры и более полное описание   
некоторых особенностей, подпадающих под тему этой главы.

866   
Глава 31. Дополнительные возможности классов   
Расширение встроенных типов  
Помимо реализации объектов новых типов, классы иногда используются для   
расширения функциональных возможностей встроенных типов языка Python,   
с целью обеспечения поддержки более экзотических структур данных. Напри-  
мер, чтобы добавить в списки дополнительные методы вставки и удаления,   
можно создать класс, который обертывает (встраивает) объект списка и экс-  
портирует методы вставки и удаления, которые особым образом обрабатывают   
список, подобно тому, как реализуется прием делегирования, рассмотренный   
в главе 30. Начиная с версии Python 2.2, для специализации встроенных типов   
можно также использовать наследование. Следующие два раздела демонстри-  
руют оба приема в действии.  
Расширение типов встраиванием  
Помните те функции для работы со множествами, которые мы написали в гла-  
вах 16 и 18? Ниже показано, как они выглядят, реанимированные в виде клас-  
са на языке Python. Следующий пример (файл setwrapper.py) реализует новый   
тип объектов за счет перемещения нескольких функций в методы и добавле-  
ния перегрузки нескольких основных операторов. По большей части этот класс   
просто обертывает список, добавляя дополнительные операции. Поскольку это   
класс, он также поддерживает возможность создания множества экземпляров   
и адаптацию своего поведения наследованием в подклассах. Использование   
классов вместо функций позволяет создавать множество независимых объ-  
ектов с предопределенными наборами данных и поведением, а не передавать   
списки в функции вручную:  
class Set:  
 def \_\_init\_\_(self, value = []): # Конструктор  
 self.data = [] # Управляет списком  
 self.concat(value)  
   
 def intersect(self, other): # other – любая последовательность  
 res = [] # self – подразумеваемый объект  
 for x in self.data:  
 if x in other: # Выбрать общие элементы  
 res.append(x)  
 return Set(res) # Вернуть новый экземпляр Set  
   
 def union(self, other): # other – любая последовательность  
 res = self.data[:] # Копировать список  
 for x in other: # Добавить элементы из other  
 if not x in res:  
 res.append(x)  
 return Set(res)  
   
 def concat(self, value): # Аргумент value: список, Set...  
 for x in value: # Удалить дубликаты  
 if not x in self.data:  
 self.data.append(x)  
   
 def \_\_len\_\_(self): return len(self.data) # len(self)  
 def \_\_getitem\_\_(self, key): return self.data[key] # self[i]  
 def \_\_and\_\_(self, other): return self.intersect(other) # self & other  
 def \_\_or\_\_(self, other): return self.union(other) # self | other  
 def \_\_repr\_\_(self): return ‘Set:’ + repr(self.data) # Вывод

Расширение встроенных типов   
867  
Используется этот класс как обычно – после создания экземпляра мы можем   
вызывать его методы и выполнять поддерживаемые операции:  
x = Set([1, 3, 5, 7])  
print(x.union(Set([1, 4, 7]))) # Выведет: Set:[1, 3, 5, 7, 4]  
print(x | Set([1, 4, 6])) # Выведет: Set:[1, 3, 5, 7, 4, 6]  
Перегрузка операции доступа к элементам по их индексам позволяет экзем-  
плярам нашего класса Set выглядеть как настоящие списки. В упражнениях   
в конце этой главы вам будет предложено организовать взаимодействие с этим   
классом и расширить его, поэтому подробнее об этом фрагменте мы поговорим   
в приложении B.   
Расширение типов наследованием  
Начиная с версии Python 2.2 все встроенные типы данных можно наследовать.   
Функции преобразования типов, такие как list, str, dict и tuple, превратились   
в имена встроенных типов. Теперь вызов функции преобразования типа (на-  
пример, list(‘spam’)) в действительности является вызовом конструктора типа.  
Это изменение позволяет адаптировать или расширять поведение встроенных   
типов с помощью инструкций class: достаточно просто создать подклассы с но-  
выми именами типов, где реализовать необходимые изменения. Экземпляры   
такого нового подкласса могут использоваться везде, где допускается исполь-  
зовать оригинальный встроенный тип. Например, предположим, вас не устра-  
ивает тот факт, что стандартные списки начинают отсчет элементов с 0, а не с 1.   
Это не проблема – вы всегда можете создать свой подкласс, который изменит   
эту характерную особенность списков. В файле typesubclass.py показано, как   
это делается:  
# Подкласс встроенного типа/класса list.  
# Отображает диапазон 1..N на 0..N-1; вызывает встроенную версию.  
   
class MyList(list):  
 def \_\_getitem\_\_(self, offset):  
 print(‘(indexing %s at %s)’ % (self, offset))  
 return list.\_\_getitem\_\_(self, offset - 1)  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 print(list(‘abc’))  
 x = MyList(‘abc’) # \_\_init\_\_ наследуется из списка  
 print(x) # \_\_repr\_\_ наследуется из списка  
   
 print(x[1]) # MyList.\_\_getitem\_\_  
 print(x[3]) # Изменяет поведение метода суперкласса  
   
 x.append(‘spam’); print(x) # Атрибуты, унаследованные от суперкласса list  
 x.reverse(); print(x)  
В этом файле подкласс MyList расширяет метод \_\_getitem\_\_ встроенных списков   
простым отображением диапазона значений от 1 до N на необходимый спискам   
диапазон от 0 до N-1. Уменьшение индекса на единицу и вызов версии метода   
из суперкласса – вот все, что в действительности делается, но этого вполне до-  
статочно для достижения поставленной цели:  
% python typesubclass.py  
[‘a’, ‘b’, ‘c’]  
[‘a’, ‘b’, ‘c’]

868   
Глава 31. Дополнительные возможности классов   
(indexing [‘a’, ‘b’, ‘c’] at 1)  
a  
(indexing [‘a’, ‘b’, ‘c’] at 3)  
c  
[‘a’, ‘b’, ‘c’, ‘spam’]  
[‘spam’, ‘c’, ‘b’, ‘a’]  
В эти результаты включен текст, который выводит метод класса при выпол-  
нении индексирования. Является ли изменение способа индексирования уни-  
версально хорошей идеей, это уже другой вопрос: пользователи вашего класса   
MyList могут быть повергнуты в недоумение таким отступлением от общепри-  
нятого поведения последовательностей в языке Python. Однако возможность   
адаптировать встроенные типы подобным образом может оказаться мощным   
инструментом.  
Например, такой шаблон порождает альтернативный способ реализации   
множества – в виде подкласса встроенного списка, а не в виде самостоятель-  
ного класса, который управляет встроенным в него объектом списка. Как мы   
узнали в главе 5, в настоящее время язык Python не только обладает мощным   
встроенным объектом множества, но и позволяет определять новые множества   
с использованием синтаксиса литералов и генераторов множеств. Тем не менее   
попытка реализовать собственный класс множеств является отличным спосо-  
бом изучить особенности наследования типов вообще.  
Следующий пример реализации класса в файле setsubclass.py адаптирует спи-  
ски, добавляя методы и операторы, используемые для работы с множествами.   
Все остальное поведение наследуется от встроенного суперкласса list, поэтому   
альтернатива получилась более короткой и простой:  
class Set(list):  
 def \_\_init\_\_(self, value = []): # Конструктор  
 list.\_\_init\_\_([]) # Адаптирует список  
 self.concat(value) # Копировать изменяемый аргумент по умолчанию  
   
 def intersect(self, other): # other – любая последовательность  
 res = [] # self – подразумеваемый объект  
 for x in self:  
 if x in other: # Выбрать общие элементы  
 res.append(x)  
 return Set(res) # Вернуть новый экземпляр Set  
   
 def union(self, other): # other – любая последовательность  
 res = Set(self) # Копировать меня и мой список  
 res.concat(other)  
 return res  
   
 def concat(self, value): # аргумент value: list, Set...  
 for x in value: # Удалить дубликаты  
 if not x in self:  
 self.append(x)  
   
 def \_\_and\_\_(self, other): return self.intersect(other)  
 def \_\_or\_\_(self, other): return self.union(other)  
 def \_\_repr\_\_(self): return ‘Set:’ + list.\_\_repr\_\_(self)  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 x = Set([1,3,5,7])  
 y = Set([2,1,4,5,6])

Классы «нового стиля»   
869  
 print(x, y, len(x))  
 print(x.intersect(y), y.union(x))  
 print(x & y, x | y)  
 x.reverse(); print(x)  
Ниже приводится вывод, полученный в результате выполнения кода само-  
проверки, находящегося в конце файла. Поскольку проблема наследования   
встроенных типов достаточно сложна, я опущу дальнейшие подробности, но   
предлагаю внимательно посмотреть на полученные результаты, чтобы изучить   
поведение подкласса:  
% python setsubclass.py  
Set:[1, 3, 5, 7] Set:[2, 1, 4, 5, 6] 4  
Set:[1, 5] Set:[2, 1, 4, 5, 6, 3, 7]  
Set:[1, 5] Set:[1, 3, 5, 7, 2, 4, 6]  
Set:[7, 5, 3, 1]  
Существуют более эффективные способы реализации множеств – с помощью   
словарей, которые позволяют заменить последовательное сканирование, ис-  
пользуемое в данной реализации, на операцию обращения по ключу (хеши-  
рование) и тем самым повысить скорость работы. (За дополнительной инфор-  
мацией обращайтесь к книге «Программирование на Python».) Если вас заин-  
Python».) Если вас заин-  
».) Если вас заин-  
тересовали множества, тогда вам также стоит взглянуть на тип объектов set,   
который рассматривался в главе 5, – этот встроенный тип реализует большое   
количество разнообразных операций над множествами. Реализация операций   
над множествами прекрасно подходит для нужд обучения, но создавать такие   
реализации в современных версиях Python больше не требуется.  
В качестве другого примера наследования можно привести новый тип bool,   
появившийся в Python 2.3. Как упоминалось ранее в этой книге, bool – это   
подкласс типа int с двумя экземплярами (True и False), которые ведут себя как   
целые числа 1 и 0, но наследуют измененные версии методов вывода, особым   
образом отображающие их имена.  
Классы «нового стиля»  
В версии Python 2.2 появилась новая разновидность классов, известная как   
классы «нового стиля». Классы, следующие оригинальной модели, называют   
«классическими классами», когда сравнивают их с новой разновидностью.   
В версии 3.0 осталась только одна разновидность классов, но для пользовате-  
лей Python 2.X классы по-прежнему делятся на две категории:  
 •  
В Python 3.0 все классы автоматически относятся к категории классов «но-  
вого стиля», независимо от того, наследуют ли они явно класс object или   
нет. Все классы наследуют object, явно или неявно, и все объекты являются   
экземплярами класса object.  
 •  
В Python 2.6 и в более ранних версиях классы должны явно наследовать   
класс object (или другой встроенный тип), чтобы считаться классами «ново-  
го стиля» и получить в свое распоряжение все особенности классов нового   
стиля.  
Поскольку в Python 3.0 все классы автоматически считаются классами ново-  
Python 3.0 все классы автоматически считаются классами ново-  
 3.0 все классы автоматически считаются классами ново-  
го стиля, особенности классов нового стиля стали обычными особенностями   
классов. Однако из уважения к пользователям Python 2.X я решил дать их

870   
Глава 31. Дополнительные возможности классов   
описание в этом разделе отдельно – классы в этой версии приобретают черты   
классов нового стиля, только если они явно наследуют класс object.  
Другими словами, когда пользователи Python 3.0 увидят в этом разделе опи-  
Python 3.0 увидят в этом разделе опи-  
 3.0 увидят в этом разделе опи-  
сание особенностей классов «нового стиля», они должны считать его описани-  
ем существующих особенностей классов. Для пользователей Python 2.6 такие   
описания являются описанием дополнительных усовершенствований.  
В Python 2.6 и в более ранних версиях единственное синтаксическое отличие   
классов нового стиля состоит в том, что они наследуют либо встроенный тип,   
такой как list, либо специальный встроенный класс object. Встроенный класс   
object играет роль суперкласса для классов нового стиля, когда ни один другой   
встроенный тип не подходит на эту роль:  
class newstyle(object):  
 ...обычный программный код...  
Любые классы, наследующие класс object или любой другой встроенный тип,   
автоматически интерпретируются как классы нового стиля. Если где-то в де-  
реве наследования класса присутствует какой-нибудь встроенный тип, этот   
класс будет считаться классом нового стиля. Классы, не наследующие встроен-  
ный класс, такой как object, считаются классическими.  
Классы нового стиля лишь немного отличаются от классических классов, и эти   
отличия совершенно незаметны для подавляющего большинства пользовате-  
лей Python. Кроме того, классическая модель классов, доступная в Python 2.6   
и используемая на протяжении последних двух десятилетий, по-прежнему ра-  
ботает именно так, как было описано выше.  
Классы нового стиля практически сохраняют обратную совместимость с клас-  
сическими классами в синтаксисе и в поведении – они привносят лишь не-  
сколько новых особенностей. Однако, т. к. они некоторым образом изменили   
поведение классов, их следует представлять как отдельный инструмент, чтобы   
избежать нежелательных воздействий на любой существующий программный   
код, работоспособность которого зависит от прежнего поведения классов. На-  
пример, некоторые малозаметные отличия, такие как ромбоидальная схема   
поиска в дереве наследования и особенности выполнения встроенных опера-  
ций над атрибутами, доступ к которым контролируется методами, такими как   
\_\_getattr\_\_, могут привести к нарушениям в работе устаревшего программного   
кода, если не внести в него соответствующие изменения.  
В следующих двух разделах приводится краткий обзор основных отличий   
классов нового стиля и их новых возможностей. Отмечу еще раз, так как на   
сегодняшний день все классы являются классами нового стиля, пользователи   
Python 2.X могут считать эти разделы описанием изменений в языке Python,   
а пользователи Python 3.0 – описанием дополнительных возможностей классов.  
Изменения в классах нового стиля  
Классы нового стиля имеют отличия от классических классов в различных ка-  
тегориях� часть этих отличий малозаметна, но может оказывать воздействие   
и на существующий программный код, написанный для работы под управле-  
нием Python 2.X, и на стиль программирования. Ниже перечислены наиболее   
заметные отличия:

Изменения в классах нового стиля   
871  
Классы и типы были объединены  
Классы теперь являются типами, а типы – классами. Фактически эти два   
термина стали синонимами. Вызов встроенной функции type(I) теперь воз-  
вращает класс, из которого был получен экземпляр� обычно тот, что указан   
в атрибуте I.\_\_class\_\_, а не обобщенный тип «instance». Кроме того, сами   
классы являются экземплярами класса type, который можно наследовать   
в подклассах для изменения процедуры создания классов, и все классы (а,   
следовательно, и типы) наследуют класс object.  
Порядок поиска в дереве наследования  
Принятая ромбоидальная схема множественного наследования немного из-  
менила порядок поиска – грубо говоря, в этой схеме поиск сначала произво-  
дится в ширину и только потом в высоту.   
Извлечение атрибутов встроенными операциями  
Встроенные операции больше не используют методы \_\_getattr\_\_ и \_\_getat-  
getat-  
tribute\_\_ для неявного извлечения атрибутов. Это означает, что данные ме-  
тоды не вызываются для получения ссылок на методы перегрузки опера-  
торов с именами вида \_\_X\_\_ – поиск таких имен начинается с класса, а не   
с экземпляра.  
Новые особенности  
Классы нового стиля приобрели ряд новых особенностей, включая слоты,   
свойства, дескрипторы и новый метод \_\_getattribute\_\_. В большинстве сво-  
ем они предназначены для использования разработчиками, создающими   
инструментальные средства.  
Третий пункт из этого списка мы уже обсуждали во врезке в главе 27 и еще   
будем подробно рассматривать в главе 37, когда будем знакомиться с возмож-  
ностью управления атрибутами, и в главе 38, при обсуждении декораторов   
частных атрибутов. Однако, так как первое и второе изменения из этого списка   
могут нарушить работоспособность существующего программного кода, напи-  
санного для работы под управлением Python 2.X, мы исследуем их более под-  
X, мы исследуем их более под-  
, мы исследуем их более под-  
робно, прежде чем перейдем к дополнениям нового стиля.  
Изменения в модели типов  
С принятием модели классов нового стиля различия между типами и классами   
полностью исчезли. Теперь классы также являются типами: сами классы яв-  
ляются экземплярами класса type, а типами экземпляров классов являются их   
классы. Фактически между встроенными типами, такими как списки и стро-  
ки, и пользовательскими типами, определенными в виде классов, нет никаких   
отличий. Именно поэтому имеется возможность наследовать встроенные типы   
в своих классах, как было показано выше в этой главе, – поскольку подкласс,   
наследующий встроенный тип, такой как list, квалифицируется, как класс   
нового стиля, он становится пользовательским типом данных.  
Кроме появившейся возможности наследовать встроенные типы, это измене-  
ние становится наиболее очевидным, когда появляется необходимость явно   
выполнить проверку типа. В модели классических классов, в Python 2.6, эк-  
Python 2.6, эк-  
 2.6, эк-  
земпляры классов автоматически относятся к обобщенному типу «instance»,   
тогда как встроенные объекты имеют более определенные типы:

872   
Глава 31. Дополнительные возможности классов   
C:\misc> c:\python26\python  
>>> class C: pass # Классические классы в 2.6  
...  
>>> I = C()  
>>> type(I) # Экземпляры классов  
<type ‘instance’>  
>>> I.\_\_class\_\_  
<class \_\_main\_\_.C at 0x025085A0>  
   
>>> type(C) # Но классы не являются типами  
<type ‘classobj’>  
>>> C.\_\_class\_\_  
AttributeError: class C has no attribute ‘\_\_class\_\_’  
   
>>> type([1, 2, 3])  
<type ‘list’>  
>>> type(list)  
<type ‘type’>  
>>> list.\_\_class\_\_  
<type ‘type’>  
С появлением классов нового стиля типом экземпляра класса является сам   
класс, из которого он был создан, то есть классы являются обычными пользо-  
вательскими типами данных, – типом экземпляра является его класс, а поль-  
зовательские классы имеют тот же тип, что и встроенные объекты типов. Те-  
перь классы тоже имеют атрибут \_\_class\_\_, потому что они являются экзем-  
плярами класса type:  
C:\misc> c:\python26\python  
>>> class C(object): pass # Классы нового стиля в 2.6  
...  
>>> I = C()  
>>> type(I) # Типом экземпляра является его класс  
<class ‘\_\_main\_\_.C’>  
>>> I.\_\_class\_\_  
<class ‘\_\_main\_\_.C’>  
   
>>> type(C) # Классы являются пользовательскими типами данных  
<type ‘type’>  
>>> C.\_\_class\_\_  
<type ‘type’>  
   
>>> type([1, 2, 3]) # Для встроенных типов ничего не изменилось  
<type ‘list’>  
>>> type(list)  
<type ‘type’>  
>>> list.\_\_class\_\_  
<type ‘type’>  
То же самое справедливо для всех классов в Python 3.0, потому что в этой вер-  
Python 3.0, потому что в этой вер-  
 3.0, потому что в этой вер-  
сии все классы автоматически считаются классами нового стиля, даже если   
они явно не наследуют никаких суперклассов. Фактически в версии 3.0 раз-  
личия между встроенными типами и пользовательскими классами полностью   
исчезли:  
C:\misc> c:\python30\python  
>>> class C: pass # Все классы являются классами нового стиля в 3.0  
...

Изменения в классах нового стиля   
873  
>>> I = C()  
>>> type(I) # Типом экземпляра является его класс  
<class ‘\_\_main\_\_.C’>  
>>> I.\_\_class\_\_  
<class ‘\_\_main\_\_.C’>  
   
>>> type(C) # Классы – это типы, а типы – это классы  
<class ‘type’>  
>>> C.\_\_class\_\_  
<class ‘type’>  
   
>>> type([1, 2, 3]) # Классы и встроенные типы ничем не отличаются  
<class ‘list’>  
>>> type(list)  
<class ‘type’>  
>>> list.\_\_class\_\_  
<class ‘type’>  
Как видите, в Python 3.0 классы являются типами, а типы являются классами.   
С технической точки зрения каждый класс создается из метакласса – класса   
с именем type или его подкласса, адаптирующего или управляющего процеду-  
рой создания классов. Помимо того, что это изменение оказывает влияние на   
программный код, выполняющий проверку типов, оно оказывается принципи-  
альным для разработчиков инструментальных средств. Далее в этой главе мы   
познакомимся с метаклассами поближе и более детально будем рассматривать   
их в главе 39.  
Значимость для операций проверки типа  
Помимо возможности создания метаклассов и адаптированных версий встро-  
енных типов, объединение классов и типов в модель классов нового стиля мо-  
жет оказывать влияние на реализацию операций проверки типов. В Python 3.0,   
например, типы экземпляров классов можно сравнивать непосредственно, точ-  
но так же, как сравниваются встроенные объекты типов. Это обусловлено тем,   
что теперь классы являются типами, а типом экземпляра является его класс:  
C:\misc> c:\python30\python  
>>> class C: pass  
...  
>>> class D: pass  
...  
>>> c = C()  
>>> d = D()  
>>> type(c) == type(d) # 3.0: сравниваются классы экземпляров  
False  
   
>>> type(c), type(d)  
(<class ‘\_\_main\_\_.C’>, <class ‘\_\_main\_\_.D’>)  
>>> c.\_\_class\_\_, d.\_\_class\_\_  
(<class ‘\_\_main\_\_.C’>, <class ‘\_\_main\_\_.D’>)  
   
>>> c1, c2 = C(), C()  
>>> type(c1) == type(c2)  
True  
Однако, в случае с классическими классами в Python 2.6 и в более ранних   
версиях, сравнивать типы экземпляров практически бессмысленно, потому

874   
Глава 31. Дополнительные возможности классов   
что все экземпляры в этой модели имеют один и тот же тип «instance». Что-  
instance». Что-  
». Что-  
бы действительно сравнить типы экземпляров, необходимо сравнить значения   
их атрибутов \_\_class\_\_ (если вас волнует проблема переносимости, отмечу, что   
в версии 3.0 этот прием действует точно так же, хотя надобность в нем отпала):  
C:\misc> c:\python26\python  
>>> class C: pass  
...  
>>> class D: pass  
...  
>>> c = C()  
>>> d = D()  
>>> type(c) == type(d) # 2.6: все экземпляры имеют один и тот же тип  
True  
>>> c.\_\_class\_\_ == d.\_\_class\_\_ # Следует явно сравнивать классы  
False  
   
>>> type(c), type(d)  
(<type ‘instance’>, <type ‘instance’>)  
>>> c.\_\_class\_\_, d.\_\_class\_\_  
(<class \_\_main\_\_.C at 0x024585A0>, <class \_\_main\_\_.D at 0x024588D0>)  
Как и следовало ожидать, классы нового стиля в 2.6 в этом отношении действу-  
ют точно так же, как и все классы в 3.0, – при сравнивании типов экземпляров   
автоматически сравниваются их классы:  
C:\misc> c:\python26\python  
>>> class C(object): pass  
...  
>>> class D(object): pass  
...  
>>> c = C()  
>>> d = D()  
>>> type(c) == type(d) # Классы нового стиля в 2.6: действуют так же,   
False # как и в 3.0  
   
>>> type(c), type(d)  
(<class ‘\_\_main\_\_.C’>, <class ‘\_\_main\_\_.D’>)  
>>> c.\_\_class\_\_, d.\_\_class\_\_  
(<class ‘\_\_main\_\_.C’>, <class ‘\_\_main\_\_.D’>)  
Конечно, как уже неоднократно отмечалось в этой книге, проверка типа в про-  
граммах на языке Python обычно не является правильным решением (для нас   
важны интерфейсы объектов, а не их типы), а в тех редких случаях, когда дей-  
ствительно необходимо проверить тип экземпляра класса, вероятно, лучше ис-  
пользовать более универсальную встроенную функцию isinstance. Тем не ме-  
нее знакомство с моделью типов в языке Python поможет вам пролить свет на   
модель классов вообще.  
Все объекты наследуют класс «object»   
Еще одно следствие изменений в типах, вызванных появлением модели клас-  
сов нового стиля, заключается в том, что теперь все классы наследуют (явля-  
ются производными) класс object, явно или неявно. То есть все типы теперь яв-  
ляются классами, поэтому каждый объект наследует встроенный класс object,   
прямо или косвенно, через свой суперкласс. Рассмотрим следующий сеанс ра-

Изменения в классах нового стиля   
875  
боты с интерактивной оболочкой Python 3.0 (чтобы этот программный код дей-  
Python 3.0 (чтобы этот программный код дей-  
 3.0 (чтобы этот программный код дей-  
ствовал точно так же в 2.6, укажите явно класс object в списке суперклассов):  
>>> class C: pass  
...  
>>> X = C()  
   
>>> type(X) # Теперь типом является класс экземпляра  
<class ‘\_\_main\_\_.C’>  
>>> type(C)  
<class ‘type’>  
Как и прежде, типом экземпляра класса является его класс, а типом класса   
является класс type, что является следствием объединения классов и типов.   
Однако также верно, что экземпляры и классы наследуют встроенный класс   
object, поскольку он явно или неявно является суперклассом любого класса:  
>>> isinstance(X, object)  
True  
>>> isinstance(C, object) # Классы всегда наследуют класс object  
True  
То же относится и к встроенным типам, таким как списки и строки, потому что   
в модели классов нового стиля типы являются классами� теперь встроенные   
типы – это классы, и их экземпляры тоже наследуют класс object:  
>>> type(‘spam’)  
<class ‘str’>  
>>> type(str)  
<class ‘type’>  
   
>>> isinstance(‘spam’, object) # То же относится и к встроенным типам   
True # (классам)  
>>> isinstance(str, object)  
True  
Фактически сам класс type наследует класс object, а класс object наследует   
класс type, даже при том, что оба они являются совершенно различными объ-  
ектами, – циклическая связь, венчающая объектную модель и вытекающая из   
того факта, что типы являются классами, которые генерируют другие классы:  
>>> type(type) # Все классы – это типы, и наоборот  
<class ‘type’>  
>>> type(object)  
<class ‘type’>  
   
>>> isinstance(type, object) # Все классы наследуют object, даже класс type  
True  
>>> isinstance(object, type) # Типы создают классы, и type является классом  
True  
>>> type is object  
False  
С практической точки зрения эта модель создает меньше особых случаев, чем   
прежняя модель классических классов, различающая типы и классы, и это по-  
зволяет писать программный код, который предполагает наличие суперкласса   
object и использует его. Примеры такого подхода мы увидим далее в этой кни-  
ге, а пока перейдем к изучению других отличий в модели классов нового стиля.

876   
Глава 31. Дополнительные возможности классов   
Ромбоидальное наследование  
Пожалуй, самым ощутимым изменением в классах нового стиля является не-  
много отличная интерпретация наследования – так называемая ромбоидаль-  
ная схема в деревьях множественного наследования, когда более одного супер-  
класса наследуют один и тот же суперкласс более высокого уровня. Ромбои-  
дальная схема – это сложная концепция проектирования, не обсуждавшаяся   
в этой книге ранее. Она крайне редко используется на практике, поэтому мы не   
будем подробно останавливаться на этой теме.  
В двух словах, в классической модели процедура поиска в дереве наследова-  
ния сначала движется строго вверх по дереву, а потом слева направо – сначала   
интерпретатор поднимается вверх всеми возможными путями по левой сторо-  
не дерева, затем возвращается назад и начинает поиск с первого суперкласса,   
расположенного правее предыдущего. В новой модели в таких случаях поиск   
сначала производится в ширину – интерпретатор сначала просматривает все   
суперклассы, стоящие правее того, где поиск уже произведен, и только потом   
начинает подъем всеми возможными путями к общему суперклассу. Другими   
словами, поиск выполняется по уровням дерева наследования. В действитель-  
ности интерпретатор использует немного более сложный алгоритм поиска, чем   
описано здесь, но этого упрощенного представления вполне достаточно для   
большинства программистов.   
Вследствие такого изменения суперклассы, расположенные ниже, получают   
возможность переопределять атрибуты суперклассов, стоящих выше, неза-  
висимо от вида деревьев множественного наследования. Кроме того, согласно   
правилам поиска в новой модели каждый суперкласс просматривается не бо-  
лее одного раза, даже если он наследуется несколькими подклассами.  
Пример ромбоидального наследования  
В качестве иллюстрации рассмотрим следующую упрощенную реализацию   
ромбоидального наследования для классических классов. Здесь оба суперклас-  
са, B и C, которые наследуются классом D, имеют общего предка – класс A:  
>>> class A: # Классическая модель (Python 2.6)  
 attr = 1  
   
>>> class B(A): # B и C имеют общего предка – A  
 pass  
   
>>> class C(A):   
 attr = 2  
   
>>> class D(B,C): # Сначала поиск дойдет до A, потом до C  
 pass  
   
>>> x = D()  
>>> x.attr # Порядок поиска: x,D,B,A  
1  
В этом случае атрибут attr будет найден в суперклассе A, потому что в клас-  
сической модели поиск в дереве наследования сначала производится в высоту,   
и только потом происходит смещение вправо – интерпретатор будет выполнять   
поиск в следующем порядке: D, B, A и затем C, впрочем, поиск прекратится, как   
только атрибут attr будет найден в суперклассе A, расположенном выше супер-  
класса B.

Изменения в классах нового стиля   
877  
В классах нового стиля, наследующих встроенный тип, такой как object, и во   
всех классах в Python 3.0 поиск будет выполняться в другом порядке: прежде   
чем просмотреть суперкласс A, интерпретатор сначала выполнит поиск в супер-  
классе C (правее суперкласса B), то есть в следующем порядке: D, B, C и затем A,   
но в этом случае поиск остановится в суперклассе C:  
>>> class A(object): # Новый стиль  
 attr = 1  
   
>>> class B(A):   
 pass  
   
>>> class C(A):   
 attr = 2  
   
>>> class D(B,C): # Сначала поиск дойдет до C, потом до A  
 pass  
   
>>> x = D() # Порядок поиска: x,D,B,С  
>>> x.attr  
2  
Это изменение процедуры поиска основано на предположении, что если вы до-  
бавляете класс C в дерево ниже, это значит, что вы хотите получить его атри-  
буты раньше, чем атрибуты класса A. Кроме того, это изменение предполагает,   
что класс C всегда будет иметь возможность переопределить атрибуты класса A,   
что, скорее всего, верно, когда пишется самостоятельный класс, но совсем не-  
верно, когда в ромбоидальной схеме принимают участие классические клас-  
сы, – вы можете даже не подозревать, что класс C может участвовать в подобной   
схеме наследования, когда пишете его.  
Поскольку в подобной ситуации наиболее вероятно, что программист под-  
разумевал, что класс C будет переопределять атрибуты класса A, новая модель   
гарантирует, что класс C будет просматриваться первым. В противном случае   
класс C мог бы оказаться практически бесполезным при использовании ром-  
боидальной схемы наследования: он был бы лишен возможности адаптировать   
класс A и мог бы использоваться только для экспортирования имен, присут-  
ствующих только в классе C.  
Явное разрешение конфликтов имен  
Проблема с предположениями в том, что они всего лишь предположения. Если   
такое отклонение в процедуре поиска кажется вам слишком трудным для за-  
поминания, или вам требуется более полное управление процедурой поиска,   
вы всегда можете произвести выбор желаемого атрибута из любого места в де-  
реве, выполнив присваивание или как-то иначе обозначив его там, где может   
возникнуть смешение классов:  
>>> class A: # Классическая модель  
 attr = 1  
   
>>> class B(A):   
 pass  
   
>>> class C(A):   
 attr = 2

878   
Глава 31. Дополнительные возможности классов   
>>> class D(B,C): # Выбрать C, справа  
 attr = C.attr  
   
>>> x = D()  
>>> x.attr # Работает как класс нового стиля   
2 # (все классы в 3.0)  
Здесь дерево классических классов имитирует порядок поиска, принятый в мо-  
дели классов нового стиля: присваивание атрибуту attr в классе D явно выбира-  
ет версию атрибута из класса C, благодаря чему нарушается обычный порядок   
поиска в дереве наследования (атрибут D.attr находится ниже в дереве). Точно   
так же классы нового стиля могут имитировать порядок поиска в классической   
модели, выбирая требуемый атрибут там, где может происходить смешение:  
>>> class A(object): # Новый стиль  
 attr = 1  
   
>>> class B(A):   
 pass   
   
>>> class C(A):   
 attr = 2  
   
>>> class D(B,C): # Выбрать A.attr, выше  
 attr = B.attr  
   
>>> x = D()  
>>> x.attr # Работает как классический класс   
1 # (по умолчанию в 2.6)  
Если вам необходимо всегда явно разрешать конфликты, подобные этим, вы   
можете просто игнорировать различия в порядке поиска и не полагаться на   
предположения о том, что имеется в виду, когда вы пишете свои классы.   
Естественно, такой способ выбора атрибутов может также применяться и к ме-  
тодам, потому что методы – это обычные объекты:  
>>> class A:  
 def meth(s): print(‘A.meth’)  
   
>>> class C(A):  
 def meth(s): print(‘C.meth’)  
   
>>> class B(A):  
 pass  
   
>>> class D(B,C): pass # Использовать порядок поиска по умолчанию  
>>> x = D() # Зависит от типа класса  
>>> x.meth() # По умолчанию – классический порядок поиска в 2.6  
A.meth  
   
>>> class D(B,C): meth = C.meth # Выбрать метод класса C: новый стиль (и 3.0)  
>>> x = D()  
>>> x.meth()  
C.meth  
   
>>> class D(B,C): meth = B.meth # Выбрать метод класса B: классическая модель  
>>> x = D()

Изменения в классах нового стиля   
879  
>>> x.meth()  
A.meth  
Здесь мы явно выбираем методы, выполняя присваивание именам, находя-  
щимся ниже в дереве. Мы могли бы просто вызвать метод желаемого класса   
явно – на практике этот подход, возможно, является более общепринятым,   
в особенности при работе с конструкторами:  
class D(B,C):  
 def meth(self): # Переопределяется ниже  
 ...  
 C.meth(self) # Вызовом выбрать метод класса C  
Такой выбор путем присваивания или вызова в точках смешения может эф-  
фективно обезопасить ваш программный код от возможных различий между   
разными моделями классов. Явное разрешение конфликтов таким способом   
гарантирует, что правильная работа вашего программного кода не будет зави-  
сеть от версии Python в будущем (независимо от необходимости наследовать   
класс object или встроенные типы, чтобы использовать новую модель).  
Даже если не учитывать расхождения между классической   
и новой моделями, данная методика иногда может пригодиться   
в случаях множественного наследования. Например, если вам   
необходимо получить часть атрибутов от суперкласса слева,   
а часть – от суперкласса справа, вам может потребоваться ука-  
зать интерпретатору Python, какие именно атрибуты следует   
выбирать, выполняя явное присваивание в подклассах. Мы еще   
вернемся к этому вопросу в разделе с описанием типичных про-  
блем в конце этой главы.   
Кроме того, обратите внимание, что ромбоидальные схемы на-  
следования в некоторых случаях могут доставлять еще больше   
хлопот, чем я описал здесь (например, что если оба конструктора   
классов B и C вызывают конструктор класса A, и при этом необ-  
ходимо вызывать оба наследуемых конструктора в подклассе?).   
Поскольку в практике такие ситуации встречаются крайне ред-  
ко, мы оставим эту тему за рамками данной книги (однако об-  
ратите внимание на встроенную функцию super – она не только   
обобщает доступ к суперклассам в деревьях с простым наследо-  
ванием, но и поддерживает кооперативный режим разрешения   
некоторых конфликтов, возникающих в деревьях множествен-  
ного наследования).  
Пределы влияния изменений в порядке поиска  
Итак, поиск в ромбоидальной схеме наследования выполняется по-разному   
в классической и в новой моделях, и это изменение нарушает обратную совме-  
стимость. Однако имейте в виду, что это изменение затрагивает только ромбои-  
дальные схемы множественного наследования – во всех других схемах прин-  
цип действия модели наследования нового стиля не изменился. Кроме того,   
вполне возможно, что вся эта проблема будет носить скорее теоретический ха-  
рактер, чем практический, – с выходом версии Python 2.2 эти изменения не   
оказали достаточно существенного влияния и не приобрели масштабного зна-

880   
Глава 31. Дополнительные возможности классов   
чения к появлению Python 3.0, потому маловероятно, что они затронут значи-  
тельную часть программного кода на языке Python.  
Кроме того, следует отметить, что даже если вы не будете применять ромбо-  
идальную схему наследования в своих классах, все равно суперкласс object   
в версии 3.0 всегда будет находиться выше любого класса, вследствие чего лю-  
бой случай множественного наследования будет соответствовать ромбоидаль-  
ной схеме. То есть в новой модели класс object автоматически играет ту же са-  
мую роль, какую играет класс A в только что рассмотренном примере. Отсюда   
следует, что новые правила поиска не только изменили логическую семантику,   
но и оптимизировали производительность, так как исключают возможность   
посещения одного и того же класса более чем один раз.  
Не менее важно то обстоятельство, что суперкласс object в новой модели предо-  
ставляет методы по умолчанию, реализующие различные встроенные опера-  
ции, включая методы вывода \_\_str\_\_ и \_\_repr\_\_. Вызовите функцию dir(object),   
чтобы увидеть, какие методы им предоставляются. Без изменений в правилах   
поиска при множественном наследовании методы по умолчанию в классе object   
всегда имели бы преимущество перед переопределенными версиями в пользо-  
вательских классах, если только переопределенные версии не находились бы   
в самом первом унаследованном суперклассе. Другими словами, новая модель   
сама делает новый порядок поиска более важным!  
Более наглядные примеры постоянного присутствия суперкласса object в вер-  
сии 3.0 и другие примеры ромбоидальной схемы наследования, создавае-  
мой им, вы найдете в результатах работы класса ListTree, в примере lister.py,   
в преды дущей главе, а также в примере classtree.py, в главе 28, где реализован   
обход дерева классов.  
Другие расширения в классах нового стиля  
Помимо изменений, описанных в предыдущем разделе (которые, честно при-  
знаться, имеют скорее академический интерес и могут не иметь большого зна-  
чения для большинства читателей этой книги), классы нового стиля предлага-  
ют некоторый набор расширенных возможностей, которые имеют более явное   
практическое применение. Ниже приводится краткий обзор каждой из таких   
особенностей, присущих классам нового стиля в Python 2.6 и всем классам   
в Python 3.0.  
Слоты экземпляров  
Присваивая список имен атрибутов в виде строк специальному атрибуту \_\_  
slots\_\_ класса, в классах нового стиля можно ограничить множество разре-  
шенных атрибутов для экземпляров класса и оптимизировать использование   
памяти и производительность.   
Обычно этот атрибут устанавливается присваиванием последовательности   
имен строк переменной \_\_slots\_\_ на верхнем уровне в инструкции class: только   
имена, перечисленные в списке \_\_slots\_\_, смогут использоваться как атрибуты   
экземпляра. Однако, как и в случае с любыми именами в языке Python, пре-  
жде чем получить доступ к атрибутам экземпляра, им должны быть присвоены   
значения, даже если они перечислены в списке \_\_slots\_\_. Например:

Другие расширения в классах нового стиля   
881  
>>> class limiter(object):  
... \_\_slots\_\_ = [‘age’, ‘name’, ‘job’]  
...  
>>> x = limiter()  
>>> x.age # Присваивание должно быть выполнено раньше использования  
AttributeError: age  
   
>>> x.age = 40  
>>> x.age  
40  
>>> x.ape = 1000 # Недопустимое имя: отсутствует в списке \_\_slots\_\_  
AttributeError: ‘limiter’ object has no attribute ‘ape’  
(AttributeError: объект ‘limiter’ не имеет атрибута ‘ape’)  
Слоты – это своего рода нарушение динамической природы языка Python,   
которая диктует, что операция присваивания может создавать любые имена.   
Однако предполагается, что эта особенность поможет ликвидировать ошибки,   
обусловленные простыми «опечатками» (обнаруживается попытка присваи-  
вания атрибутам, отсутствующим в списке \_\_slots\_\_), и обеспечит некоторую   
оптимизацию. Выделение памяти для словаря с именами атрибутов в каждом   
экземпляре может оказаться слишком дорогим удовольствием, когда требу-  
ется создать большое количество экземпляров, каждый из которых обладает   
небольшим числом атрибутов. Для экономии пространства в памяти и повы-  
шения производительности (получающийся выигрыш в значительной степени   
зависит от самой программы) атрибуты, перечисленные в слотах, сохраняются   
не в словаре, а в виде последовательности, что обеспечивает более высокую ско-  
рость их поиска.  
Слоты и обобщенные инструменты  
Фактически некоторые экземпляры со слотами вообще могут не иметь атри-  
бут словаря \_\_dict\_\_, что может сделать некоторые метапрограммы намного   
более сложными (включая некоторые из тех, что представлены в этой книге).   
Обобщенные инструменты, которые получают списки атрибутов или обраща-  
ются к атрибутам, используя имена в виде строк, например, должны исполь-  
зовать более универсальные механизмы, чем атрибут \_\_dict\_\_. К таким меха-  
низмам можно отнести встроенные функции getattr, setattr и dir, способные   
отыскивать атрибуты в обоих хранилищах, \_\_dict\_\_ и \_\_slots\_\_. В некоторых   
случаях для полноты картины может потребоваться проверить оба источника   
атрибутов.  
Например, экземпляры классов, где используются слоты, обычно не имеют   
атрибут словаря \_\_dict\_\_ – вместо него пространство для атрибутов в экзем-  
пляре выделяется с применением дескрипторов класса, которые будут рассма-  
триваться в главе 37. Только имена, перечисленные в списке \_\_slots\_\_, смогут   
использоваться как атрибуты экземпляра, однако значения этих атрибутов   
могут извлекаться и изменяться обычными способами. В Python 3.0 (и в 2.6,   
в случае классов, наследующих object):  
>>> class C:  
... \_\_slots\_\_ = [‘a’, ‘b’] # По умолчанию наличие \_\_slots\_\_ означает   
... # отсутствие \_\_dict\_\_  
>>> X = C()  
>>> X.a = 1

882   
Глава 31. Дополнительные возможности классов   
>>> X.a  
1  
>>> X.\_\_dict\_\_  
AttributeError: ‘C’ object has no attribute ‘\_\_dict\_\_’  
>>> getattr(X, ‘a’)  
1  
>>> setattr(X, ‘b’, 2) # Однако функции getattr() и setattr()   
>>> X.b # по-прежнему работают  
2  
>>> ‘a’ in dir(X) # И dir() также отыскивает атрибуты в слотах  
True  
>>> ‘b’ in dir(X)  
True  
В отсутствие словаря с пространством имен невозможно присвоить значения   
атрибутам экземпляра, имена которых отсутствуют в списке слотов:  
>>> class D:  
... \_\_slots\_\_ = [‘a’, ‘b’]  
... def \_\_init\_\_(self): self.d = 4 # Невозможно добавить новый атрибут,...  
>>> X = D() # когда отсутствует атрибут \_\_dict\_\_  
AttributeError: ‘D’ object has no attribute ‘d’  
Однако возможность добавлять новые атрибуты все-таки существует – для это-  
го необходимо включить имя \_\_dict\_\_ в список \_\_slots\_\_, разрешив тем самым   
создать словарь с пространством имен. В этом случае действовать будут оба ме-  
ханизма хранения имен, однако обобщенные инструменты, такие как getattr,   
будут воспринимать их, как единое множество атрибутов:  
>>> class D:  
... \_\_slots\_\_ = [‘a’, ‘b’, ‘\_\_dict\_\_’] # Добавить \_\_dict\_\_ в слоты  
... c = 3 # Атрибуты класса действуют как обычно  
... def \_\_init\_\_(self): self.d = 4 # Имя d будет добавлено в \_\_dict\_\_,   
... # а не в \_\_slots\_\_  
>>> X = D()  
>>> X.d  
4  
>>> X.\_\_dict\_\_ # Некоторые объекты имеют оба атрибута, \_\_dict\_\_ и \_\_slots\_\_  
{‘d’: 4} # getattr() может извлекать атрибуты любого типа  
>>> X.\_\_slots\_\_  
[‘a’, ‘b’, ‘\_\_dict\_\_’]  
>>> X.c  
3  
>>> X.a # Все атрибуты экземпляра не определены,   
AttributeError: a # пока им не будет присвоено значение  
>>> X.a = 1  
>>> getattr(X, ‘a’,), getattr(X, ‘c’), getattr(X, ‘d’)  
(1, 3, 4)  
Если потребуется реализовать универсальный способ получения значений   
всех атрибутов экземпляра, необходимо учесть наличие двух форм хранения   
атрибутов или использовать функцию dir, которая дополнительно возвращает   
все унаследованные атрибуты (для получения ключей в следующем примере   
используется итератор словаря):  
>>> for attr in list(X.\_\_dict\_\_) + X.\_\_slots\_\_:  
... print(attr, ‘=>’, getattr(X, attr))

Другие расширения в классах нового стиля   
883  
d => 4  
a => 1  
b => 2  
\_\_dict\_\_ => {‘d’: 4}  
Поскольку любой из этих атрибутов может отсутствовать, более правильный   
способ выглядит, как показано ниже (функция getattr позволяет определять   
возвращаемое значение по умолчанию):  
>>> for attr in list(getattr(X,’\_\_dict\_\_’, [])) + getattr(X, ‘\_\_slots\_\_’, []):  
... print(attr, ‘=>’, getattr(X, attr))  
   
d => 4  
a => 1  
b => 2  
\_\_dict\_\_ => {‘d’: 4}  
Несколько суперклассов со списками \_\_slot\_\_  
Обратите внимание, что в этой реализации просматривается содержимое атри-  
бута \_\_slots\_\_ только самого нижнего в дереве класса, наследуемого экземпля-  
ром. Если в дереве имеется несколько классов, обладающих собственными   
атрибутами \_\_slots\_\_, универсальные инструменты должны иначе подходить   
к получению списка атрибутов (например, рассматривать имена слотов, как   
атрибуты классов, а не экземпляров).  
Объявления слотов могут присутствовать сразу в нескольких классах в дереве,   
но они имеют дополнительные ограничения, которые будет слишком сложно   
объяснить, пока вы еще не знаете, что слоты реализованы в виде дескрипторов   
на уровне класса (эту особенность мы детально будем рассматривать в послед-  
ней части книги):  
 •  
Если подкласс наследует суперкласс, который не имеет атрибута \_\_slots\_\_,   
атрибут \_\_dict\_\_ суперкласса будет доступен всегда, что делает бессмыслен-  
ным использование атрибута \_\_slots\_\_ в подклассе.  
 •  
Если класс определяет слот с тем же именем, что и суперкласс, версия име-  
ни, объявленная в суперклассе, будет доступна только при непосредствен-  
ном обращении к дескриптору в суперклассе.  
 •  
Поскольку объявление \_\_slots\_\_ имеет значение только для класса, в кото-  
ром оно присутствует, подклассы автоматически получат атрибут \_\_dict\_\_,   
если не определят свой атрибут \_\_slots\_\_.  
В общем для получения списка атрибутов экземпляра при использовании   
слотов в нескольких классах может потребоваться: подъем по дереву классов   
вручную, использование функции dir или подход, при котором имена слотов   
рассматриваются, как совершенно отдельная категория имен:  
>>> class E:  
... \_\_slots\_\_ = [‘c’, ‘d’] # Суперкласс имеет слоты  
...  
>>> class D(E):  
... \_\_slots\_\_ = [‘a’, ‘\_\_dict\_\_’] # Его подкласс также имеет слоты  
...  
>>> X = D()  
>>> X.a = 1; X.b = 2; X.c = 3 # Экземпляр объединяет слоты в себе  
>>> X.a, X.c

884   
Глава 31. Дополнительные возможности классов   
(1, 3)  
   
>>> E.\_\_slots\_\_ # Но в классах слоты не объединяются  
[‘c’, ‘d’]  
>>> D.\_\_slots\_\_  
[‘a’, ‘\_\_dict\_\_’]  
>>> X.\_\_slots\_\_ # Экземпляр наследует \_\_slots\_\_   
[‘a’, ‘\_\_dict\_\_’] # \*ближайшего\* класса  
>>> X.\_\_dict\_\_ # И имеет собственный атрибут \_\_dict\_\_  
{‘b’: 2}  
>>> for attr in list(getattr(X,’\_\_dict\_\_’, [])) + getattr(X, ‘\_\_slots\_\_’, []):  
... print(attr, ‘=>’, getattr(X, attr))  
...  
b => 2 # Слоты суперкласса отсутствуют!  
a => 1  
\_\_dict\_\_ => {‘b’: 2}  
>>> dir(X) # dir() включает имена всех слотов  
[...множество имен опущено... ‘a’, ‘b’, ‘c’, ‘d’]  
Когда требуется выработать универсальное решение, слоты, вероятно, лучше   
рассматривать как атрибуты класса, а не пытаться представлять их, как обыч-  
ные атрибуты экземпляра. Дополнительную общую информацию о слотах вы   
найдете в стандартном руководстве по языку Python. Кроме того, ознакомьтесь   
с примерами использования атрибутов на основе обоих механизмов хранения,   
\_\_slots\_\_ и \_\_dict\_\_, которые приводятся в обсуждении декоратора Private,   
в главе 38.  
В качестве показательного примера, с какой стороны могут возникнуть про-  
блемы со слотами при создании универсальных инструментов, посмотрите   
в предыдущей главе реализацию примесных классов lister.py, в разделе, посвя-  
щенном множественному наследованию, – там, в примечании, описываются   
проблемы в примере, имеющие отношение к слотам. В таких инструментах,   
пытающихся обобщить процесс получения списка атрибутов, для работы со   
слотами потребуется либо написать дополнительный программный код, либо   
вообще выработать собственную политику обращения с атрибутами в слотах.  
Свойства класса  
Механизм, известный как свойства, обеспечивает в классах нового стиля еще   
один способ определения методов, вызываемых автоматически при обращении   
или присваивании атрибутам экземпляра. Эта особенность во многих случаях   
представляет собой альтернативу методам перегрузки операторов \_\_getattr\_\_   
и \_\_setattr\_\_, которые мы рассматривали в главе 29. Свойства обладают тем же   
эффектом, что и эти два метода, только в этом случае выполняется вызов мето-  
да даже при простом обращении к атрибуту, что бывает полезно для атрибутов,   
значения которых вычисляются динамически. Свойства (и слоты) основаны на   
новом понятии дескрипторов атрибутов – темы слишком сложной, чтобы об-  
суждать ее здесь.  
Проще говоря, свойства – это тип объектов, который присваивается именам   
атрибутов класса. Они создаются вызовом встроенной функции property, ко-  
торой передаются три метода (обработчики операций чтения, присваивания   
и удаления), и строкой документирования – если в каком-либо аргументе пере-  
дается значение None, следовательно, эта операция не поддерживается. Опре-  
деление свойств обычно производится на верхнем уровне в инструкции class

Другие расширения в классах нового стиля   
885  
(например, name = property(...)). Когда выполняется такое присваивание, при   
попытке доступа к атрибуту класса (то есть, obj.name) автоматически будет   
вызываться один из методов доступа. Например, метод \_\_getattr\_\_ позволяет   
классам перехватывать попытки доступа к неопределенным атрибутам класса:  
>>> class classic:  
... def \_\_getattr\_\_(self, name):  
... if name == ‘age’:  
... return 40  
... else:  
... raise AttributeError  
...  
>>> x = classic()  
>>> x.age # Вызовет метод \_\_getattr\_\_  
40  
>>> x.name # Вызовет метод \_\_getattr\_\_  
AttributeError  
Ниже тот же пример, но уже с использованием свойств (обратите внимание,   
что свойства могут использоваться в любых классах, но для корректной рабо-  
ты операции присваивания в версии 2.6 необходимо, чтобы классы прямо или   
косвенно наследовали класс object):  
>>> class newprops(object):  
... def getage(self):  
... return 40  
... age = property(getage, None, None, None) # get,set,del,docs  
...  
>>> x = newprops()  
>>> x.age # Вызовет метод getage  
40  
>>> x.name # Нормальная операция извлечения  
AttributeError: newprops instance has no attribute ‘name’  
(AttributeError: экземпляр newprops не имеет атрибута ‘name’)  
В некоторых случаях свойства могут быть менее сложными и работать быстрее,   
чем при использовании традиционных подходов. Например, когда добавляется   
поддержка операции присваивания атрибуту, свойства становятся более при-  
влекательными – программный код выглядит компактнее и в операцию при-  
сваивания не вовлекаются дополнительные вызовы методов, если не требуется   
производить дополнительных вычислений:  
>>> class newprops(object):  
... def getage(self):  
... return 40  
... def setage(self, value):  
... print(‘set age:’, value)  
... self.\_age = value  
... age = property(getage, setage, None, None)  
...  
>>> x = newprops()  
>>> x.age # Вызовет метод getage  
40  
>>> x.age = 42 # Вызовет метод setage  
set age: 42  
>>> x.\_age # Нормальная операция извлечения; нет вызова getage  
42

886   
Глава 31. Дополнительные возможности классов   
>>> x.job = ‘trainer’ # Нормальная операция присваивания; нет вызова setage  
>>> x.job # Нормальная операция извлечения; нет вызова getage  
‘trainer’  
При эквивалентном классическом решении проблемы класс мог бы произво-  
дить лишние вызовы метода и, возможно, выполнять присваивание значения   
атрибуту с использованием словаря (или с помощью нового метода \_\_setattr\_\_,   
наследуемого классами нового стиля от суперкласса object), чтобы избежать   
зацикливания:  
>>> class classic:  
... def \_\_getattr\_\_(self, name): # При ссылке на неопределенный атрибут  
... if name == ‘age’:  
... return 40  
... else:  
... raise AttributeError  
... def \_\_setattr\_\_(self, name, value): # Для всех операций присваивания  
... print(‘set:’, name, value)  
... if name == ‘age’:  
... self.\_\_dict\_\_[‘\_age’] = value  
... else:  
... self.\_\_dict\_\_[name] = value  
...  
>>> x = classic()  
>>> x.age # Вызовет метод \_\_getattr\_\_  
40  
>>> x.age = 41 # Вызовет метод \_\_setattr\_\_  
set: age 41  
>>> x.\_age # Определен: нет вызова \_\_getattr\_\_  
41  
>>> x.job = ‘trainer’ # Запустит метод \_\_setattr\_\_ опять  
>>> x.job # Определен: нет вызова \_\_getattr\_\_  
Для этого примера свойства обладают неоспоримым преимуществом. Однако   
в некоторых приложениях методы \_\_getattr\_\_ и \_\_setattr\_\_ по-прежнему могут   
быть востребованы для обеспечения более динамичных или универсальных   
интерфейсов, чем можно реализовать с помощью свойств. Например, во мно-  
гих случаях невозможно заранее определить набор поддерживаемых атрибу-  
тов, которые могут даже не существовать вообще в каком-либо виде на момент   
написания класса (например, при делегировании ссылок на произвольные ме-  
тоды в обернутых/встроенных объектах). В таких случаях использование бо-  
лее универсальных методов обслуживания атрибутов \_\_getattr\_\_ и \_\_setattr\_\_,   
которым передаются имена атрибутов, может оказаться предпочтительнее.   
Кроме того, простейшие ситуации могут обслуживаться этими обработчика-  
ми, поэтому свойства следует рассматривать как дополнительное и необяза-  
тельное к использованию расширение.  
В заключительной части книги, в главе 37, приводится дополнительная ин-  
формация по этим двум возможностям. Там вы узнаете, что существует воз-  
можность определять свойства с помощью декораторов  функций, о которых   
мы поговорим ниже, в этой главе.  
Метод \_\_getattribute\_\_ и дескрипторы  
Метод \_\_getattribute\_\_ имеется только в классах нового стиля и позволя-  
ет классам перехватывать все попытки обращения к атрибутам, а не только

Статические методы и методы класса   
887  
к неопределенным (как метод \_\_getattr\_\_). Кроме того, этот метод более сложен   
в обращении, чем \_\_getattr\_\_, из-за более высокой вероятности зацикливания,   
и чем \_\_setattr\_\_, но уже по другим причинам.   
В дополнение к свойствам и методам перегрузки операторов в языке Python   
поддерживается понятие дескрипторов атрибутов – классов, с методами \_\_  
get\_\_ и \_\_set\_\_, которые присваиваются атрибутам классов. Они наследуются   
экземплярами и перехватывают попытки доступа к определенным атрибутам.   
Дескрипторы представляют собой, в некотором смысле, более обобщенную   
форму свойств. Фактически свойства – это упрощенный вариант определения   
дескрипторов специфического типа, основанных на вызовах функций, управ-  
ляющих доступом к атрибутам. Кроме того, дескрипторы используются для   
реализации слотов, с которыми мы познакомились выше.  
Поскольку свойства, метод \_\_getattribute\_\_ и дескрипторы – это достаточно   
сложные темы, мы отложим их дальнейшее обсуждение до главы 37, в заклю-  
чительной части книги, пока поближе не познакомимся со свойствами.  
Метаклассы  
Большинство изменений и дополнительных особенностей в классах нового сти-  
ля связано с возможностью наследования типов, о чем говорилось выше в этой   
главе, потому что возможность наследования типов и классы нового стиля   
были введены одновременно с объединением понятий тип/класс в Python 2.2.   
Как мы уже видели, в версии 3.0 это объединение было завершено: теперь клас-  
сы – это типы, а типы – классы.  
Наряду с этими изменениями в языке Python был выработан более согласо-  
Python был выработан более согласо-  
 был выработан более согласо-  
ванный протокол метаклассов, которые являются подклассами объекта type   
и реализуют операции создания классов. Они обеспечивают отличную воз-  
можность управления объектами классов и их расширения. Тема метаклассов   
достаточно сложна, а кроме того, они не являются необходимыми для боль-  
шинства программистов, поэтому здесь мы не будем углубляться в детали. Мы   
еще столкнемся с метаклассами ниже, в этой главе, когда будем знакомиться   
с декораторами классов, и исследуем их во всех подробностях в главе 39, в за-  
ключительной части книги.  
Статические методы и методы класса  
Начиная с версии Python 2.2, появилась возможность определять методы клас-  
са, которые могут вызываться без участия экземпляра: статические методы   
работают почти так же, как обычные функции, только расположенные внутри   
класса, а методы  класса получают сам класс вместо экземпляра. Несмотря   
на то что эта особенность была добавлена вместе с классами нового стиля, об-  
суждавшимися в предыдущих разделах, статические методы и методы класса   
можно использовать и в классических классах тоже.   
Чтобы сделать возможными эти режимы работы методов, внутри класса долж-  
ны вызываться специальные встроенные функции staticmethod и classmethod   
или использоваться декораторы, с которыми мы познакомимся ниже, в этой   
главе. В Python 3.0 методы, которым не передается ссылка на экземпляр и ко-  
торые вызываются только через имя класса, не требуют объявления с помо-  
щью функции staticmethod, но такое объявление обязательно для методов, ко-  
торые предполагается вызывать через экземпляры.

888   
Глава 31. Дополнительные возможности классов   
Зачем нужны специальные методы?  
Как мы уже знаем, обычно методы получают объект экземпляра в первом ар-  
гументе, который играет роль подразумеваемого объекта вызова метода. При   
этом на сегодняшний день существует две возможности изменить эту модель.   
Прежде чем я расскажу о них, я должен пояснить, почему это может быть важ-  
ным для нас.  
Иногда в программах бывает необходимо организовать обработку данных,   
связанных с классами, а не с экземплярами. Например, следить за числом эк-  
земпляров класса или вести список всех экземпляров класса, находящихся   
в настоящий момент в памяти. Такого рода информация связана с классами   
и должна обрабатываться на уровне класса, а не экземпляров. То есть такая   
информация обычно сохраняется в самом классе и обрабатывается, независи-  
мо от наличия экземпляров класса.  
Для решения таких задач часто бывает достаточно простых функций, опреде-  
ления которых находятся за пределами классов. Такие функции могут обра-  
щаться к атрибутам класса через его имя – им требуется доступ только к дан-  
ным класса и никогда – к экземплярам. Однако, чтобы теснее связать такой   
программный с классом и обеспечить возможность его адаптации с помощью   
механизма наследования, будет лучше помещать такого рода функции внутрь   
самого класса. Для этого нам и нужны методы класса, которые не ожидают по-  
лучить аргумент self с экземпляром.  
В языке Python для этих целей поддерживаются статические методы – про-  
стые функции без аргумента self, вложенные в определение класса и пред-  
назначенные для работы с атрибутами класса, а не экземпляра. Статические   
методы никогда автоматически не получают ссылку self на экземпляр, неза-  
висимо от того, вызываются они через имя класса или через экземпляр. Такие   
методы обычно используются для обработки информации, имеющей отноше-  
ние ко всем экземплярам, а не для реализации поведения экземпляров.  
Кроме того, в языке Python поддерживается также понятие методов класса.   
На практике методы класса используются реже, и в первом аргументе им ав-  
томатически передается объект класса, независимо от того, вызываются они   
через имя класса или через экземпляр. Такие методы могут получить доступ   
к данным класса через аргумент self, даже когда они вызываются относитель-  
но экземпляра. Обычные методы (которые формально называются методами   
экземпляра) при вызове получают подразумеваемый экземпляр, а статические   
методы и методы класса – нет.   
Статические методы в 2.6 и 3.0  
Статические методы поддерживаются в обеих версиях Python, 2.6 и 3.0, но   
в версии 3.0 требования к их реализации несколько изменились. Поскольку   
в этой книге рассматриваются обе версии, я должен объяснить основные раз-  
личия между моделями, лежащими в основе, прежде чем перейти к программ-  
ному коду.   
В действительности мы уже начинали поднимать эту тему в предыдущей главе,   
когда говорили о несвязанных методах. Напомню, что в Python 2.6 и 3.0, когда   
метод вызывается относительно экземпляра, ему всегда передается ссылка на   
этот экземпляр. Однако в Python 3.0 извлечение методов через имя класса ин-  
Python 3.0 извлечение методов через имя класса ин-  
 3.0 извлечение методов через имя класса ин-  
терпретируется иначе, чем в 2.6:

Статические методы и методы класса   
889  
 •  
В Python 2.6 операция извлечения метода по имени класса возвращает не-  
связанный метод, при вызове которого требуется вручную передавать эк-  
земпляр.   
 •  
В Python 3.0 операция извлечения метода по имени класса возвращает про-  
стую функцию, которой не требуется передавать ссылку на экземпляр.  
Другими словами, в Python 2.6 методам всегда необходимо передавать экзем-  
Python 2.6 методам всегда необходимо передавать экзем-  
 2.6 методам всегда необходимо передавать экзем-  
пляр, независимо от того, вызываются они через имя класса или через эк-  
земпляр. В Python 3.0, напротив, экземпляр требуется передавать методам,   
только если они ожидают получить его, – методы без аргумента self могут вы-  
зываться через имя класса без передачи им ссылки на экземпляр. То есть в вер-  
сии 3.0 допускается объявлять простые функции внутри класса, при условии,   
что они не ожидают получить и им не будет передаваться аргумент со ссылкой   
на экземпляр. В результате:  
 •  
В Python 2.6 мы всегда должны объявлять метод как статический, чтобы   
иметь возможность вызывать его без передачи ссылки на экземпляр, неза-  
висимо от того, вызываются он через имя класса или через экземпляр.  
 •  
В Python 3.0 от нас не требуется объявлять метод, как статический, если он   
будет вызываться только через имя класса, но мы обязаны объявлять его   
статическим, если он может вызываться через экземпляр.  
В качестве примера предположим, что необходимо использовать атрибуты   
класса для подсчета числа экземпляров, созданных из класса. В следующем   
файле spam.py представлена первая попытка – класс содержит счетчик в виде   
атрибута класса, конструктор, который наращивает счетчик при создании но-  
вого экземпляра, и метод, который выводит значение счетчика. Не забывайте,   
что атрибуты класса совместно используются всеми экземплярами. Поэтому   
наличие счетчика непосредственно в объекте класса гарантирует, что он будет   
хранить число всех экземпляров:  
class Spam:  
 numInstances = 0  
 def \_\_init\_\_(self):  
 Spam.numInstances = Spam.numInstances + 1  
 def printNumInstances():  
 print(“Number of instances created: “, Spam.numInstances)  
Метод printNumInstances предназначен для обработки данных класса, а не эк-  
земпляров – эти данные являются общими для всех экземпляров. Вследствие   
этого нам необходима возможность вызывать его, не передавая ссылку на эк-  
земпляр. Действительно, зачем нам создавать новый экземпляр для получения   
числа экземпляров, ведь это изменит число экземпляров, которое мы пытаем-  
ся получить! Другими словами, нам нужен «статический» метод, не имеющий   
аргумента self.  
Будет ли работать такая реализация, зависит от версии интерпретатора и от   
способа вызова метода – через имя класса или через экземпляр. В версии 2.6   
(да и в любой версии 2.X) такая реализация не будет работать – вызов метода   
printNumInstances без аргумента self, как через имя класса, так и через экзем-  
пляр, будет терпеть неудачу. (Я опустил часть текста сообщения об ошибке для   
экономии места):  
C:\misc> c:\python26\python  
>>> from spam import Spam

890   
Глава 31. Дополнительные возможности классов   
>>> a = Spam() # В 2.6 невозможно вызывать несвязанные методы  
>>> b = Spam() # По умолчанию методы ожидают получить self  
>>> c = Spam()  
   
>>> Spam.printNumInstances()  
TypeError: unbound method must be called with Spam instance as first argument (got   
nothing instead)  
>>> a.printNumInstances()  
TypeError: printNumInstances() takes no arguments (1 given)  
Проблема состоит в том, что в версии 2.6 несвязанные методы экземпляра – это   
не то же самое, что простые функции. То есть вызов метода, такого как print-  
NumInstances, через имя класса и без передачи ему экземпляра будет терпеть не-  
удачу в Python 2.6, но будет работать в Python 3.0. С другой стороны, попытка   
вызвать метод относительно экземпляра будет терпеть неудачу в обеих версиях   
Python, потому что в этом случае интерпретатор автоматически передаст экзем-  
, потому что в этом случае интерпретатор автоматически передаст экзем-  
пляр методу, который не имеет соответствующего аргумента, чтобы принять его:  
Spam.printNumInstances() # Ошибка в 2.6, работает в 3.0  
instance.printNumInstances() # Ошибка в обеих версиях, 2.6 и 3.0  
Если вы используете версию 3.0 и предполагаете вызывать метод без аргумента   
self только через имя класса, можете считать, что вы уже создали статический   
метод. Однако, чтобы иметь возможность вызывать методы без аргумента self   
через имя класса в версии 2.6 и через экземпляры в обеих версиях, 2.6 и 3.0,   
вам необходимо либо использовать иной подход к реализации, либо каким-то   
образом пометить подобные методы, как специальные. Рассмотрим обе воз-  
можности по порядку.  
Альтернативы статическим методам  
Кроме возможности пометить метод, как специальный, существуют и другие   
приемы, которые можно попробовать. Если для доступа к атрибутам класса   
требуется вызывать функции, которые не принимают ссылку на экземпляр,   
самая простая мысль, которая приходит в голову, – сделать метод обычной   
функцией, а не методом класса. При таком способе функции не требуется пе-  
редавать экземпляр класса. Например, следующая версия spam.py действует   
одинаково в Python 3.0 и 2.6 (правда, в этой версии инструкция print отобра-  
жает лишние круглые скобки при выполнении под управлением Python 2.6):  
def printNumInstances():  
 print(“Number of instances created: “, Spam.numInstances)  
   
class Spam:  
 numInstances = 0  
 def \_\_init\_\_(self):  
 Spam.numInstances = Spam.numInstances + 1  
   
>>> import spam  
>>> a = spam.Spam()  
>>> b = spam.Spam()  
>>> c = spam.Spam()  
>>> spam.printNumInstances() # Но функция может находиться слишком далеко от   
Number of instances created: 3 # определения класса и не может   
>>> spam.Spam.numInstances # адаптироваться в подклассах  
3

Статические методы и методы класса   
891  
Поскольку имя класса доступно простой функции в виде глобальной перемен-  
ной, все работает прекрасно. Кроме того, обратите внимание, что имя самой   
функции также является глобальным, но только в этом единственном модуле –   
оно не будет конфликтовать с именами в других модулях программы.  
До появления статических методов в Python такой способ был единственным.   
Поскольку в языке Python уже имеются модули, которые играют роль инстру-  
Python уже имеются модули, которые играют роль инстру-  
 уже имеются модули, которые играют роль инстру-  
мента разделения пространства имен, можно было бы утверждать, что обычно   
нет никакой необходимости упаковывать функции в классы, если они не реа-  
лизуют функциональность объектов. Простые функции внутри модуля, как   
в данном примере, способны решать большую часть задач, которые возлага-  
ются на методы класса, не имеющие аргумента self, и уже связаны с классом,   
потому что располагаются в том же самом модуле.   
К сожалению, такой подход далек от идеала. С одной стороны, в область ви-  
димости файла добавляется лишнее имя, которое используется для работы   
с единственным классом. С другой – функция не имеет тесной связи с классом.   
Фактически определение функции может находиться за сотни строк от опре-  
деления класса. Но самое неприятное, пожалуй, состоит в том, что простые   
функции, как в данном примере, не могут адаптироваться в подклассах, пото-  
му что они располагаются за пределами пространства имен класса: подклассы   
не могут непосредственно переопределять или замещать такие функции.  
Мы могли бы попытаться обеспечить работоспособность этого примера в обе-  
их версиях интерпретатора, используя обычный метод, и всегда вызывать его   
через (или передавать вручную) экземпляр:  
class Spam:  
 numInstances = 0  
 def \_\_init\_\_(self):  
 Spam.numInstances = Spam.numInstances + 1  
 def printNumInstances(self):  
 print(“Number of instances created: “, Spam.numInstances)  
   
>>> from spam import Spam  
>>> a, b, c = Spam(), Spam(), Spam()  
>>> a.printNumInstances()  
Number of instances created: 3  
>>> Spam.printNumInstances(a)  
Number of instances created: 3  
>>> Spam().printNumInstances() # Эта попытка извлечь счетчик изменяет его!  
Number of instances created: 4  
К сожалению, как уже упоминалось выше, такой подход полностью неприго-  
ден в случае отсутствия доступного экземпляра, а создание нового экземпляра   
изменяет данные класса, как видно в последней строке этого примера. Лучшее   
решение заключается в том, чтобы каким-либо способом пометить метод клас-  
са, который не требует передавать ему ссылку на экземпляр. Как это сделать,   
демонстрируется в следующем разделе.   
Использование статических методов и методов класса  
На сегодняшний день существует еще одна возможность писать простые функ-  
ции, связанные с классом, которые могут вызываться через имя класса или че-  
рез экземпляры. Начиная с версии Python 2.2, имеется возможность создавать   
классы со статическими методами и с методами класса, ни один из которых не

892   
Глава 31. Дополнительные возможности классов   
требует передачи экземпляра класса в виде аргумента. Чтобы определить та-  
кие методы, в классах необходимо вызывать встроенные функции staticmethod   
и classmethod, как упоминалось в обсуждении классов нового стиля. Обе функ-  
ции помечают объект функции как специальный, то есть как не требующий   
передачи экземпляра, в случае применения функции staticmethod, и как требу-  
ющий передачи класса, в случае применения функции classmethod. Например:  
class Methods:  
 def imeth(self, x): # Обычный метод экземпляра  
 print(self, x)  
 def smeth(x): # Статический метод: экземпляр не передается  
 print(x)  
 def cmeth(cls, x): # Метод класса: получает класс, но не экземпляр  
 print(cls, x)  
   
 smeth = staticmethod(smeth) # Сделать smeth статическим методом  
 cmeth = classmethod(cmeth) # Сделать cmeth методом класса.  
Обратите внимание, как две последние операции присваивания в этом фраг-  
менте просто переприсваивают имена методов smeth и cmeth. Атрибуты созда-  
ются и изменяются с помощью операции присваивания в инструкции class,   
поэтому эти заключительные операции присваивания переопределяют ин-  
струкции def, выполненные ранее.  
С технической точки зрения, язык Python теперь поддерживает три разновид-  
Python теперь поддерживает три разновид-  
 теперь поддерживает три разновид-  
ности методов: методы  экземпляра, статические  методы и методы  клас-  
са. Кроме того, в Python 3.0 эта модель дополнена возможностью без лишних   
сложностей создавать в классах простые функции, которые играют роль стати-  
ческих методов при обращении к ним через имя класса.  
 Методы экземпляра – это обычные (и используемые по умолчанию) методы,   
которые мы видели в этой книге. Для воздействия на объект экземпляра всегда   
следует вызывать методы экземпляра. Когда методы вызываются через экзем-  
пляр, интерпретатор автоматически передает экземпляр в первом аргументе –   
когда метод вызывается через имя класса, экземпляр необходимо передавать   
методам вручную (для простоты я опустил инструкции импортирования не-  
которых классов):  
>>> obj = Methods() # Создать экземпляр  
   
>>> obj.imeth(1) # Обычный вызов, через экземпляр  
<\_\_main\_\_.Methods object...> 1 # Будет преобразован в вызов imeth(obj, 1)  
   
>>> Methods.imeth(obj, 2) # Обычный вызов, через класс  
<\_\_main\_\_.Methods object...> 2 # Экземпляр передается явно  
Статические методы, напротив, вызываются без аргумента с экземпляром.   
В отличие от простых функций, за пределами класса их имена ограничены об-  
ластью видимости класса, в котором они определяются и могут отыскиваться   
механизмом наследования. Функции, которым не передаются ссылки на эк-  
земпляры, в Python 3.0 могут вызываться обычным образом через имя клас-  
Python 3.0 могут вызываться обычным образом через имя клас-  
 3.0 могут вызываться обычным образом через имя клас-  
са, но в Python 2.6 по умолчанию – никогда. Применение встроенной функции   
staticmethod обеспечивает возможность вызывать такие методы через экзем-  
пляр в версии 3.0, и через имя класса и через экземпляр – в Python 2.6 (в вер-  
Python 2.6 (в вер-  
 2.6 (в вер-  
сии 3.0 вызов через имя класса возможен и без применения staticmethod, но вы-  
зов через экземпляр – нет):

Статические методы и методы класса   
893  
>>> Methods.smeth(3) # Вызов статического метода, через имя класса  
3 # Экземпляр не передается и не ожидается  
>>> obj.smeth(4) # Вызов статического метода, через экземпляр  
4 # Экземпляр не передается  
Методы  класса похожи на них, но интерпретатор автоматически передает   
методам класса сам класс (а не экземпляр) в первом аргументе, независимо от   
того, вызываются они через имя класса или через экземпляр:  
>>> Methods.cmeth(5) # Вызов метода класса, через имя класса  
<class ‘\_\_main\_\_.Methods’> 5 # Будет преобразован в вызов cmeth(Methods, 5)   
>>> obj.cmeth(6) # Вызов метода класса, через экземпляр  
<class ‘\_\_main\_\_.Methods’> 6 # Будет преобразован в вызов cmeth(Methods, 6)   
Подсчет количества экземпляров с помощью   
статических методов  
Теперь, зная о существовании встроенных функций, можно реализовать ста-  
тический метод, эквивалентный оригинальному примеру этого раздела, – он   
помечен, как специальный, поэтому ему никогда автоматически не будет пере-  
даваться ссылка на экземпляр:  
class Spam: # Для доступа к данным класса используется  
 numInstances = 0 # статический метод  
 def \_\_init\_\_(self):   
 Spam.numInstances += 1  
 def printNumInstances():  
 print(“Number of instances:”, Spam.numInstances)  
 printNumInstances = staticmethod(printNumInstances)  
   
Использование встроенной функции staticmethod позволяет вызывать метод,   
не принимающий аргумент self, через имя класса или через любой экземпляр   
в обеих версиях Python, 2.6 и 3.0:  
>>> a = Spam()  
>>> b = Spam()  
>>> c = Spam()  
>>> Spam.printNumInstances() # Вызывается, как простая функция  
Number of instances: 3  
>>> a.printNumInstances() # Аргумент с экземпляром не передается  
Number of instances: 3  
По сравнению с простым перемещением printNumInstances за пределы класса,   
как описывалось ранее, эта версия требует дополнительный вызов функции   
staticmethod. При этом здесь область видимости имени функции ограничена   
классом (имя не будет вступать в конфликт с другими именами в модуле), про-  
граммный код перемещен туда, где он используется (внутрь инструкции class),   
и подклассы получают возможность адаптировать статический метод насле-  
дованием – этот подход более удобен, чем импортирование функций из файлов,   
в которых находятся определения суперклассов. Это иллюстрирует следую-  
щий подкласс и листинг нового интерактивного сеанса:  
class Sub(Spam):  
 def printNumInstances(): # Переопределяет статический метод

894   
Глава 31. Дополнительные возможности классов   
 print(“Extra stuff...”) # Который вызывает оригинал  
 Spam.printNumInstances()  
 printNumInstances = staticmethod(printNumInstances)  
   
>>> a = Sub()  
>>> b = Sub()  
>>> a.printNumInstances() # Вызов через экземпляр подкласса  
Extra stuff...  
Number of instances: 2  
>>> Sub.printNumInstances() # Вызов через имя подкласса  
Extra stuff...  
Number of instances: 2  
>>> Spam.printNumInstances()  
Number of instances: 2  
Кроме того, классы могут наследовать статические методы, не переопределяя   
их, – они будут вызываться без передачи им ссылки на экземпляр, независимо   
от их местоположения в дереве классов:  
>>> class Other(Spam): pass # Наследует оригинальный статический метод   
   
>>> c = Other()  
>>> c.printNumInstances()  
Number of instances: 3  
Подсчет экземпляров с помощью методов класса  
Интересно отметить, что аналогичные действия можно реализовать с помощью   
метода класса – следующий класс обладает тем же поведением, что и класс   
со статическим методом, представленный выше, но в нем используется метод   
класса, который в первом аргументе принимает класс экземпляра. Методы   
класса автоматически получают объект класса:  
class Spam:  
 numInstances = 0 # Вместо статического метода используется метод класса  
 def \_\_init\_\_(self):  
 Spam.numInstances += 1  
 def printNumInstances(cls):  
 print(“Number of instances:”, cls.numInstances)  
 printNumInstances = classmethod(printNumInstances)  
Используется этот класс точно так же, как и предыдущая версия, но его ме-  
тод printNumInstances принимает объект класса, а не экземпляра, независимо от   
того, вызывается он через имя класса или через экземпляр:  
>>> a, b = Spam(), Spam()  
>>> a.printNumInstances() # В первом аргументе передается класс  
Number of instances: 2  
>>> Spam.printNumInstances() # Также в первом аргументе передается класс  
Number of instances: 2  
Однако, используя методы класса, имейте в виду, что они принимают класс,   
самый близкий к объекту вызова. Это влечет за собой ряд важных послед-  
ствий, который оказывают влияние на попытки изменить данные класса через   
переданный методу класс. Например, если в модуле test.py мы определим под-  
класс, адаптирующий предыдущую версию метода Spam.printNumInstances так,   
чтобы он дополнительно выводил свой аргумент cls, и запустим новый сеанс:

Статические методы и методы класса   
895  
class Spam:  
 numInstances = 0 # Отслеживает количество экземпляров  
 def \_\_init\_\_(self):  
 Spam.numInstances += 1  
 def printNumInstances(cls):  
 print(“Number of instances:”, cls.numInstances, cls)  
 printNumInstances = classmethod(printNumInstances)  
   
class Sub(Spam):  
 def printNumInstances(cls): # Переопределяет метод класса  
 print(“Extra stuff...”, cls) # Но вызывает оригинал  
 Spam.printNumInstances()  
 printNumInstances = classmethod(printNumInstances)  
   
class Other(Spam): pass # Наследует метод класса  
всякий раз, когда будет вызываться метод класса, интерпретатор будет переда-  
вать ему самый близкий класс, даже для подклассов, не имеющих собственной   
реализации метода класса:  
>>> x, y = Sub(), Spam()  
>>> x.printNumInstances() # Вызов через экземпляр подкласса  
Extra stuff... <class ‘test.Sub’>  
Number of instances: 2 <class ‘test.Spam’>  
>>> Sub.printNumInstances() # Вызов через сам подкласс  
Extra stuff... <class ‘test.Sub’>  
Number of instances: 2 <class ‘test.Spam’>  
>>> y.printNumInstances()  
Number of instances: 2 <class ‘test.Spam’>  
В первом случае здесь метод класса вызывается через экземпляр подкласса Sub,   
и интерпретатор передает методу ближайший к экземпляру класс Sub. В дан-  
ном случае никаких проблем не возникает, так как версия метода, переопреде-  
ленная в классе Sub, явно вызывает оригинальную версию метода в суперклас-  
се Spam, при этом метод суперкласса принимает класс Spam в первом аргументе.   
Но посмотрите, что произойдет в случае обращения к объекту, который просто   
наследует метод класса:  
>>> z = Other()  
>>> z.printNumInstances()  
Number of instances: 3 <class ‘test.Other’>  
Здесь в последнем вызове методу класса Spam передается класс Other. В данном   
случае метод работает потому, что он всего лишь извлекает значение счетчи-  
ка, который обнаруживает в классе Spam, благодаря механизму наследования.   
Однако если бы этот метод попытался присвоить новое значение атрибуту клас-  
са, он изменил бы атрибут класса Other, а не Spam! В данном конкретном случае,   
вероятно, было бы лучше жестко указать имя класса, в котором производится   
изменение данных, чем полагаться на передаваемый аргумент класса.  
Подсчет экземпляров для каждого класса   
с помощью методов класса  
Фактически методы класса всегда получают ближайший класс в дереве насле-  
дования, поэтому:  
 •  
Применение статических методов, в которых явно указывается имя класса,   
может оказаться более удачным решением для обработки данных класса.

896   
Глава 31. Дополнительные возможности классов   
 •  
Методы классов лучше подходят для обработки данных, которые могут от-  
личаться для каждого конкретного класса в иерархии.  
Например, для реализации счетчиков экземпляров каждого класса в отдель-  
ности лучше подошли бы методы класса. В следующем примере суперкласс   
определяет метод класса, управляющий информацией о состоянии, которая   
отличается для разных классов в дереве, – подобно тому, как методы экзем-  
пляра управляют информацией о состоянии экземпляров:  
class Spam:  
 numInstances = 0  
 def count(cls): # Счетчик экземпляров для каждого отдельного класса  
 cls.numInstances += 1 # cls – ближайший к экземпляру класс  
 def \_\_init\_\_(self):  
 self.count() # Передаст self.\_\_class\_\_ для подсчета  
 count = classmethod(count)  
   
class Sub(Spam):  
 numInstances = 0  
 def \_\_init\_\_(self): # Переопределяет \_\_init\_\_  
 Spam.\_\_init\_\_(self)  
   
class Other(Spam): # Наследует \_\_init\_\_  
 numInstances = 0  
   
>>> x = Spam()  
>>> y1, y2 = Sub(), Sub()  
>>> z1, z2, z3 = Other(), Other(), Other()  
>>> x.numInstances, y1.numInstances, z1.numInstances  
(1, 2, 3)  
>>> Spam.numInstances, Sub.numInstances, Other.numInstances  
(1, 2, 3)  
Статические методы и методы класса могут использоваться и в других ситуа-  
циях, которые мы не будем рассматривать здесь, – ищите дополнительную ин-  
формацию в других источниках. Однако в последних версиях Python создание   
статических методов и методов класса можно упростить, воспользовавшись де-  
кораторами функций – способом применения одной функции к другой. Вообще   
декораторы функций имеют более широкую область применения, чем простое   
объявление статических методов, которое, впрочем, стало основной причиной   
их появления. Синтаксис декораторов позволяет нам также расширять классы   
в Python 2.6 и 3.0 – инициализировать данные, такие как счетчик numInstances   
в последнем примере. Как это делается, описывается в следующем разделе.  
Декораторы и метаклассы: часть 1  
Прием с вызовом функции staticmethod, описанный в предыдущем разделе, вы-  
глядит малопонятным для некоторых пользователей, поэтому была добавлена   
возможность, упрощающая эту операцию. Декораторы функций обеспечива-  
ют способ определения специальных режимов работы функций, обертывая их   
дополнительным слоем логики, реализованной в виде других функций.  
Декораторы функций представляют собой более универсальные инструмен-  
ты: их удобно использовать для добавления самой разной логики не только   
к статическим методам, но и к любым другим функциям. Например, их мож-  
но использовать для расширения функций программным кодом, выполняю-

Декораторы и метаклассы: часть 1   
897  
щим регистрацию вызовов этих функций, проверяющим типы передаваемых   
аргументов в процессе отладки и так далее. В некоторой степени декораторы   
функций напоминают шаблон проектирования делегирования, исследованный   
нами в главе 30, но их главная цель состоит в том, чтобы расширять определен-  
ные функции или методы, а не весь интерфейс объекта.  
Язык Python предоставляет несколько встроенных декораторов функций для   
выполнения таких действий, как создание статических методов, но програм-  
мисты также имеют возможность создавать свои собственные декораторы. Не-  
смотря на то что они строго не привязаны к классам, тем не менее пользова-  
тельские декораторы функций часто оформляются как классы, в которых со-  
храняется оригинальная функция наряду с другими данными, такими как ин-  
формация о состоянии. Кроме того, недавно появилось похожее расширение,   
доступное в Python 2.6 и 3.0: декораторы классов, непосредственно связанные   
с моделью классов, и метаклассы, играющие похожую роль.  
Основы декораторов функций   
Синтаксически декоратор функции – это разновидность объявления функции   
времени выполнения. Декоратор функции записывается в строке, непосред-  
ственно перед строкой с инструкцией def, которая определяет функцию или   
метод, и состоит из символа @, за которым следует то, что называется мета-  
функцией, – функция (или другой вызываемый объект), которая управляет   
другой функцией. В настоящее время статические методы, к примеру, могут   
быть оформлены в виде декораторов, как показано ниже:  
class C:  
 @staticmethod # Синтаксис декорирования  
 def meth():  
 ...  
С технической точки зрения, это объявление имеет тот же эффект, что и фраг-  
мент ниже (передача функции декоратору и присваивание результата первона-  
чальному имени функции):  
class C:  
 def meth():  
 ...  
 meth = staticmethod(meth) # Повторное присваивание имени  
Результат, возвращаемый функцией-декоратором, повторно присваивается   
имени метода. В результате вызов метода по имени функции фактически бу-  
дет приводить к вызову результата, полученному от декоратора staticmethod.   
Декоратор может возвращать объекты любого типа, поэтому данный прием по-  
зволяет декоратору вставлять дополнительный уровень логики, который будет   
запускаться при каждом вызове. Декоратор функции может возвращать как   
оригинальную функцию, так и новый объект, в котором хранится оригиналь-  
ная функция, переданная декоратору, которая будет вызываться косвенно по-  
сле того, как будет выполнен дополнительный слой логики.  
Благодаря этому расширению мы располагаем более удачным способом объ-  
явить статический метод в примере из предыдущего раздела в обеих версиях   
Python, 2.6 и 3.0 (декоратор classmethod используется точно так же):  
class Spam:  
 numInstances = 0

898   
Глава 31. Дополнительные возможности классов   
 def \_\_init\_\_(self):  
 Spam.numInstances = Spam.numInstances + 1  
 @staticmethod  
 def printNumInstances():  
 print(“Number of instances created: “, Spam.numInstances)  
   
a = Spam()  
b = Spam()  
c = Spam()  
Spam.printNumInstances() # Теперь вызовы могут производиться как через класс,   
 # так и через экземпляр!  
a.printNumInstances() # В обоих случаях будет выведено   
 # “Number of instances created: 3”  
Имейте в виду, что staticmethod – это все та же встроенная функция. Она может   
использоваться как декоратор просто потому, что принимает другую функцию   
в виде аргумента и возвращает вызываемый объект. Фактически любая такая   
функция может использоваться в качестве декоратора, даже пользовательские   
функции, которые мы пишем сами, как описывается в следующем разделе.  
Первый пример декоратора функций  
В языке Python уже имеется несколько удобных встроенных функций, кото-  
Python уже имеется несколько удобных встроенных функций, кото-  
 уже имеется несколько удобных встроенных функций, кото-  
рые можно использовать как декораторы, но при этом мы также можем писать   
свои собственные декораторы. Из-за широты применения декораторов мы по-  
святим их созданию целую главу, в следующей части книги. А пока, в каче-  
стве предварительного знакомства, рассмотрим простой пример декоратора,   
определяемого пользователем.  
Вспомните, как в главе 29 говорилось, что метод перегрузки оператора \_\_call\_\_   
реализует в экземплярах классов интерфейс вызова функций. В следующем   
примере этот метод используется в определении класса, который сохраняет де-  
корируемую функцию в экземпляре и перехватывает вызовы по оригинально-  
му имени. А так как это класс, кроме всего прочего в нем имеется возможность   
хранить информацию о состоянии (счетчик произведенных вызовов):  
class tracer:  
 def \_\_init\_\_(self, func):  
 self.calls = 0  
 self.func = func  
 def \_\_call\_\_(self, \*args):  
 self.calls += 1  
 print(‘call %s to %s’ % (self.calls, self.func.\_\_name\_\_))  
 self.func(\*args)  
   
@tracer # То же, что и spam = tracer(spam)  
def spam(a, b, c): # Обертывает spam в объект-декоратор  
 print a, b, c  
   
spam(1, 2, 3) # В действительности вызывается объект-обертка  
spam(‘a’, ‘b’, ‘c’) # То есть вызывается метод \_\_call\_\_ в классе  
spam(4, 5, 6) # Метод \_\_call\_\_ выполняет дополнительные действия  
 # и вызывает оригинальную функцию  
Функция spam передается декоратору tracer, поэтому, когда производится об-  
ращение к оригинальному имени spam, в действительности вызывается метод   
\_\_call\_\_ в классе. Этот метод подсчитывает и регистрирует вызовы, а затем вы-

Декораторы и метаклассы: часть 1   
899  
зывает оригинальную обернутую функцию. Обратите внимание, как исполь-  
зуется синтаксис аргумента \*name для упаковывания и распаковывания аргу-  
ментов, передаваемых функции, – благодаря этому данный декоратор может   
использоваться для обертывания любой функции, с любым числом позицион-  
ных аргументов.  
В результате к оригинальной функции spam добавляется слой дополнительной   
логики. Ниже приводится вывод, полученный от сценария, – первая строка   
создана классом tracer, а вторая – функцией spam:  
call 1 to spam  
1 2 3  
call 2 to spam  
a b c  
call 3 to spam  
4 5 6  
Исследуйте программный код этого примера повнимательнее, чтобы вникнуть   
в его суть. Итак, данный декоратор действует, как обычная функция, при-  
нимающая позиционные аргументы, но он не возвращает результат вызова   
декорируемой функции, не обрабатывает именованные аргументы и не может   
декорировать методы классов (при декорировании методов метод \_\_call\_\_ мог   
бы передавать только экземпляр класса tracer). Как мы узнаем в восьмой ча-  
сти книги, существует множество способов декорирования функций, включая   
вложенные инструкции def, – некоторые из них лучше подходят для декориро-  
вания методов, чем способ, представленный здесь.  
Декораторы классов и метаклассы  
Декораторы функций оказались настолько удобны в обращении, что эта модель   
была расширена в Python 2.6 и 3.0 и теперь она позволяет применять декорато-  
Python 2.6 и 3.0 и теперь она позволяет применять декорато-  
 2.6 и 3.0 и теперь она позволяет применять декорато-  
ры не только к функциям, но и к классам. В двух словах, декораторы классов   
похожи на декораторы функций, но они запускаются после инструкции class,   
чтобы повторно присвоить имя класса вызываемому объекту. Кроме того, они   
могут использоваться для изменения классов сразу после их создания или до-  
бавлять дополнительный слой логики уже после создания экземпляров. При   
применении декоратора к классу программный код вида:  
def decorator(aClass): ...  
   
@decorator  
class C: ...  
отображается в следующий эквивалент:  
def decorator(aClass): ...  
   
class C: ...  
C = decorator(C)  
Декоратор класса может расширить функциональность самого класса или вер-  
нуть объект, который будет перехватывать последующие попытки конструи-  
рования экземпляров. Так, в примере из раздела «Подсчет экземпляров для   
каждого класса с помощью методов класса» выше, мы могли бы использовать   
этот прием для автоматического добавления в классы счетчика экземпляров   
и любых других необходимых данных:

900   
Глава 31. Дополнительные возможности классов   
def count(aClass):  
 aClass.numInstances = 0  
 return aClass # Возвращает сам класс, а не обертку  
   
@count  
class Spam: ... # То же, что и Spam = count(Spam)  
   
@count  
class Sub(Spam): ... # Инструкция numInstances = 0 не нужна здесь  
   
@count  
class Other(Spam): ...  
Метаклассы представляют собой похожий инструмент на основе классов, об-  
ласть применения которого отчасти перекрывает область применения декора-  
торов классов. Они предоставляют альтернативную модель управления созда-  
нием объектов классов за счет создания подклассов класса type и включения их   
в инструкцию class:  
class Meta(type):  
 def \_\_new\_\_(meta, classname, supers, classdict): ...  
   
class C(metaclass=Meta): ...   
В Python 2.6 результат получается тем же, но способ включения метакласса от-  
Python 2.6 результат получается тем же, но способ включения метакласса от-  
 2.6 результат получается тем же, но способ включения метакласса от-  
личается – вместо именованного аргумента в заголовке инструкции class для   
этих целей используется атрибут metaclass:  
class C:  
 \_\_metaclass\_\_ = Meta  
 ...  
Обычно метакласс переопределяет метод \_\_new\_\_ или \_\_init\_\_ класса type, с це-  
лью взять на себя управление созданием или инициализацией нового объек-  
та класса. Как и при использовании декораторов классов, суть состоит в том,   
чтобы определить программный код, который будет вызываться автоматиче-  
ски на этапе создания класса. Оба способа позволяют расширять классы или   
возвращать произвольные объекты для его замены – протокол с практически   
неограниченными возможностями.  
Дополнительная информация  
Естественно, декораторы и метаклассы имеют намного больше свойств и осо-  
бенностей, чем я показал здесь. Декораторы и метаклассы являют собой уни-  
версальный механизм, и тем не менее, эта дополнительная особенность пред-  
ставляет интерес в первую очередь для разработчиков инструментальных   
средств, а не для прикладных программистов, поэтому я отложу подробное их   
описание до заключительной части книги:  
 •  
В главе 37 демонстрируется определение свойств с помощью декораторов.  
 •  
В главе 38 подробнее рассказывается о декораторах и приводятся более пол-  
ные примеры.  
 •  
В главе 39 описываются метаклассы и подробнее рассказывается об управ-  
лении классами и экземплярами.

Типичные проблемы при работе с классами   
901  
В этих главах не только рассматриваются более сложные темы, в них также   
предоставляется шанс увидеть работу интерпретатора на более интересных   
примерах, чем в остальной части книги.  
Типичные проблемы при работе с классами  
Большая часть типичных проблем, связанных с классами, сводится к пробле-  
мам, связанным с пространствами имен (особенно если учесть, что классы –   
это всего лишь пространства имен с некоторыми дополнительными особенно-  
стями). Некоторые темы, которые мы затронем в этом разделе, скорее являют-  
ся передовыми приемами использования классов, чем проблемами, а решение   
одной-двух из этих проблем было упрощено в последних версиях Python.  
Изменение атрибутов класса   
может приводить к побочным эффектам  
Теоретически классы (и экземпляры классов) относятся к категории изменяе-  
мых объектов. Подобно таким встроенным типам, как списки и словари, они   
могут изменяться непосредственно, путем присваивания значений атрибутам,   
и как и в случае со списками и словарями, это означает, что изменение класса   
или экземпляра может оказывать влияние на множественные ссылки на них.  
Обычно это именно то, что нам требуется (так объекты изменяют свое состо-  
яние), но, изменяя атрибуты, об этом необходимо помнить. Все экземпляры   
класса совместно используют одно и то же пространство имен класса, поэто-  
му любые изменения на уровне класса будут отражаться на всех экземплярах,   
если, конечно, они не имеют собственных версий атрибутов класса.  
Классы, модули и экземпляры – это всего лишь объекты с пространствами   
имен атрибутов, поэтому во время выполнения они обычно изменяются с по-  
мощью операций присваивания. Рассмотрим следующий класс. В теле класса   
выполняется присваивание имени a, в результате чего создается атрибут X.a,   
который во время выполнения располагается в объекте класса и будет унасле-  
дован всеми экземплярами класса X:  
>>> class X:  
... a = 1 # Атрибут класса  
...  
>>> I = X()  
>>> I.a # Унаследован экземпляром  
1  
>>> X.a  
1  
Пока все неплохо – это обычный случай. Но обратите внимание, что происхо-  
дит, когда атрибут класса изменяется динамически, за пределами инструкции   
class: это приводит к одновременному изменению атрибута во всех объектах,   
наследующих его от класса. Кроме того, новые экземпляры класса, созданные   
в ходе интерактивного сеанса или во время работы программы, получают ди-  
намически установленное значение, независимо от того, что написано в исход-  
ном программном коде класса:

902   
Глава 31. Дополнительные возможности классов   
>>> X.a = 2 # Может измениться не только в классе X  
>>> I.a # Объект I тоже изменился  
2  
>>> J = X() # J наследует значение, установленное во время выполнения  
>>> J.a # (но присваивание имени J.a изменяет a в J, но не в X или I)  
2  
Что это – полезная особенность или опасная ловушка? Решать вам. Фактиче-  
ски вы можете выполнять все необходимые действия по изменению атрибутов   
класса, не создавая ни единого экземпляра, – с помощью этого приема можно   
имитировать «записи» и «структуры» данных, имеющиеся в других языках   
программирования. Чтобы освежить воспоминания, рассмотрим следующую,   
не совсем обычную, но вполне допустимую программу на языке Python:  
class X: pass # Создать несколько пространств имен атрибутов  
class Y: pass  
   
X.a = 1 # Использовать атрибуты класса как переменные  
X.b = 2 # В программе нет ни одного экземпляра класса  
X.c = 3  
Y.a = X.a + X.b + X.c  
   
for X.i in range(Y.a): print(X.i) # Выведет 0..5  
Здесь классы X и Y действуют как модули «без файлов» – пространства имен для   
хранения переменных, которые не конфликтуют между собой. Это совершенно   
допустимый прием на языке Python, но он не подходит для применения к клас-  
Python, но он не подходит для применения к клас-  
, но он не подходит для применения к клас-  
сам, написанным другими программистами, – вы не всегда можете быть уве-  
рены, что атрибуты класса, которые вы изменяете, не являются критически   
важными для внутренних механизмов класса. Если вы имитируете структуру   
на языке C, лучше изменять экземпляры, а не класс, поскольку в этом случае   
изменения будут касаться единственного объекта:  
class Record: pass  
X = Record()  
X.name = ‘bob’  
X.job = ‘Pizza maker’  
Модификация изменяемых атрибутов класса   
также может иметь побочные эффекты  
Данная проблема в действительности является продолжением предыдущей.   
Атрибуты класса совместно используются всеми его экземплярами, поэтому,   
если атрибут класса ссылается на изменяемый объект, изменение этого объек-  
та из любого экземпляра отразится сразу на всех экземплярах:  
>>> class C:  
... shared = [] # Атрибут класса  
... def \_\_init\_\_(self):  
... self.perobj = [] # Атрибут экземпляра  
...  
>>> x = C() # Два экземпляра  
>>> y = C() # неявно используют один и тот же атрибут класса  
>>> y.shared, y.perobj  
([], [])  
   
>>> x.shared.append(‘spam’) # Окажет влияние на объект y также!

Типичные проблемы при работе с классами   
903  
>>> x.perobj.append(‘spam’) # Изменит данные, принадлежащие только объекту x  
>>> x.shared, x.perobj  
([‘spam’], [‘spam’])  
   
>>> y.shared, y.perobj # В объекте y наблюдаются изменения,   
([‘spam’], []) # произведенные через объект x  
>>> C.shared # Сохраненный в классе и совместно используемый  
[‘spam’]  
Этот случай ничем не отличается от многих других, представленных в этой   
книге: разделяемые объекты, на которые ссылаются несколько простых пере-  
менных, глобальные объекты, которые совместно используются несколькими   
функциями, объекты уровня модуля, которые совместно используются не-  
сколькими импортирующими модулями, и изменяемые аргументы функций,   
которые совместно используются вызывающим и вызываемым программным   
кодом. Все эти случаи являются разновидностями ситуации наличия несколь-  
ких ссылок на изменяемый объект – изменения в объекте, выполненные с по-  
мощью любой из этих ссылок, можно будет наблюдать с помощью всех осталь-  
ных. В данном случае всеми экземплярами совместно используются атрибуты   
класса, через механизм наследования, но по сути это тот же самый феномен,   
который может осложняться различными способами присваивания атрибутам   
экземпляров:  
x.shared.append(‘spam’) # Изменит разделяемый объект, присоединенный к классу  
x.shared = ‘spam’ # Изменит или создаст атрибут экземпляра x  
Но опять-таки это не является проблемой, это всего лишь одна из особенностей   
использования изменяемых объектов в атрибутах классов, о которой следует   
помнить, – разделяемые, изменяемые атрибуты класса вполне могут найти   
применение в программах на языке Python.  
Множественное наследование: порядок имеет значение  
Это достаточно очевидно, но тем не менее стоит подчеркнуть: в случае ис-  
пользования множественного наследования порядок, в котором перечислены   
суперклассы в строке заголовка инструкции class, может иметь критическое   
значение. В ходе поиска интерпретатор всегда просматривает суперклассы сле-  
ва направо, в соответствии с порядком их следования в заголовке инструкции.  
Например, в примере множественного наследования, который был продемон-  
стрирован в главе 30, предположим, что класс Super тоже реализует метод \_\_  
str\_\_:  
class ListTree:  
 def \_\_str\_\_(self): ...  
   
class Super:  
 def \_\_str\_\_(self): ...  
   
class Sub(ListTree, Super): # Будет унаследован метод \_\_str\_\_ класса ListTree,  
 # так как он стоит в списке первым  
x = Sub() # Поиск сначала будет выполняться в классе   
 # ListTree, а затем в классе Super  
От какого класса мы унаследовали бы метод – от класса Lister или Super? Это   
зависело бы от того, какой класс стоит первым в заголовке объявления клас-  
са Sub, так как поиск унаследованных атрибутов производится слева направо.

904   
Глава 31. Дополнительные возможности классов   
Очевидно, мы поставили бы класс ListTree первым в списке, потому что его   
основная цель состоит в предоставлении метода \_\_str\_\_ (в действительности,   
мы сделали нечто похожее в главе 30, смешав этот класс с классом Button из   
библиотеки tkinter, который имеет собственный метод \_\_str\_\_).  
А теперь предположим, что классы Super и ListTree имеют свои собственные   
версии еще одного одноименного атрибута. Если необходимо, чтобы одно имя   
наследовалось от класса Super, а другое от класса ListTree, изменение порядка   
их расположения в заголовке инструкции определения подкласса уже не по-  
может – мы должны вручную переопределить результат наследования, явно   
выполнив присваивание имени атрибута в классе Sub:  
class ListTree:  
 def \_\_str\_\_(self): ...  
 def other(self): ...  
class Super:  
 def \_\_str\_\_(self): ...  
 def other(self): ...  
class Sub(ListTree, Super): # Унаследует \_\_str\_\_ класса ListTree, так как он   
 # первый в списке  
 other = Super.other # Явно выбирается версия атрибута из класса Super  
 def \_\_init\_\_(self):  
 ...  
x = Sub() # Поиск сначала выполняется в Sub и только потом в ListTree/Super  
Здесь присваивание атрибуту с именем other в классе Sub создает атрибут Sub.  
other – ссылку на объект Super.other. Поскольку данная ссылка находится ниже   
в дереве классов, это не позволит механизму наследования выбрать версию   
атрибута ListTree.other, который был обнаружен первым при обычных обстоя-  
тельствах. Точно так же, если бы класс Super стоял первым в списке, то что-  
бы атрибут other наследовался обычным образом, нам могло бы потребоваться   
явно выбрать методы класса ListTree:  
class Sub(Super, ListTree): # Получить Super.other по наследованию  
 \_\_str\_\_ = ListTree.\_\_str\_\_ # Явно выбрать ListTree.\_\_str\_\_  
Множественное наследование – это довольно сложная тема. Даже если вы по-  
няли предыдущий абзац, все равно этот прием лучше использовать осторож-  
но и только в случае крайней необходимости. В противном случае могут воз-  
никать ситуации, когда значение имени атрибута будет зависеть от порядка   
следования классов в инструкции определения подкласса. (Еще один пример   
этого приема в действии приводится в этой же главе в разделе «Классы нового   
стиля», где обсуждалось явное разрешение конфликтов имен.)  
Как правило, множественное наследование дает лучшие результаты, когда   
суперклассы являются максимально автономными, – поскольку они могут   
использоваться в разных контекстах, они не должны делать каких-либо пред-  
положений об именах, связанных с другими классами в дереве. Псевдочаст-  
ные атрибуты с именами вида \_\_X, которые рассматривались в главе 30, могут   
помочь в локализации имен, на владение которыми опирается класс, и огра-  
ничить вероятность появления конфликтов имен в суперклассах, которые вы   
добавляете в список наследуемых классов. Например, в данном случае класс   
ListTree служит только для того, чтобы экспортировать метод \_\_str\_\_, поэтому

Типичные проблемы при работе с классами   
905  
он мог бы дать своему второму методу имя \_\_other, чтобы избежать конфликтов   
с именами в других классах.  
Методы, классы и вложенные области видимости  
Эта проблема была ликвидирована в Python 2.2 введением областей видимо-  
сти вложенных функций, но я сохранил это описание исключительно ради   
истории, для тех из вас, кому приходилось работать с более старыми версия-  
ми Python и с целью продемонстрировать, что происходит в случае вложения   
функций, когда один из уровней вложенности является классом.  
Классы, как и функции, обладают своими локальными областями видимости,   
поэтому области видимости обладают сходными проявлениями в теле инструк-  
ции class. Кроме того, методы, по сути, являются вложенными функциями,   
поэтому здесь имеют место те же самые проблемы. Похоже, что путаница осо-  
бенно часто возникает, когда имеются классы, вложенные друг в друга.  
В следующем примере (файл nester.py) функция generate возвращает экземпляр   
вложенного класса Spam. Внутри этой функции имя класса Spam находится в ло-  
кальной области видимости функции generate. Но в версиях Python, появив-  
шихся до версии 2.2, внутри метода method имя класса Spam недоступно – method   
имеет доступ только к своей локальной области видимости, к области види-  
мости модуля, вмещающего окружающую функцию generate, и к встроенным   
именам:  
def generate(): # Терпит неудачу при выполнении под управлением   
 class Spam: # Python до версии 2.2  
 count = 1  
 def method(self): # Имя Spam недоступно:  
 print(Spam.count)# Не локальное (def), не глобальное (модуль),   
 # не встроенное  
 return Spam()  
   
generate().method()  
   
C:\python\examples> python nester.py  
...текст сообщения об ошибке опущен...  
 print(Spam.count) # Не локальное (def), не глобальное (модуль),   
NameError: Spam # не встроенное  
Этот пример будет работать в версии Python 2.2 и выше, потому что все локаль-  
ные области вмещающих функций автоматически видимы для вложенных   
функций (включая и вложенные методы, как в данном примере). Но он не ра-  
ботал в версиях Python, вышедших до версии 2.2 (некоторые возможные реше-  
ния приводятся ниже).  
Обратите внимание, что даже в версии 2.2 методам недоступна локальная об-  
ласть видимости вмещающего класса – им доступны только области видимо-  
сти вмещающих функций. Именно по этой причине методы должны исполь-  
зовать аргумент self с экземпляром или имя класса, чтобы вызывать другие   
методы или обращаться к другим атрибутам, определенным во вмещающей   
инструкции class. Например, программный код метода не может использовать   
простое имя count, он должен использовать имя self.count или Spam.count.  
Если вам приходится работать с версией ниже 2.2, скажу, что существует не-  
сколько способов заставить предыдущий пример работать. Самый простой за-  
ключается в том, чтобы переместить имя Spam в область видимости вмещаю-

906   
Глава 31. Дополнительные возможности классов   
щего модуля с помощью глобального объявления. Поскольку методу method   
доступны глобальные имена в модуле, попытка сослаться на Spam уже не будет   
вызывать ошибку:  
def generate():  
 global Spam # Перенести имя Spam в область видимости модуля  
 class Spam:  
 count = 1  
 def method(self):  
 print(Spam.count) # Работает: глобальное имя (вмещающий модуль)  
 return Spam()  
   
generate().method() # Выведет 1  
Лучше было бы реструктурировать программный код так, чтобы вместо ис-  
пользования объявления global определение класса Spam находилось на верх-  
нем уровне модуля. После этого вложенный метод method и функция generate   
будут отыскивать класс Spam в глобальной области видимости:  
def generate():  
 return Spam()  
   
class Spam: # Определение на верхнем уровне в модуле  
 count = 1  
 def method(self):  
 print(Spam.count) # Работает: глобальное имя (вмещающий модуль)  
   
generate().method()  
В действительности такой подход рекомендуется использовать во всех версиях   
Python – программный код выглядит проще, если в нем отсутствуют вложен-  
ные классы и функции.  
Если вам требуется нечто более сложное и замысловатое, то можно просто из-  
бавиться от ссылки на имя Spam в методе method, используя специальный атри-  
бут \_\_class\_\_, который возвращает класс объекта экземпляра:  
def generate():  
 class Spam:  
 count = 1  
 def method(self):  
 print(self.\_\_class\_\_.count) # Работает: используется атрибут для   
 return Spam() # получения класса  
   
generate().method()  
Делегирование в версии 3.0:   
\_\_getattr\_\_ и встроенные операции  
Мы сталкивались с этой проблемой в главе 27, когда изучали практический   
пример программирования классов и когда рассматривали прием делегиро-  
вания в главе 30: классы, использующие метод \_\_getattr\_\_ для делегирования   
обернутым объектам операций обращения к атрибутам, будут терпеть неуда-  
чу в Python 3.0, если методы перегрузки операторов не будут переопределены   
в классе-обертке. В Python 3.0 (и в 2.6, при использовании классов нового сти-  
Python 3.0 (и в 2.6, при использовании классов нового сти-  
 3.0 (и в 2.6, при использовании классов нового сти-  
ля) обращения к методам перегрузки операторов производятся встроенными   
операциями неявно, минуя обычную схему выбора методов-обработчиков. На-

В заключение   
907  
пример, метод \_\_str\_\_, используемый для вывода, никогда не вызывает \_\_ge-  
ge-  
tattr\_\_. Вместо этого в Python 3.0 интерпретатор пытается отыскать требуе-  
Python 3.0 интерпретатор пытается отыскать требуе-  
 3.0 интерпретатор пытается отыскать требуе-  
мые имена в классах, пропуская этап поиска в экземпляре. Чтобы решить эту   
проблему, подобные методы должны быть переопределены в классах-обертках   
(вручную, с помощью других инструментов или с помощью суперклассов). Мы   
еще вернемся к этой проблеме в главах 37 и 38.  
«Многослойное обертывание»   
При грамотном использовании способность объектно-ориентированного про-  
граммного кода к многократному использованию поможет существенно сни-  
зить затраты времени на его разработку. Однако иногда неправильное исполь-  
зование потенциала абстракции ООП может серьезно осложнить понимание   
программного кода. Если классы наслоены друг на друга слишком глубоко,   
программный код становится малопонятным – возможно, вам придется изу-  
чить множество классов, чтобы выяснить, что делает единственная операция.  
Например, однажды мне пришлось работать с библиотекой, написанной на   
языке C++, содержащей тысячи классов (часть которых была сгенерирована   
машиной) и до 15 уровней наследования. Расшифровка вызовов методов в та-  
кой сложной системе классов часто оказывалась неподъемной задачей: даже   
в простейшую операцию оказывались вовлеченными сразу несколько классов.   
Логика системы оказалась такой многослойной, что в некоторых случаях, что-  
бы понять принцип действия какого-либо участка программного кода, требо-  
валось несколько дней копаться в нескольких файлах.  
Здесь также вполне применимо одно из самых универсальных правил языка   
Python: не усложняйте решение задачи, если оно не является таковым. Обер-  
тывание программного кода несколькими слоями классов на грани непости-  
жимости – всегда плохая идея. Абстракция – это основа полиморфизма и ин-  
капсуляции, и при грамотном использовании она может быть весьма эффек-  
тивным инструментом. Однако вы упростите отладку и сопровождение, если   
сделаете интерфейсы своих классов интуитивно понятными. Избегайте чрез-  
мерной абстракции и сохраняйте иерархии своих классов короткими и плоски-  
ми, если не существует веских причин сделать иначе.  
В заключение  
В этой главе было представлено несколько расширенных возможностей клас-  
сов, включая наследование встроенных типов, классы нового стиля, статиче-  
ские методы и декораторы. Большинство из них являются необязательными   
расширениями модели ООП в языке Python, но они могут стать более полезны-  
ми, когда вы начнете создавать крупные объектно-ориентированные програм-  
мы. Как уже упоминалось ранее, обсуждение некоторых из наиболее сложных   
особенностей классов будет продолжено в заключительной части книги – не   
стесняйтесь заглядывать вперед, если вам потребуется более подробная инфор-  
мация о свойствах, дескрипторах и метаклассах.  
Это конец части, посвященной классам, поэтому ниже вы найдете обычные   
в этом случае упражнения – обязательно проработайте их, чтобы получить не-  
которую практику создания настоящих классов. В следующей главе мы нач-  
нем изучение последней базовой темы – исключений. Исключения – это ме-  
ханизм взаимодействий с ошибками и другими ситуациями, возникающими

908   
Глава 31. Дополнительные возможности классов   
в программном коде. Это относительно несложная тема, но я оставил ее напо-  
следок, потому что в настоящее время исключения оформлены в виде классов.   
Но прежде чем заняться этой последней темой, ознакомьтесь с контрольными   
вопросами к этой главе и выполните упражнения.  
Закрепление пройденного  
Контрольные вопросы  
1. Назовите два способа расширения встроенных типов.  
2. Для чего используются декораторы функций?  
3. Как создать класс нового стиля?  
4. Чем отличаются классы нового стиля и классические классы?  
5. Чем отличаются обычные и статические методы?  
6. Сколько секунд нужно выждать, прежде чем бросить «Пресвятую Ручную   
Гранату»?  
Ответы  
1. Можно заключать встроенные классы в классы-обертки или наследовать   
встроенные типы в подклассах. Последний вариант проще, так как в этом   
случае подклассы наследуют большую часть поведения оригинальных   
классов.  
2. Декораторы функций обычно используются для добавления дополнитель-  
ного слоя логики к существующим функциям, который запускается при   
каждом вызове функции. Они могут использоваться для регистрации вы-  
зовов этих функций, проверки типов передаваемых аргументов и так далее.   
Кроме того, они используются для «объявления» статических методов –   
простых функций в классе, которым не передается экземпляр класса.  
3. Классы нового стиля создаются наследованием встроенного класса object   
(или любого другого встроенного типа). В Python 3.0 все классы по умол-  
чанию являются классами нового стиля, поэтому наследовать класс object   
необязательно. В версии 2.6 классы, наследующие класс object, являются   
классами нового стиля, а не наследующие его являются «классическими»   
классами.  
4. При использовании ромбоидальной схемы наследования в классах нового   
стиля поиск в дереве наследования выполняется иначе – сначала произ-  
водится поиск в ширину, а не в высоту. Кроме того, классы нового стиля   
изменяют результат вызова встроенной функции type для экземпляров   
и классов, при выполнении встроенных операций не используют обычную   
схему поиска атрибутов с привлечением метода \_\_getattr\_\_ и поддерживают   
ряд новых дополнительных особенностей, включая свойства, дескрипторы   
и список атрибутов экземпляра \_\_slots\_\_.  
5. Обычные методы (методы экземпляра) принимают аргумент self (под-  
разумеваемый экземпляр), а статические методы – нет. Статические мето-  
ды – это простые функции, вложенные в объект класса. Чтобы превратить   
обычный метод в статический, его необходимо передать специальной встро-  
енной функции, или декоратору, с использованием правил декорирования.

Закрепление пройденного   
909  
Python 3.0 позволяет добавлять в определение класса простые функции, ко-  
торые впоследствии будут вызываться через имя класса, минуя этот шаг, од-  
нако, если такие методы предполагается вызывать относительно экземпля-  
ра, декорирование статических методов остается необходимым условием.  
6. Три секунды. (Или, если быть более точным:1 «И сказал Господь: Допре-  
же всего Пресвятую Чеку извлечь долженствует. Опосля же того, сочти до   
трех, не более и не менее. Три есть цифирь, до коей счесть потребно, и со-  
чтенья твои суть три. До четырех счесть не моги, паче же до двух, опричь   
токмо коли два предшествует трем. О пяти и речи быть не может. Аще же до-  
стигнешь ты цифири три, что есть и пребудет третьею цифирью, брось Пре-  
святою Антиохийскою Гранатою твоею во врага твоего, и оный враг, будучи   
ничтожен пред лицем моим, падёт.»)2  
Упражнения к шестой части  
В этих упражнениях вам будет предложено написать несколько классов и поэк-  
спериментировать с существующим программным кодом. Единственная про-  
блема существующего кода состоит в том, что он должен существовать. Чтобы   
поэкспериментировать с набором классов в упражнении 5, вам нужно либо за-  
грузить файл с исходными текстами с веб-сайта книги (читайте предисловие),   
или ввести его вручную (он достаточно короткий). Поскольку программы ста-  
новятся все сложнее, обязательно ознакомьтесь с решениями в конце книги.   
Решения вы найдете в приложении B, в разделе «Часть VI, Классы и ООП».  
1. Наследование. Напишите класс с именем Adder, экспортирующий метод   
add(self, x, y), который выводит сообщение «Not Implemented» («Не реали-  
Not Implemented» («Не реали-  
 Implemented» («Не реали-  
Implemented» («Не реали-  
» («Не реали-  
зовано»). Затем определите два подкласса класса Adder, которые реализуют   
метод add:  
ListAdder  
С методом add, который возвращает результат конкатенации двух спи-  
сков из аргументов.  
DictAdder  
С методом add, который возвращает новый словарь, содержащий элемен-  
ты из обоих словарей, передаваемых как аргументы (подойдет любое   
определение сложения).  
Поэкспериментируйте с экземплярами всех трех классов в интерактивной   
оболочке, вызывая их методы add.  
Теперь расширьте суперкласс Adder, добавив сохранение объекта в экзем-  
пляре с помощью конструктора (например, присваивая список или словарь   
атрибуту self.data), и реализуйте перегрузку оператора + с помощью метода   
\_\_add\_\_ так, чтобы он автоматически вызывал ваш метод add (например, вы-  
ражение X + Y должно приводить к вызову метода X.add(X.data, Y)). Где лучше   
разместить методы конструктора и перегрузки оператора (то есть в каком из   
классов)? Объекты какого типа смогут складывать ваши классы?  
1   
Цитата из фильма «Monty Python and the Holy Grail».  
2   
 Этот фильм имеется в переводе на русский язык, вышел под названием «Монти Пай-  
тон и Священный Грааль». Перевод взят из Википедии. – Прим. перев.

910   
Глава 31. Дополнительные возможности классов   
На практике гораздо проще написать метод add, который принимает один   
действительный аргумент (например, add(self, y)) и складывает его с те-  
кущими данными экземпляра (например, self.data + y). Будет ли в такой   
реализации больше смысла, чем в реализации, которая принимает два ар-  
гумента? Можно ли сказать, что это делает ваши классы более «объектно-  
ориентированными»?  
2. Перегрузка операторов. Напишите класс с именем MyList, который «обер-  
тывает» списки языка Python: он должен перегружать основные операторы   
действий над списками, включая +, доступ к элементам по индексу, итера-  
ции, извлечение среза и такие методы списка, как append и sort. Полный   
перечень методов, поддерживаемых списками, вы найдете в справочном ру-  
ководстве по языку Python. Кроме того, напишите конструктор для своего   
класса, который принимает существующий список (или экземпляр класса   
MyList) и копирует его в атрибут экземпляра. Поэкспериментируйте со сво-  
им классом в интерактивной оболочке. В ходе экспериментов выясните сле-  
дующее:  
a. Почему здесь так важно копировать начальное значение?  
b. Можно ли использовать пустой срез (например, start[:]) для копирова-  
ния начального значения, если им является экземпляр MyList?  
c. Существует ли универсальный способ передачи управления методам   
обернутого списка?  
d. Можно ли складывать MyList и обычный список? А список и MyList?  
e. Объект какого типа должны возвращать операции сложения и извлече-  
ния среза? А операции извлечения элементов по индексу?  
f. Если у вас достаточно новая версия Python (2.2 или выше), вы сможе-  
те реализовать такого рода класс-обертку, встраивая настоящий список   
в отдельный класс или наследуя класс list. Какой из двух способов про-  
ще и почему?  
3. Подклассы. Напишите подкласс с именем MyListSub, наследующий класс   
MyList из упражнения 2, который расширяет класс MyList возможностью   
вывода сообщения на stdout перед выполнением каждой перегруженной   
операции и подсчета числа вызовов. Класс MyListSub должен наследовать   
методы MyList. При сложении MyListSub с последовательностями должно вы-  
водиться сообщение, увеличиваться счетчик вызовов операции сложения   
и вызываться метод суперкласса. Кроме того, добавьте новый метод, кото-  
рый будет выводить счетчики операций на stdout, и поэкспериментируйте   
с этим классом в интерактивной оболочке. Как работают ваши счетчики –   
считают ли они операции для всего класса (для всех экземпляров класса)   
или для каждого экземпляра в отдельности? Как бы вы реализовали каж-  
дый из этих случаев? (Подсказка: зависит от того, в каком объекте произво-  
дится присваивание значения счетчика: атрибут класса используется все-  
ми экземплярами, а атрибуты аргумента self хранят данные экземпляра.)  
4. Методы метакласса. Напишите класс с именем Meta с методами, которые   
перехватывают все обращения к атрибутам (как получение значения, так   
и присваивание) и выводят сообщения, перечисляющие их аргументы, на   
stdout. Создайте экземпляр класса Meta и поэкспериментируйте с ним в ин-  
терактивной оболочке. Что произойдет, если попытаться использовать эк-  
земпляр класса в выражении? Попробуйте выполнить над своим классом

Закрепление пройденного   
911  
операции сложения, доступа к элементам по индексу и получения среза.   
(Примечание: самый типичный способ, основанный на использовании \_\_ge-  
tattr\_\_, будет действовать в 2.6, но не будет в 3.0, по причинам, упомянутым   
в главе 30 и вновь приведенным в решении этого упражнения.)  
5. Объекты множеств. Поэкспериментируйте с набором классов, описанных   
в разделе «Расширение типов встраиванием». Выполните команды, кото-  
рые выполняют следующие операции:  
 •  
Создайте два множества целых чисел и найдите их пересечение и объе-  
динение с помощью операторов & и |.  
 •  
Создайте множество из строки и поэкспериментируйте с извлечением   
элементов множества по индексу. Какой метод в классе при этом вызы-  
вается?  
 •  
Попробуйте выполнить итерации через множество, созданное из строки,   
с помощью цикла for. Какой метод вызывается на этот раз?  
 •  
Попробуйте найти пересечение и объединение множества, созданного из   
строки, и простой строки. Возможно ли это?  
 •  
Теперь расширьте класс множества наследованием, чтобы подкласс мог   
обрабатывать произвольное число операндов, используя для этого форму   
аргумента \*args. (Подсказка: вернитесь к рассмотрению этих алгоритмов   
в главе 18.) Найдите пересечение и объединение нескольких операндов   
с помощью вашего подкласса множества. Как можно реализовать вычис-  
ление пересечения трех и более множеств, если оператор & работает всего   
с двумя операндами?  
 •  
Как бы вы реализовали другие операции над списками в классе множе-  
ства? (Подсказка: метод \_\_add\_\_ перехватывает операцию конкатена-  
ции, а метод \_\_getattr\_\_ может передавать большинство вызовов методов   
списка в обернутый список.)  
6. Связи в дереве классов. В разделе «Пространства имен: окончание истории»   
в главе 28 и в разделе «Множественное наследование: примесные классы»   
в главе 30 я упоминал, что классы имеют атрибут \_\_bases\_\_, который воз-  
вращает кортеж объектов суперклассов (тех, что перечислены в круглых   
скобках в заголовке инструкции class). Используя атрибут \_\_bases\_\_, рас-  
ширьте классы в файле lister.py (глава 30) так, чтобы они выводили имена   
прямых суперклассов экземпляров класса. При этом первая строка в этом   
выводе должна выглядеть, как показано ниже (значение адреса у вас может   
отличаться):  
<Instance of Sub(Super, ListTree), address 7841200:  
7. Композиция. Сымитируйте сценарий оформления заказа в ресторане бы-  
строго питания, определив четыре класса:  
Lunch  
Вмещающий и управляющий класс.  
Customer  
Действующее лицо, покупающее блюдо.  
Employee  
Действующее лицо, принимающее заказ.

912   
Глава 31. Дополнительные возможности классов   
Food  
То, что приобретает заказчик.  
Чтобы вам было с чего начать, определите следующие классы и методы:  
class Lunch:  
 def \_\_init\_\_(self) # Создает и встраивает Customer и Employee  
 def order(self, foodName) # Имитирует прием заказа  
 def result(self) # Запрашивает у клиента название блюда  
   
class Customer:  
 def \_\_init\_\_(self) # Инициализирует название блюда значением None  
 def placeOrder(self, foodName, employee) # Передает заказ официанту  
 def printFood(self) # Выводит название блюда  
   
class Employee:  
 def takeOrder(self, foodName) # Возвращает блюдо с указанным названием  
   
class Food:  
 def \_\_init\_\_(self, name) # Сохраняет название блюда  
Имитация заказа работает следующим образом:  
a. Конструктор класса Lunch должен создать и встроить экземпляр класса   
Customer и экземпляр класса Employee, а кроме того, экспортировать ме-  
тод с именем order. При вызове этот метод должен имитировать прием   
заказа у клиента (Customer) вызовом метода placeOrder. Метод placeOrder   
класса Customer должен в свою очередь имитировать получение блю-  
да (новый объект Food) у официанта (Employee) вызовом метода takeOrder   
класса Employee.  
b. Объекты типа Food должны сохранять строку с названием блюда (напри-  
мер, «буррито»), которое передается через Lunch.order в Customer.placeOr-  
der, затем в Employee.takeOrder и, наконец, в конструктор класса Food. Кро-  
ме того, класс Lunch должен еще экспортировать метод result, который   
предлагает клиенту (Customer) вывести название блюда, полученного от   
официанта (Employee) в результате выполнения заказа (этот метод может   
использоваться для проверки имитации).  
Обратите внимание: экземпляр класса Lunch должен передавать клиенту   
(Customer) либо экземпляр класса Employee (официант), либо себя самого, что-  
бы клиент (Customer) мог вызвать метод официанта (Employee).  
Поэкспериментируйте с получившимися классами в интерактивной обо-  
лочке, импортируя класс Lunch и вызывая его метод order, чтобы запустить   
имитацию, а также метод result, чтобы проверить, что клиент (Customer)   
получил именно то, что заказывал. При желании можете добавить в файл   
с классами программный код самотестирования, используя прием с атри-  
бутом \_\_name\_\_ из главы 24. В этой имитации активность проявляет клиент   
(Customer)� как бы вы изменили свои классы, чтобы инициатором взаимодей-  
ствий между клиентом и официантом был официант (Employee)?  
8. Классификация животных в зоологии. Изучите дерево классов, представ-  
ленное на рис. 31.1. Напишите шесть инструкций class, которые имитиро-  
вали бы эту модель классификации средствами наследования в языке Py-  
Py-  
thon. Затем добавьте к каждому из классов метод speak, который выводил   
бы уникальное сообщение, и метод reply в суперклассе Animal, являющемся   
вершиной иерархии, который просто вызывал бы self.speak, чтобы вывести

Закрепление пройденного   
913  
текст сообщения, характерного для каждой категории, в подклассах, рас-  
положенных ниже (это вынудит начинать поиск в дереве наследования от   
экземпляра self). Наконец, удалите метод speak из класса Hacker, чтобы для   
него по умолчанию выводилось сообщение, унаследованное от класса выше.   
Когда вы закончите, ваши классы должны работать следующим образом:  
% python  
>>> from zoo import Cat, Hacker  
>>> spot = Cat()  
>>> spot.reply() # Animal.reply; вызывается Cat.speak  
meow  
>>> data = Hacker() # Animal.reply; вызывается Primate.speak  
>>> data.reply()  
Hello world!  
9. Сценка с мертвым попугаем. Изучите схему встраивания объектов, пред-  
ставленную на рис. 31.2. Напишите набор классов на языке Python, кото-  
Python, кото-  
, кото-  
рые реализовали бы эту схему средствами композиции. Класс Scene (сцена)   
должен определять метод action и встраивать в себя экземпляры классов   
Customer (клиент), Clerk (клерк) и Parrot (попугай), каждый из которых дол-  
Parrot (попугай)  
Scene (сцена)  
Customer (клиент)  
Clerk (клерк)  
action  
line  
Рис. 31.2. Составная сцена, представленная управляющим классом (Scene),   
который встраивает и управляет экземплярами трех других классов    
(Customer, Clerk, Parrot). Встроенные экземпляры классов могут также   
участвовать в иерархии наследования – композиция и наследование часто   
являются одинаково полезными способами организации классов с целью обе-  
спечения возможности повторного использования программного кода  
Hacker (хакер)  
Animal (животное)  
Mammal (млекопитающее)  
Cat (кошка)  
Dog (собака)  
Primate (примат)  
Рис. 31.1. Классификация животных в зоологии, составленная из классов,   
связанных в дерево наследования. Класс Animal имеет общий метод «reply»,   
но каждый из классов имеет свой собственный метод «speak», который вы-  
speak», который вы-  
», который вы-  
зывается методом «reply»

914   
Глава 31. Дополнительные возможности классов   
жен определять метод line, выводящий уникальное сообщение. Встраива-  
емые объекты могут наследовать один общий суперкласс, определяющий   
метод line, который просто выводит текст указанного ему сообщения, или   
определяют собственные реализации метода line. В конечном итоге ваши   
классы должны действовать, как показано ниже:  
% python  
>>> import parrot  
>>> parrot.Scene().action() # Активировать встроенные объекты  
customer: “that’s one ex-bird!”  
clerk: “no it isn’t...”  
parrot: None  
Придется держать в уме: ООП глазами специалистов  
Когда я рассказываю о классах в языке Python, я все время обнаружи-  
ваю, что в середине лекции о классах люди, имевшие опыт ООП в про-  
шлом, заметно активизируются, а те, кто такого опыта не имеет, начи-  
нают сникать (или вообще засыпают). Преимущества этой технологии   
не так очевидны.  
В такой книге, как эта, у меня есть уникальная возможность включить   
обзорный материал, которой я воспользовался в главе 25, – настоятель-  
но рекомендую вам перечитать эту главу, как только вам начинает ка-  
заться, что ООП – это всего лишь некоторое украшение в программиро-  
вании.  
В реальной аудитории, чтобы привлечь (и удержать) внимание начи-  
нающих программистов, я обычно останавливаюсь и спрашиваю у при-  
сутствующих опытных специалистов, почему они используют ООП. От-  
веты, которые они дают, могут пролить свет на цели, которые преследу-  
ет ООП, для тех, кто плохо знаком с этой темой.  
Ниже приводятся лишь самые общие причины, побуждающие исполь-  
зовать ООП, которые были высказаны моими студентами за эти годы:  
Повторное использование программного кода  
Это самая простая (и самая основная) причина использования ООП.   
Возможность наследования в классах позволяет программисту пи-  
сать программы, адаптируя существующий программный код, а не   
писать каждый новый проект с самого начала.  
Инкапсуляция  
Сокрытие деталей реализации за интерфейсом объекта предохраня-  
ет пользователей класса от необходимости изменять свой программ-  
ный код.  
Организация   
Классы предоставляют новые локальные области видимости, кото-  
рые минимизируют вероятность конфликтов имен. Кроме того, они   
обеспечивают место для естественного размещения программного   
кода реализации и управления состоянием объекта.

Закрепление пройденного   
915  
Поддержка  
Классы обеспечивают естественное разделение программного кода,   
что позволяет уменьшить его избыточность. Благодаря организации   
и возможности повторного использования программного кода в слу-  
чае необходимости бывает достаточно изменить всего одну копию   
программного кода.  
Непротиворечивость  
Классы и возможность наследования позволяют реализовать общие   
интерфейсы и, следовательно, обеспечить единообразие вашего про-  
граммного кода – такой код легко поддается отладке, выглядит бо-  
лее осмысленно и прост в сопровождении.  
Полиморфизм  
Это скорее свойство ООП, чем причина его использования, но благо-  
даря поддержке общности программного кода полиморфизм делает   
код более гибким, расширяет область его применения и, следова-  
тельно, увеличивает его шансы на повторное использование.  
Другие  
И конечно, причина номер один состоит в том, что упоминание о вла-  
дении приемами ООП увеличивает шанс быть принятым на работу!   
(Согласен, я привел эту причину в шутку, но если вы собираетесь ра-  
ботать на ниве программирования, для вас очень важно будет иметь   
знакомство с ООП.)  
И в заключение, не забывайте, что я говорил в начале шестой части: вы   
не сможете полностью оценить достоинства ООП, пока не будете исполь-  
зовать его какое-то время. Выберите себе проект, изучите большие при-  
меры, поработайте над упражнениями – это заставит вас попотеть над   
объектно-ориентированным программным кодом, но оно стоит того.

Часть VII.  
Исключения и инструменты

Глава 32.  
   
Основы исключений  
В этой части книги рассказывается об исключениях, которые, по сути, явля-  
ются событиями, способными изменить ход выполнения программы. Исклю-  
чения в языке Python возбуждаются автоматически, когда программный код   
допускает ошибку, а также могут возбуждаться и перехватываться самим про-  
граммным кодом. Обрабатываются исключения четырьмя инструкциями. Эти   
инструкции мы и будем изучать в данной части книги. Первая из инструкций   
имеет две разновидности (ниже они перечислены отдельно), а последняя – яв-  
ляется дополнительным расширением до выхода версий Python 2.6 и 3.0:   
try/except  
Перехватывает исключения, возбужденные интерпретатором или вашим   
программным кодом, и выполняет восстановительные операции.  
try/finally  
Выполняет заключительные операции независимо от того, возникло ис-  
ключение или нет.  
raise  
Дает возможность возбудить исключение программно.  
assert  
Дает возможность возбудить исключение программно, при выполнении   
определенного условия.  
with/as  
Реализует менеджеры контекста в версиях Python 2.6 и 3.0 (в версии 2.5 яв-  
Python 2.6 и 3.0 (в версии 2.5 яв-  
 2.6 и 3.0 (в версии 2.5 яв-  
ляется дополнительным расширением).  
Эта тема была оставлена напоследок потому, что для работы с исключения-  
ми необходимо знание классов. Тем не менее за несколькими исключениями   
(преднамеренная игра слов), как будет показано ниже, обработка исключений   
в языке Python выполняется очень просто, потому что они интегрированы не-  
Python выполняется очень просто, потому что они интегрированы не-  
 выполняется очень просто, потому что они интегрированы не-  
посредственно в сам язык, как и другие высокоуровневые средства.

920   
Глава 32. Основы исключений   
Зачем нужны исключения?  
В двух словах, исключения позволяют перепрыгнуть через фрагмент програм-  
мы произвольной длины. Рассмотрим пример с машиной по приготовлению   
пиццы, о которой говорилось ранее в этой книге. Предположим, что мы более   
чем серьезно отнеслись к этой идее и действительно построили такую машину.   
Чтобы приготовить пиццу, наш кулинарный автомат должен выполнить про-  
грамму, написанную на языке Python: она должна принимать заказ, пригото-  
Python: она должна принимать заказ, пригото-  
: она должна принимать заказ, пригото-  
вить тесто, выбрать добавки, выпечь основу и так далее.  
Теперь предположим, что что-то пошло совсем не так во время «выпекания   
основы». Возможно, сломалась печь или, возможно, наш робот ошибся в рас-  
четах расстояния до печи и воспламенился. Совершенно очевидно, что нам не-  
обходимо предусмотреть быстрый переход к программному коду, который бы-  
стро обрабатывает такие ситуации. Кроме того, поскольку в таких необычных   
условиях у нас нет никакой надежды на успешное окончание процесса приго-  
товления пиццы, мы могли бы также вообще отказаться от выполнения всего   
плана целиком.  
Это именно то, что позволяют делать исключения: программа может перейти   
к обработчику исключения за один шаг, отменив все вызовы функций. После   
этого обработчик исключения может выполнить действия, соответствующие   
ситуации (например, вызвать пожарную охрану!).  
Исключение – это своего рода «супер-goto». Обработчик исключений (инструк-  
ция try) ставит метку и выполняет некоторый программный код. Если затем   
где-нибудь в программе возникает исключение, интерпретатор немедленно   
возвращается к метке, отменяя все активные вызовы функций, которые были   
произведены после установки метки. Такой подход позволяет соответствую-  
щим способом реагировать на необычные события. Кроме того, переход к об-  
работчику исключения выполняется немедленно, поэтому обычно нет никакой   
необходимости проверять коды возврата каждой вызванной функции, которая   
могла потерпеть неудачу.  
Назначение исключений  
В программах на языке Python исключения могут играть разные роли. Ниже   
приводятся некоторые из них, являющиеся наиболее типичными:  
Обработка ошибок  
Интерпретатор возбуждает исключение всякий раз, когда обнаружива-  
ет ошибку во время выполнения программы. Программа может перехва-  
тывать такие ошибки и обрабатывать их или просто игнорировать. Если   
ошибка игнорируется, интерпретатор выполняет действия, предусмотрен-  
ные по умолчанию, – он останавливает выполнение программы и выводит   
сообщение об ошибке. Если такое поведение по умолчанию является неже-  
лательным, можно добавить инструкцию try, которая позволит перехваты-  
вать обнаруженные ошибки и продолжить выполнение программы после   
инструкции try.  
Уведомления о событиях  
Исключения могут также использоваться для уведомления о наступлении   
некоторых условий, что устраняет необходимость передавать куда-либо

Обработка исключений: краткий обзор   
921  
флаги результата или явно проверять их. Например, функция поиска мо-  
жет возбуждать исключение в случае неудачи, вместо того чтобы возвра-  
щать целочисленный признак в виде результата (и надеяться, что этот при-  
знак всегда будет интерпретироваться правильно).  
Обработка особых ситуаций  
Некоторые условия могут наступать так редко, что было бы слишком рас-  
точительно предусматривать проверку наступления таких условий с целью   
их обработки. Нередко такие проверки необычных ситуаций можно заме-  
нить обработчиками исключений.  
Заключительные операции  
Как будет показано далее, инструкция try/finally позволяет гарантировать   
выполнение завершающих операций независимо от наличия исключений.  
Необычное управление потоком выполнения  
И, наконец, так как исключения – это своего рода оператор «goto», их мож-  
но использовать как основу для экзотического управления потоком выпол-  
нения программы. Например, обратная трассировка не является частью   
самого языка, но она может быть реализована с помощью исключений   
и некоторой логики поддержки, выполняющей раскручивание операций   
присваивания.1 В языке Python отсутствует оператор «goto» (к счастью!), но   
исключения иногда могут с успехом заменить его.  
Далее в этой части книги мы увидим примеры этих типичных применений.   
А пока начнем с обзора средств языка Python, предназначенных для обработки   
исключений.  
Обработка исключений: краткий обзор  
В сравнении с некоторыми другими основными возможностями, которые рас-  
сматривались в этой книге, исключения в языке Python представляют собой   
чрезвычайно легкий инструмент. Поскольку они так просты, перейдем сразу   
к первому примеру.   
Обработчик исключений по умолчанию  
Предположим, что у нас имеется следующая функция:  
>>> def fetcher(obj, index):  
... return obj[index]  
...  
1   
Настоящая обратная трассировка – это довольно сложная тема, и данная возможность   
не является частью самого языка Python (даже функции-генераторы и выражения-  
генераторы, с которыми мы встречались в главе 20, не являются настоящей обратной   
трассировкой – они просто отвечают на вызов функции next(G)). Грубо говоря, обрат-  
ная трассировка отменяет все вычисления перед переходом – исключения этого не де-  
лают (то есть в переменных, которым было выполнено присваивание между моментом   
выполнения инструкции try и до момента возбуждения исключения, прежние значе-  
ния не восстанавливаются). Если вам любопытна эта тема, обращайтесь к книгам по   
искусственному интеллекту или языкам программирования Prolog или Icon.

922   
Глава 32. Основы исключений   
Эта функция делает не так много – она просто извлекает элемент из объекта по   
заданному индексу. При нормальном стечении обстоятельств она возвращает   
результат:  
>>> x = ‘spam’  
>>> fetcher(x, 3) # Все равно, что x[3]  
‘m’  
Однако если передать функции индекс, выходящий за пределы строки, то при   
попытке выполнить выражение obj[index] будет возбуждено исключение. Об-  
наруживая выход за пределы последовательности, интерпретатор сообщает об   
этом, возбуждая встроенное исключение IndexError:  
>>> fetcher(x, 4) # Обработчик по умолчанию – интерактивная оболочка  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
 File “<stdin>”, line 2, in fetcher  
IndexError: string index out of range  
(IndexError: выход индекса за пределы диапазона)  
Поскольку наш программный код не перехватывает это исключение явно, оно   
возвращает выполнение на верхний уровень программы и вызывает обработ-  
чик исключений по умолчанию, который просто выводит стандартное сообще-  
ние об ошибке. К настоящему моменту вы наверняка видели в своих програм-  
мах подобные сообщения об ошибках. Они включают тип исключения, а также   
диагностическую информацию – список строк и функций, которые были ак-  
тивны в момент появления исключения.   
Текст сообщения об ошибке, который приводится выше, был получен в Py-  
Py-  
thon 3.0 – он может несколько отличаться в разных версиях интерпретатора.   
При работе в интерактивной оболочке файлом является «<stdin>», то есть стан-  
дартный поток ввода. При работе в IDLE файлом является «<pyshell>» и до-  
полнительно выводятся номера строк. В любом случае номера строк не несут   
сколько-нибудь полезной информации (далее в этой книге вы увидите куда бо-  
лее интересные сообщения об ошибках):  
>>> fetcher(x, 4) # Обработчик по умолчанию - IDLE GUI  
Traceback (most recent call last):  
 File “<pyshell#6>”, line 1, in <module>  
 fetcher(x, 4)  
 File “<pyshell#3>”, line 2, in fetcher  
 return obj[index]  
IndexError: string index out of range  
В настоящей программе, запущенной не в интерактивной оболочке, после вы-  
вода сообщения обработчик по умолчанию сразу же завершает работу про-  
граммы. Такое действие имеет смысл для простых сценариев – как правило,   
ошибки в таких сценариях должны быть фатальными и лучшее, что можно   
сделать при их появлении, – это ознакомиться с текстом сообщения.  
Обработка исключений  
Иногда это совсем не то, что нам требуется. Например, серверные программы   
обычно должны оставаться активными даже после появления внутренних   
ошибок. Если вам требуется избежать реакции на исключение по умолчанию,

Обработка исключений: краткий обзор   
923  
достаточно просто перехватить исключение, обернув вызов функции инструк-  
цией try:  
>>> try:  
... fetcher(x, 4)  
... except IndexError: # Перехватывает и обрабатывает исключение  
... print(‘got exception’)  
...  
got exception  
>>>  
Теперь, когда исключение будет возникать при выполнении инструкций в бло-  
ке try, интерпретатор будет автоматически переходить к вашему обработчику   
(блок под предложением except, в котором указано имя исключения). При ра-  
боте в интерактивной оболочке, как в примере выше, после выполнения блока   
except происходит возврат в приглашение к вводу. В настоящих программах   
инструкции try не только перехватывают исключения, но и выполняют дей-  
ствия по восстановлению после ошибок:  
>>> def catcher():  
... try:  
... fetcher(x, 4)  
... except IndexError:  
... print(‘got exception’)  
... print(‘continuing’)  
...  
>>> catcher()  
got exception  
continuing  
>>>  
На этот раз после того как исключение было перехвачено и обработано, про-  
грамма продолжила выполнение ниже всей инструкции try – именно поэтому   
в данном примере было выведено сообщение «continuing». Стандартное сооб-  
continuing». Стандартное сооб-  
». Стандартное сооб-  
щение об ошибке не появилось на экране, и программа продолжила работу как   
ни в чем не бывало.  
Возбуждение исключений  
До сих пор все исключения, которые мы наблюдали, возбуждались интерпре-  
татором, когда он встречал наши ошибки (на сей раз ошибка была допущена   
нарочно!), однако наши сценарии также способны возбуждать исключения – то   
есть исключения могут возбуждаться интерпретатором или самой программой   
и могут перехватываться или не перехватываться. Чтобы возбудить исключе-  
ние вручную, достаточно просто выполнить инструкцию raise. Исключения,   
определяемые программой, перехватываются точно так же, как и встроенные   
исключения. Следующий фрагмент, возможно, содержит не самый полезный   
программный код, когда-либо написанный, но он проясняет вышесказанное:  
>>> try:  
... raise IndexError # Возбуждает исключение вручную  
... except IndexError:  
... print(‘got exception’)  
...  
got exception

924   
Глава 32. Основы исключений   
Если исключение, определяемое программой, не перехватывается, оно будет   
передано обработчику исключений по умолчанию, что приведет к завершению   
программы с выводом стандартного сообщения об ошибке:   
>>> raise IndexError  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
IndexError  
Как будет показано в следующей главе, исключения могут также возбуждать-  
ся с помощью инструкции assert – это условная форма инструкции raise, кото-  
рая используется в основном для отладки в процессе разработки:  
>>> assert False, ‘Nobody expects the Spanish Inquisition!’  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
AssertionError: Nobody expects the Spanish Inquisition!  
Исключения, определяемые пользователем  
Инструкция raise, представленная в предыдущем разделе, возбуждает встро-  
енные исключения, которые определены во встроенной области видимости.   
Как вы узнаете далее в этой части книги, вы также можете определять новые   
исключения для внутренних нужд своих программ. Пользовательские исклю-  
чения создаются в виде классов, наследующих один из классов встроенных ис-  
ключений, – обычно класс с именем Exception. Исключения на базе классов по-  
зволяют сценариям создавать категории исключений, наследовать поведение   
и добавлять к ним информацию о состоянии:  
>>> class Bad(Exception): # Пользовательское исключение  
... pass  
...  
>>> def doomed():  
... raise Bad() # Возбудит экземпляр исключения  
...  
>>> try:  
... doomed()  
... except Bad: # Перехватить исключение по имени класса  
... print(‘got Bad’)  
...  
got Bad  
>>>  
Заключительные операции  
Наконец, инструкции try могут включать блоки finally. Эти блоки выглядят   
точно так же, как обработчики except. Комбинация try/finally определяет за-  
вершающие действия, которые всегда выполняются «на выходе», независимо   
от того, возникло исключение в блоке try или нет:  
>>> try:  
... fetcher(x, 3)  
... finally: # Заключительные операции  
... print(‘after fetch’)  
...  
‘m’

Обработка исключений: краткий обзор   
925  
after fetch  
>>>  
Здесь, если блок try выполнится без ошибок, будет выполнен блок finally   
и программа продолжит свою работу дальше. В этом случае данная инструк-  
ция кажется бессмысленной – мы могли бы просто добавить инструкцию print   
сразу вслед за вызовом функции и вообще убрать инструкцию try:  
fetcher(x, 3)  
print(‘after fetch’)  
Однако в таком подходе имеется одна проблема: если в функции возникнет   
исключение, инструкция print не будет выполнена. Комбинация try/final-  
ly позволяет ликвидировать эту проблему – когда в блоке try действительно   
произойдет исключение, блок finally будет выполнен, пока программа будет   
раскручиваться:  
>>> def after():  
... try:  
... fetcher(x, 4)  
... finally:  
... print(‘after fetch’)  
... print(‘after try?’)  
...  
>>> after()  
after fetch  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
 File “<stdin>”, line 3, in after  
 File “<stdin>”, line 2, in fetcher  
IndexError: string index out of range  
(IndexError: выход индекса за пределы диапазона)  
>>>  
Здесь мы не получили сообщение «after try?», потому что работа программы не   
была продолжена после блока try/finally, когда возникло исключение. Вместо   
этого интерпретатор выполнил действия, предусмотренные блоком finally, по-  
сле чего исключение достигло предыдущего обработчика (в данном случае – об-  
работчик по умолчанию). Если изменить вызов внутри функции action, чтобы   
он не вызывал исключение, блок finally все равно будет выполнен, но програм-  
ма продолжит работу после выхода из инструкции try:  
>>> def after():  
... try:  
... fetcher(x, 3)  
... finally:  
... print(‘after fetch’)  
... print(‘after try?’)  
...  
>>> after()  
after fetch  
after try?  
>>>  
На практике комбинацию try/except удобно использовать для перехвата и вос-  
становления после исключений, а комбинацию try/finally – в случаях, когда не-  
обходимо гарантировать выполнение заключительных действий независимо от

926   
Глава 32. Основы исключений   
того, возникло исключение в блоке try или нет. Например, комбинацию try/ex-  
cept можно было бы использовать для перехвата ошибок, возникающих в импор-  
тированной библиотеке, созданной сторонним разработчиком, а комбинацию   
try/finally – чтобы гарантировать закрытие файлов и соединений с сервером.   
Некоторые из таких практических примеров будут показаны далее в этой книге.  
Несмотря на то, что эти две комбинации служат двум различным целям, тем не   
менее, начиная с версии Python 2.5, появилась возможность смешивать пред-  
Python 2.5, появилась возможность смешивать пред-  
 2.5, появилась возможность смешивать пред-  
ложения except и finally в одной и той же инструкции try – блок finally бу-  
дет выполняться всегда, независимо от того, было ли перехвачено исключение   
предложением except.  
Как мы узнаем в следующей главе, в Python 2.6 и 3.0 существует альтернатива   
конструкции try/finally, используемая при работе с некоторыми типами объ-  
ектов. Инструкция with/as выполняет логику, предусмотренную реализацией   
объекта, гарантируя выполнение заключительных операций:  
>>> with open(‘lumberjack.txt’, ‘w’) as file: # Всегда закрывает файл   
... file.write(‘The larch!\n’) # при выходе  
Такой подход позволяет сократить объем программного кода, однако он может   
применяться при работе лишь с некоторыми типами объектов, поэтому кон-  
струкция try/finally представляет более универсальный способ, гарантирую-  
щий выполнение заключительных операций. С другой стороны, конструкция   
with/as способна выполнять начальные операции и поддерживает возможность   
определять пользовательскую реализацию управления контекстом.  
Придется держать в уме: проверка ошибок  
Один из способов увидеть, насколько полезными могут быть исключе-  
ния, состоит в том, чтобы сравнить стили программирования на языке   
Python и на языке, не имеющем исключений. Например, если вы хотите   
написать надежную программу на языке C, вам потребуется проверять   
возвращаемые значения или коды состояния после выполнения каждой   
операции, которая может быть выполнена с ошибкой, и передавать ре-  
зультаты проверок в ходе выполнения программы:  
doStuff()  
{ # Программа на языке C  
 if (doFirstThing() == ERROR) # Проверить наличие ошибки  
 return ERROR; # даже если здесь она не обрабатывается  
 if (doNextThing() == ERROR)  
 return ERROR;  
 ...  
 return doLastThing();  
}  
   
main()  
{  
 if (doStuff() == ERROR)  
 badEnding();  
 else  
 goodEnding();  
}

В заключение   
927  
Фактически в настоящих программах на языке C значительная доля   
всего программного кода выполняет проверку наличия ошибок. Но   
в языке Python не требуется так же настойчиво и методично выполнять   
проверки. Достаточно просто обернуть произвольные участки програм-  
мы обработчиками исключений и писать эти участки в предположении,   
что никаких ошибок возникать не будет:  
def doStuff(): # Программный код на языке Python  
 doFirstThing() # Нас не беспокоят возможные исключения,  
 doNextThing() # поэтому можно не выполнять проверку  
 ...  
 doLastThing()  
   
if\_\_name\_\_ == ‘\_\_main\_\_’:  
 try:  
 doStuff() # Здесь нас интересуют возможные результаты,  
 except: # поэтому это единственное место, где нужна проверка  
 badEnding()  
 else:  
 goodEnding  
Так как в случае исключения управление немедленно будет передано   
обработчику, здесь нет никакой необходимости разбрасывать проверки   
по всему программному коду, чтобы обезопасить себя от ошибок. Кро-  
ме того, благодаря тому, что интерпретатор Python автоматически об-  
наруживает ошибки, ваши программы обычно не требуют выполнять   
подобные проверки вообще. Таким образом, исключения позволяют   
в значительной степени игнорировать возможные необычные ситуации   
и отказаться от использования программного кода, выполняющего про-  
верки на наличие ошибок.  
В заключение  
Вот в основном и все, что требуется знать об исключениях, – исключения дей-  
ствительно являются очень простым инструментом.   
Исключения в языке Python – это высокоуровневый инструмент управления   
потоком выполнения. Они могут возбуждаться интерпретатором или самой   
программой – в любом из этих случаев их можно игнорировать (что вызовет   
срабатывание обработчика по умолчанию) или перехватывать с помощью ин-  
струкций try (для обработки в своем программном коде). Инструкция try мо-  
жет использоваться в двух логических разновидностях, которые, начиная   
с версии Python 2.5, могут комбинироваться – одна разновидность выполняет   
обработку исключений, а другая выполняет завершающий программный код   
независимо от того, возникло исключение или нет. Исключения можно воз-  
буждать вручную, с помощью инструкций raise и assert (как встроенные, так   
и новые, которые могут создаваться нами в виде классов) – инструкция with/as   
предоставляет альтернативный способ, гарантирующий выполнение заключи-  
тельных операций для объектов, которые поддерживают такую возможность.  
Далее в этой части книги мы подробнее поговорим о самих инструкциях, ис-  
следуем разные виды предложений, которые могут появляться в инструкции

928   
Глава 32. Основы исключений   
try, и обсудим объекты исключений, основанные на классах. В следующей   
главе мы поближе познакомимся с инструкциями, представленными здесь. Но   
прежде чем вы перевернете страницу, ответьте на несколько контрольных во-  
просов, чтобы освежить знания, полученные в этой главе.   
Закрепление пройденного  
Контрольные вопросы  
1. Назовите три области, где можно было бы использовать операции с исклю-  
чениями.  
2. Что произойдет с программой в случае исключения, если вы не предусмо-  
трите его обработку?  
3. Как можно реализовать восстановление нормальной работы сценария после   
исключения?  
4. Назовите два способа возбуждения исключений в сценариях.  
5. Назовите два способа, с помощью которых можно было бы организовать вы-  
полнение заключительных операций независимо от того, возникло исклю-  
чение или нет.  
Ответы  
1. Операции с исключениями удобно использовать для обработки ошибок, вы-  
полнения заключительных операций и для уведомления о наступивших со-  
бытиях. Кроме того, операции с исключениями могут упростить обработку   
специальных случаев и использоваться для реализации альтернативного   
способа управления потоком выполнения. Вообще, операции с исключения-  
ми могут помочь ликвидировать программный код, реализующий проверку   
ошибок – поскольку все ошибки в конечном итоге передаются обработчи-  
кам, отпадает необходимость проверять результат каждой операции.  
2. Любое неперехваченное исключение в конечном итоге передается обработ-  
чику по умолчанию, который предоставляется интерпретатором. Этот об-  
работчик выводит сообщение об ошибке и завершает программу.  
3. Если вы не желаете, чтобы в случае ошибки выводилось сообщение и про-  
исходило завершение программы, вы можете воспользоваться инструкцией   
try/except, чтобы перехватывать и обрабатывать возникающие исключения.   
После того как исключение будет перехвачено и обработано, оно уничтожа-  
ется, и программа может продолжить свою работу.  
4. Для возбуждения исключений можно использовать инструкции raise и as-  
sert, как если бы они были возбуждены интерпретатором. В принципе, вы   
можете возбудить исключение, выполнив ошибочную операцию, но это   
трудно себе представить явной целью при разработке!  
5. Инструкция try/finally гарантирует выполнение заключительных опера-  
ций после выхода из блока try независимо от того, возникло исключение   
или нет. Инструкция with/as также может использоваться для организации   
выполнения заключительных действий, но только в случаях, когда обраба-  
тываемый объект поддерживает такую возможность.

Глава 33.  
   
Особенности использования   
исключений  
В предыдущей главе мы коротко познакомились с инструкциями, связанны-  
ми с исключениями. Здесь мы займемся более глубоким исследованиями – эта   
глава представляет собой более формальное введение в синтаксис обработки   
исключений в языке Python. В частности, мы исследуем механизмы, стоящие   
за инструкциями try, raise, assert и with. Как вы увидите далее, все эти ин-  
струкции достаточно просты в употреблении, но они представляют собой мощ-  
ные инструменты, предназначенные для работы с исключениями в програм-  
мах на языке Python.  
Предварительное замечание: За последние годы реализация ис-  
ключений сильно изменилась. Начиная с версии Python 2.5, по-  
явилась возможность употреблять предложение finally вместе   
с предложениями except и else в одной инструкции try (ранее это   
было невозможно). Кроме того, в версиях Python 3.0 и 2.6 офи-  
циально была введена новая инструкция with менеджера кон-  
текста, а пользовательские исключения теперь должны созда-  
ваться как экземпляры классов, наследующих один из классов   
встроенных исключений. Помимо этого в версии 3.0 немного из-  
менился синтаксис инструкции raise и предложения except.   
В этом издании книги основное внимание будет уделяться реа-  
лизации механизма исключений в Python 2.6 и 3.0, но так как   
на протяжении еще какого-то времени вы наверняка будете   
встречать оригинальные приемы работы с исключениями, я по-  
путно буду рассказывать, как развивались эти механизмы.  
Инструкция try/except/else  
Теперь, когда вы познакомились с основами, пришло время приступить к ис-  
следованию деталей. В следующем обсуждении я сначала представлю try/ex-  
cept/else и try/finally как разные инструкции, потому что они имеют разное

930   
Глава 33. Особенности использования исключений   
предназначение и не могут комбинироваться в версиях Python ниже, чем 2.5.   
Как уже говорилось, начиная с версии Python 2.5, except и finally могут сме-  
шиваться в одной инструкции try – я объясню суть этого изменения после того   
как будут исследованы две оригинальные формы по отдельности.  
Инструкция try – это составная инструкция. Полная ее форма приводится   
ниже. Она начинается со строки заголовка try, вслед за которой располагает-  
ся блок инструкций (как правило) с отступами, затем следует одно или более   
предложений except, которые определяют обрабатываемые исключения, и за-  
тем следует необязательное предложение else. Слова try, except и else должны   
располагаться с одним и тем же отступом (то есть должны быть выровнены по   
вертикали). Для справки ниже приводится полный формат инструкции:  
try:  
 <statements> # Сначала выполняются эти действия  
except <name1>:  
 <statements> # Запускается, если в блоке try возникло исключение name1  
except (name2, name3):  
 <statements> # Запускается, если возникло любое из этих исключений  
except <name4> as <data>:  
 <statements> # Запускается в случае исключения name4  
 # и получает экземпляр исключения  
except:  
 <statements> # Запускается для всех (остальных) возникших исключений  
else:  
 <statements> # Запускается, если в блоке try не возникло исключения  
В этой инструкции блок под заголовком try представляет основное действие   
инструкции – программный код, который следует попытаться выполнить.   
Предложения except определяют обработчики исключений, возникших в ходе   
выполнения блока try, а предложение else (если присутствует) определяет об-  
работчик для случая отсутствия исключений. Элемент <data> имеет отноше-  
ние к особенности инструкций raise, которая будет обсуждаться далее в этой   
главе.  
Ниже описывается принцип действия инструкции try. Когда запускается ин-  
струкция try, интерпретатор помечает текущий контекст программы, чтобы   
вернуться к нему, если возникнет исключение. В первую очередь выполняются   
инструкции, расположенные под заголовком try. Что произойдет дальше, за-  
висит от того, будет ли возбуждено исключение в блоке try:  
 •  
Если исключение возникнет во время выполнения инструкций в блоке try,   
интерпретатор вернется к инструкции try и выполнит первое предложение   
except, соответствующее возбужденному исключению. После выполнения   
блока except управление будет передано первой инструкции, находящейся   
за всей инструкцией try (при условии, что в блоке except не возникло друго-  
го исключения).  
 •  
Если в блоке try возникло исключение и не было найдено ни одного соот-  
ветствия среди предложений except, исключение будет передано инструк-  
ции try, стоящей выше в программе, или на верхний уровень процесса (что   
вынудит интерпретатор аварийно завершить работу программы и вывести   
сообщение об ошибке по умолчанию).  
 •  
Если в процессе выполнения блока try не возникло исключение, интерпре-  
татор выполнит инструкции в блоке else (если имеются) и затем выполнение   
продолжится с первой инструкции, находящейся за всей инструкцией try.

Инструкция try/except/else   
931  
Другими словами, предложения except перехватывают любые исключения, ко-  
торые могут возникнуть при выполнении блока try, а блок else выполняется   
только в случае отсутствия исключений в блоке try.  
В предложениях except находятся обработчики исключений – они перехваты-  
вают исключения, которые возникли только в инструкциях блока try. Однако   
инструкции в блоке try могут вызывать функции, расположенные в разных   
частях программы, поэтому сам источник исключения может располагаться   
за пределами самой инструкции try. Мы еще поговорим об этом, когда будем   
исследовать вложенные инструкции try в главе 35.  
Предложения инструкции try  
В инструкции try могут присутствовать разные предложения, располагающие-  
ся вслед за блоком try. В табл. 33.1 приводятся все возможные формы, из ко-  
торых хотя бы одна должна присутствовать. Мы уже встречали некоторые из   
них: как вы уже знаете, предложение except перехватывает исключения, пред-  
ложение finally выполняется при выходе из инструкции, а предложение else   
выполняется, когда в блоке try не возникло исключение.   
С точки зрения синтаксиса, в инструкции может присутствовать несколько   
предложений except, но только одно предложение else. Вплоть до версии Py-  
Py-  
thon 2.4 предложение finally должно было быть единственным (без предложе-  
ний else или except). В действительности try/finally – это отдельная инструкция.   
Однако начиная с версии Python 2.5 предложение finally может присутствовать   
в той же инструкции, что и предложения except и else (подробнее о правилах,   
определяющих порядок их следования, будет рассказываться ниже в этой гла-  
ве, когда будет обсуждаться объединенная инструкция try).  
Таблица 33.1. Различные формы предложений в инструкции try  
Форма предложения  
Интерпретация  
except:  
Перехватывает все (остальные) типы   
исключений.  
except name:  
Перехватывает только указанное   
исключение.  
except name as value:  
Перехватывает указанное исключение   
и получает соответствующий экземпляр.  
except (name1, name2):  
Перехватывает любое из перечисленных   
исключений.  
except (name1, name2) as value:  
Перехватывает любое из перечисленных   
исключений и получает соответствующий   
экземпляр.  
else:  
Выполняется, если не было исключений.  
finally:  
Этот блок выполняется всегда.  
Исследованием дополнительного значения as value мы займемся, когда будем   
рассматривать инструкцию raise. Оно обеспечивает доступ к объекту, который   
играет роль исключения.

932   
Глава 33. Особенности использования исключений   
Новыми здесь для нас являются первая и четвертая строки в табл. 33.1:  
 •  
Предложения except, в которых отсутствуют имена исключений (except:),   
перехватывают все исключения, ранее не перечисленные в инструкции try.  
 •  
Предложения except, где в круглых скобках перечислены имена исключе-  
ний (except (e1, e2, e3):), перехватывают любое из перечисленных исключе-  
ний.  
Интерпретатор Python просматривает предложения except сверху вниз в поис-  
ках соответствия, поэтому версию предложения с круглыми скобками можно   
рассматривать как аналог нескольким отдельным выражениям except, по одно-  
му для каждого исключения из списка, только в этом случае тело обработчика   
является общим для всех указанных исключений. Ниже приводится пример   
использования нескольких предложений except, который демонстрирует поря-  
док определения обработчиков:  
try:  
 action()  
except NameError:  
 ...  
except IndexError  
 ...  
except KeyError:  
 ...  
except (AttributeError, TypeError, SyntaxError):  
 ...  
else:  
 ...  
В этом примере, если при выполнении функции action возникает исключение,   
интерпретатор возвращается к инструкции try и пытается отыскать первое   
предложение except, в котором указано возникшее исключение. Поиск среди   
предложений except ведется сверху вниз, слева направо, и выполняются ин-  
струкции в первом найденном совпадении. Если совпадений не будет найдено,   
исключение продолжит распространение выше этой инструкции try. Обрати-  
те внимание, что блок else выполняется только при отсутствии исключения   
в функции action – этот блок не выполняется при наличии исключения, кото-  
рому не было найдено соответствующее предложение except.   
Если вам действительно необходимо организовать перехват всех исключений,   
используйте пустое предложение except:  
try:  
 action()  
except NameError:  
 ... # Обработать исключение NameError  
except IndexError:  
 ... # Обработать исключение IndexError  
except:  
 ... # Обработать все остальные исключения  
else:  
 ... # Обработка случая отсутствия исключений  
Предложение except без имени исключения – это своего рода шаблонный сим-  
вол, потому что оно перехватывает любые исключения, что позволяет вам соз-  
давать и универсальные, и специфичные обработчики по своему усмотрению.

Инструкция try/except/else   
933  
В некоторых случаях эта форма может быть более удобна, чем перечисление   
всех возможных исключений в инструкции try. Так, в следующем примере вы-  
полняется перехват всех исключений:  
try:  
 action()  
except:  
 ... # Перехватить все возможные исключения  
Однако применение пустых предложений except влечет за собой определенные   
проблемы проектирования. Несмотря на удобство, они могут перехватывать   
нежелательные системные исключения, не связанные с работой вашего про-  
граммного кода, и по случайности прерывать распространение исключений,   
предназначенных для других обработчиков. Например, даже выход из про-  
граммы в языке Python возбуждает исключение, и поэтому было бы желатель-  
но, чтобы это исключение было пропущено. Кроме того, такая конструкция бу-  
дет перехватывать исключения, вызванные обычными ошибками программи-  
рования, которые вам наверняка хотелось бы обнаружить. Мы вернемся к этой   
проблеме в конце этой части книги. А пока я скажу лишь, что предложение   
except требует внимательного отношения.  
В Python 3.0 была введена альтернатива, решающая одну из этих проблем, –   
предложение except Exception имеет практически тот же эффект, что и пустое   
предложение except, но оно не перехватывает исключения, имеющие отноше-  
ние к завершению программы:  
try:  
 action()  
except Exception:  
 ... # Перехватит все исключения, кроме завершения программы  
Данная форма обеспечивает практически те же удобства, что и пустое предло-  
жение except, но при этом таит в себе практически те же самые опасности. Мы   
исследуем работу этой формы в следующей главе, когда будем изучать классы   
исключений.  
Примечание,  касающееся  различий  между  версиями: В версии   
Python 3.0 следует использовать форму предложения except E as   
V:, представленную в третьей строке табл. 33.1, вместо более ста-  
рой формы except E, V:. Последнюю форму по-прежнему допуска-  
ется использовать (но не рекомендуется) в Python 2.6: в случае ее   
использования она автоматически преобразуется в первую фор-  
му. Это изменение было внесено с целью ликвидировать возмож-  
ность перепутать более старую форму со случаем, когда указы-  
вается два или более имен альтернативных исключений в виде   
except (E1, E2):. В Python 3.0 поддерживается только форма с ис-  
пользованием ключевого слова as – запятая в предложении ex-  
cept всегда будет означать кортеж исключений независимо от   
наличия круглых скобок, а значения будут интерпретироваться   
как альтернативные исключения, которые требуется перехва-  
тить. Кроме того, были изменены правила видимости: при ис-  
пользовании ключевого слова as переменная V автоматически   
удаляется в конце блока except.

934   
Глава 33. Особенности использования исключений   
Предложение try/else  
Назначение предложения else в инструкции try на первый взгляд не всегда   
очевидно для тех, кто только начинает осваивать язык Python. Тем не менее   
без этого предложения нет никакого другого способа узнать (не устанавливая   
и не проверяя флаги) – выполнение программы продолжилось потому, что ис-  
ключение в блоке try не было возбуждено, или потому, что исключение было   
перехвачено и обработано:  
try:  
 ...выполняемый код...  
except IndexError:  
 ...обработка исключения...  
# Программа оказалась здесь потому, что исключение было обработано  
# или потому, что его не возникло?  
Точно так же, как предложение else в операторах цикла делает причину вы-  
хода из цикла более очевидной, предложение else в инструкции try однозначно   
и очевидно сообщает о произошедшем:  
try:  
 ...выполняемый код...  
except IndexError:  
 ...обработка исключения...  
else:  
 ...исключение не было возбуждено...  
То же самое поведение можно имитировать, переместив содержимое блока   
else в блок try:  
try:  
 ...выполняемый код...  
 ...исключение не было возбуждено...  
except IndexError:  
 ...обработка исключения...  
Но это может привести к некорректной классификации исключения. Если   
какая-либо из инструкций в блоке «исключение не было возбуждено» приведет   
к появлению исключения IndexError, оно будет зарегистрировано как ошибка   
в блоке try и соответственно, ошибочно будет передано обработчику исключе-  
ния ниже (тонко, но верно!). При явном использовании выражения else логика   
выполнения становится более очевидной и гарантируется, что обработчики ис-  
ключений будут вызываться только для обработки истинных ошибок в блоке,   
обернутом инструкцией try, а не при выполнении действий, предусматривае-  
мых в блоке else.  
Пример: поведение по умолчанию  
Поскольку объяснить порядок выполнения программы проще на языке   
Python, чем на естественном языке, рассмотрим несколько примеров, иллю-  
стрирующих основы исключений. Я уже упоминал, что исключения, не пере-  
хваченные инструкциями try, распространяются до самого верхнего уровня   
процесса и запускают логику обработки исключений по умолчанию (то есть   
интерпретатор аварийно завершает работающую программу и выводит стан-

Инструкция try/except/else   
935  
дартное сообщение об ошибке). Рассмотрим пример. При попытке запустить   
следующий модуль bad.py возникает исключение деления на ноль:  
def gobad(x, y):  
 return x / y  
   
def gosouth(x):  
 print(gobad(x, 0))  
   
gosouth(1)  
Так как программа сама не обрабатывает это исключение, интерпретатор за-  
вершает ее и выводит сообщение:  
% python bad.py  
Traceback (most recent call last):  
 File “bad.py”, line 7, in <module>  
 gosouth(1)  
 File “bad.py”, line 5, in gosouth  
 print(gobad(x, 0))  
 File “bad.py”, line 2, in gobad  
 return x / y  
ZeroDivisionError: int division or modulo by zero  
(ZeroDivisionError: целочисленное деление или деление по модулю на ноль)  
Я запускал этот пример под управлением Python 3.0. Сообщение состоит из со-  
Python 3.0. Сообщение состоит из со-  
 3.0. Сообщение состоит из со-  
держимого стека вызовов («Traceback») и имени (с дополнительными данны-  
Traceback») и имени (с дополнительными данны-  
») и имени (с дополнительными данны-  
ми) исключения. В содержимом стека перечислены все строки, которые были   
активны в момент появления исключения, в порядке от более старых к более   
новым. Обратите внимание: так как в данном случае мы работаем в командной   
строке системы, а не в интерактивной оболочке интерпретатора, имена файлов   
и номера строк содержат полезную для нас информацию. Например, здесь вид-  
но, что ошибка произошла во 2-й строке в файле bad.py в инструкции return.1  
Так как интерпретатор Python определяет и сообщает обо всех ошибках, поя-  
Python определяет и сообщает обо всех ошибках, поя-  
 определяет и сообщает обо всех ошибках, поя-  
вившихся во время выполнения программы, возбуждая исключения, эти ис-  
ключения тесно связаны с идеями обработки ошибок и отладки вообще. Если   
вы работали с примерами из этой книги, вы без сомнений встречались с не-  
сколькими исключениями – даже опечатки нередко приводят к возбуждению   
исключения SyntaxError или других при импортировании и выполнении файла   
(то есть, когда запускается компилятор). По умолчанию интерпретатор выво-  
дит полезные информативные сообщения, как показано выше, которые позво-  
ляют легко отыскать источник проблем.  
Нередко стандартные сообщения об ошибках – это все, что необходимо для   
разрешения проблем в программном коде. Для более надежной отладки своих   
программ вы можете перехватывать исключения с помощью инструкций try   
или использовать средства отладки, которые будут представлены в главе 35   
(такие как модуль pdb из стандартной библиотеки).  
1   
Как уже упоминалось в предыдущей главе, текст сообщений и отладочная информа-  
ция могут изменяться в зависимости от версии интерпретатора и от используемой   
оболочки. Поэтому не надо беспокоиться, если ваши сообщения не соответствуют   
в точности тем, что приводятся здесь. Например, когда я запускал этот пример в сре-  
де IDLE, входящей в состав Python 3.0, в тексте сообщения выводились полные пути   
к файлам.

936   
Глава 33. Особенности использования исключений   
Пример: перехват встроенных исключений  
Обработка исключений, которая выполняется интерпретатором по умолчанию,   
зачастую удовлетворяет всем нашим потребностям, особенно для программно-  
го кода верхнего уровня, где ошибки должны приводить к немедленному завер-  
шению программы. Для большинства программ нет никакой необходимости   
предусматривать какие-то особые варианты обработки ошибок.  
Однако иногда бывает необходимо перехватить ошибку и выполнить восстано-  
вительные действия после нее. Если для вас нежелательно, чтобы программа   
завершалась, когда интерпретатор возбуждает исключение, достаточно просто   
перехватить его, обернув участок программы в инструкцию try. Это очень важ-  
ная возможность для таких программ, как серверы сети, которые должны про-  
должать работать постоянно. Например, следующий фрагмент перехватывает   
и обрабатывает исключение TypeError, которое возбуждается интерпретатором   
при попытке выполнить операцию конкатенации для списка и строки (опера-  
тор + требует, чтобы слева и справа были указаны последовательности одного   
и того же типа):  
def kaboom(x, y):  
 print(x + y) # Возбуждает исключение TypeError  
   
try:  
 kaboom([0,1,2], “spam”)  
except TypeError: # Исключение перехватывается и обрабатывается здесь  
 print(‘Hello world!’)  
print(‘resuming here’) # Программа продолжает работу независимо от того,   
 # было ли исключение или нет  
Когда в функции kaboom возникает исключение, управление передается пред-  
ложению except в инструкции try, где выводится текст сообщения. После того,   
как исключение перехватывается, оно становится неактуальным, поэтому   
программа продолжает выполнение ниже инструкции try вместо того, чтобы   
завершиться. Программный код действительно обрабатывает и ликвидирует   
ошибку:  
% python kaboom.py  
Hello world!  
resuming here  
Обратите внимание: как только ошибка будет перехвачена, выполнение про-  
должается с того места, где ошибка была перехвачена (то есть после инструк-  
ции try), – нет никакой возможности вернуться к тому месту, где возникла   
ошибка (в данном случае – в функцию kaboom). В некотором смысле это делает   
исключения более похожими на инструкции перехода, чем на вызовы функ-  
ций, – нет никакой возможности вернуться к программному коду, вызвавшему   
ошибку.  
Инструкция try/finally  
Другая разновидность инструкции try специализируется на выполнении за-  
ключительных операций. Если в инструкцию try включено предложение fi-

Инструкция try/finally   
937  
nally, интерпретатор всегда будет выполнять этот блок инструкций при «вы-  
ходе» из инструкции try независимо от того, произошло ли исключение во вре-  
мя выполнения инструкций в блоке try. Общая форма этой инструкции имеет   
следующий вид:  
try:  
 <statements> # Выполнить эти действия первыми  
finally:  
 <statements> # Всегда выполнять этот блок кода при выходе  
При использовании этой инструкции интерпретатор Python в первую очередь   
выполняет инструкции в блоке try. Что произойдет дальше, зависит от того,   
возникло ли исключение в блоке try:  
 •  
Если во время выполнения инструкций в блоке try исключение не возник-  
ло, интерпретатор переходит к выполнению блока finally и затем продол-  
жает выполнять программу ниже инструкции try.  
 •  
Если во время выполнения инструкций в блоке try возникло исключение,   
интерпретатор также выполнит инструкции в блоке finally, но после этого   
исключение продолжит свое распространение до вышестоящей инструкции   
try или до обработчика исключений по умолчанию – программа не будет   
выполняться вслед за инструкцией try. То есть инструкции в блоке finally   
будут выполнены, даже если исключение будет возбуждено, но в отличие   
от предложения except, предложение finally не завершает распространение   
исключения – оно остается актуальным после выполнения блока finally.  
Форму try/finally удобно использовать, когда необходимо гарантировать вы-  
полнение некоторых действий независимо от реакции программы на исклю-  
чение. С практической точки зрения, эта форма инструкции позволяет опреде-  
лять завершающие действия, которые должны выполняться всегда, такие как   
закрытие файлов или закрытие соединений с сервером.  
Обратите внимание: в Python 2.4 и в более ранних версиях предложение fi-  
nally не может использоваться в той же инструкции try, где уже используется   
предложение except или else, поэтому форму try/finally лучше считать отдель-  
ной формой инструкции при работе со старыми версиями. Однако в Python 2.5   
предложение finally может присутствовать в инструкции try вместе с пред-  
ложениями except и else, поэтому в настоящее время существует единая ин-  
струкция try, которая может употребляться с несколькими необязательными   
предложениями (вскоре мы поговорим об этом подробнее). Какую бы версию   
Python вы не использовали, назначение предложения funally остается преж-  
ним – определить завершающие действия, которые должны выполняться всег-  
да, независимо от возникновения исключений.  
Как будет показано далее в этой главе, в версии Python 2.6 и 3.0   
инструкция with и контекстные менеджеры обеспечивают объ-  
ект но-ориентированный подход к выполнению аналогичных за-  
вершающих действий. Но, в отличие от finally, эта новая ин-  
струкция поддерживает возможность выполнения действий по   
инициализации, хотя и ограничивается областью видимости   
объектов, которые реализуют протокол менеджеров контекста.

938   
Глава 33. Особенности использования исключений   
Пример: реализация завершающих действий   
с помощью инструкции try/finally  
Выше мы видели несколько простых примеров применения инструкции try/  
finally. Ниже приводится более близкий к действительности пример, иллю-  
стрирующий типичное применение этой инструкции:  
class MyError(Exception): pass  
   
def stuff(file):  
 raise MyError()  
   
file = open(‘data’, ‘w’) # Открыть файл для вывода  
try:  
 stuff(file) # Возбуждает исключение  
finally:  
 file.close() # Всегда закрывать файл, чтобы вытолкнуть буферы  
print('not reached') # Продолжить с этого места,   
 # только если не было исключения  
В этом фрагменте мы обернули вызов функции в инструкцию try с предложе-  
нием finally, чтобы гарантировать, что файл будет закрыт при любых обстоя-  
тельствах, независимо от того, будет возбуждено исключение в функции или   
нет. При таком подходе расположенный далее программный код может быть   
уверен, что содержимое выходных буферов файла было вытолкнуто из памяти   
на диск. Подобная структура программного кода может гарантировать закры-  
тие соединения с сервером и так далее.  
Как мы узнали в главе 9, объекты файлов автоматически закрываются на   
этапе сборки мусора, что особенно удобно при работе с временными файлами,   
которые не присваиваются каким-либо переменным. Однако далеко не всегда   
можно предсказать, когда будет выполняться сборка мусора, особенно в круп-  
ных программах. Инструкция try позволяет сделать операцию закрытия фай-  
ла более явной, предсказуемой и принадлежащей определенному блоку про-  
граммного кода. Она гарантирует, что файл будет закрыт при выходе из блока,   
независимо от того, произошло исключение или нет.  
Функция в этом примере не делает ничего полезного (она просто возбуждает   
исключение), но обернув ее в инструкцию try/finally, мы гарантируем, что   
действия по завершению будут выполняться всегда. Напомню еще раз, что ин-  
терпретатор всегда выполняет программный код в блоке finally независимо от   
того, было возбуждено исключение в блоке try или нет.1   
Когда функция в этом примере возбуждает исключение, управление переда-  
ется обратно инструкции try и начинает выполняться блок finally, в котором   
производится закрытие файла. После этого исключение продолжает свое рас-  
пространение либо пока не встретит другую инструкцию try, либо пока не бу-  
дет достигнут обработчик по умолчанию, который выведет стандартное сооб-  
1   
Если, конечно, сам интерпретатор не завершит свою работу аварийно. Разработчики   
Python упорно трудятся над тем, чтобы избежать подобного развития событий, про-  
веряя все возможные ошибки во время работы. Полное обрушение программы вместе   
с интерпретатором часто происходит из-за ошибок в расширениях, написанных на   
языке C, которые выполняются не под управлением Python.

Объединенная инструкция try/except/finally   
939  
щение об ошибке и остановит работу программы – инструкция, находящаяся   
ниже инструкции try, никогда не будет достигнута. Если бы функция в этом   
примере не возбуждала исключение, программа точно так же выполнила бы   
блок finally, чтобы закрыть файл, и затем продолжила бы свое выполнение   
ниже инструкции try.  
Кроме того, обратите внимание, что здесь исключение опять определено как   
класс – как будет показано в следующей главе, в версиях Python 2.6 и 3.0 все   
исключения должны быть классами.  
Объединенная инструкция try/except/finally  
Во всех версиях Python, вышедших до версии 2.5 (в течение первых 15 лет   
жизни или что-то около того), инструкция try существовала в двух разновид-  
ностях, и в действительности имелось две отдельные инструкции. Мы могли   
либо использовать предложение finally, чтобы гарантировать выполнение за-  
вершающего программного кода, либо писать блоки except, чтобы перехваты-  
вать определенные исключения и выполнять действия по восстановлению по-  
сле них и при желании использовать предложение else, которое выполняется   
в случае отсутствия исключений.  
То есть предложение finally нельзя было смешивать с предложениями except   
и else. Такое положение дел сохранялось отчасти из-за проблем с реализацией,   
а отчасти из-за неясности смысла такого смешивания – перехват и восстанов-  
ление после исключений выглядит никак не связанным с выполнением заклю-  
чительных операций.  
Однако в Python 2.5 (а также в Python 2.6 и 3.0, которые описываются в этой   
книге) две инструкции были объединены. Сейчас у нас имеется возможность   
смешивать предложения funally, except и else в одной и той же инструкции. То   
есть теперь можно написать инструкцию, имеющую следующий вид:  
try: # Объединенная форма  
 основное действие  
except Exception1:  
 обработчик1  
except Exception2:  
 обработчик2  
...  
else:  
 блок else  
finally:  
 блок finally  
Первым, как обычно, выполняется программный код в блоке основное действие.   
Если при выполнении этого блока возбуждается исключение, выполняется   
проверка всех блоков except, одного за другим, в поисках блока, соответствую-  
щего возникшему исключению. Если было возбуждено исключение Exception1,   
будет выполнен блок обработчик1, исключение Exception2 приведет к запуску   
обработчика2 и так далее. Если исключение не было возбуждено, будет выполнен   
блок else.  
Независимо от того, что происходило раньше, блок finally будет выполнен   
только после выполнения основных действий и после обработки любых воз-

940   
Глава 33. Особенности использования исключений   
никших исключений. В действительности, блок finally будет выполнен, даже   
если исключение возникнет в самом обработчике исключения или в блоке else.  
Как всегда, предложение finally не прекращает распространение исключе-  
ния – если к моменту выполнения блока finally имеется активное исключение,   
оно продолжает свое распространение после выполнения блока finally и управ-  
ление передается куда-то в другое место программы (другой инструкции try   
или обработчику по умолчанию). Если к моменту, когда блок finally будет вы-  
полнен, нет активного исключения, выполнение программы продолжится сра-  
зу же вслед за инструкцией try.  
Таким образом, блок finally выполняется всегда, когда:  
 •  
В блоке основного действия возникло исключение и было обработано.  
 •  
В блоке основного действия возникло исключение и не было обработано.  
 •  
В блоке основного действия не возникло исключение.  
 •  
В одном из обработчиков возникло новое исключение.  
Напомню еще раз, предложение finally служит, чтобы организовать выполне-  
ние завершающих действий, которые должны выполняться всегда при выходе   
из инструкции try независимо от того, было ли возбуждено исключение и было   
ли оно обработано.  
Синтаксис объединенной инструкции try  
Инструкция try, как минимум, должна содержать либо предложение except,   
либо предложение finally, и составляющие ее части должны следовать в таком   
порядке:  
try -> except -> else -> finally  
где предложения else и finally являются необязательными и может присут-  
ствовать ноль или более предложений except, но в случае присутствия пред-  
ложения else должно быть указано хотя бы одно предложение except. В дей-  
ствительности инструкция try состоит из двух частей: из предложений except   
с необязательным предложением else и/или предложения finally.  
Фактически более правильным будет изобразить синтаксис объединенной ин-  
струкции, как показано ниже (квадратные скобки означают, что заключенное   
в них предложение является необязательным, а звездочка означает «ноль или   
более раз»):  
try: # Формат 1  
 statements  
except [type [as value]]: # [type [, value]] в Python 2  
 statements  
[except [type [as value]]:  
 statements]\*  
[else:  
 statements]  
[finally:  
 statements]  
   
try: # Формат 2  
 statements  
finally:  
 statements

Объединенная инструкция try/except/finally   
941  
Согласно этим правилам предложение else может присутствовать, только если   
в инструкции присутствует хотя бы одно предложение except, и всегда допу-  
скается одновременно указывать предложения except и finally, независимо от   
присутствия предложения else. Кроме того, допускается одновременно указы-  
вать предложения finally и else, но только если в инструкции присутствует   
предложение except (при этом допускается указывать предложение except без   
имени перехватываемого исключения, чтобы перехватывать любые исключе-  
ния и запускать инструкцию raise, которая описывается ниже, чтобы повтор-  
но возбудить текущее обрабатываемое исключение). Если порядок следования   
предложений в инструкции будет нарушен, интерпретатор возбудит исключе-  
ние, свидетельствующее о синтаксической ошибке, еще до того, как программ-  
ный код будет выполнен.  
Объединение finally и except вложением  
До появления версии Python 2.5 существовала возможность объединять пред-  
ложения finally и except в инструкции try за счет вложения инструкции try/ex-  
cept в блок try инструкции try/finally (более полно этот прием будет рассматри-  
ваться в главе 35). В действительности фрагмент ниже имеет тот же эффект,   
что и новая форма инструкции, представленная в начале этого раздела:  
try: # Вложенные инструкции, эквивалентные объединенной форме  
 try:  
 основное действие  
 except Exception1:  
 обработчик1  
 except Exception2:  
 обработчик2  
 ...  
 else:  
 нет ошибок  
finally:  
 завершающие действия  
Здесь также блок finally всегда выполняется при выходе из инструкции try не-  
зависимо от того, что произошло в блоке основного действия, и независимо от   
того, выполнялись ли обработчики исключений во вложенной инструкции try   
(представьте, как в этом случае будут развиваться четыре варианта событий,   
перечисленные выше, и вы увидите, что все будет выполняться точно так же).   
Поскольку предложение else всегда требует наличия хотя бы одного предложе-  
ния except, эта вложенная форма имеет те же ограничения, что и объединенная   
форма инструкции, представленная в предыдущем разделе.   
Однако этот эквивалент выглядит менее понятным, чем новая, объединенная,   
форма инструкции, и для ее записи требуется больше программного кода (по   
крайней мере, на одну четырехсимвольную строку). Смешанная форма ин-  
струкции проще в написании и выглядит понятнее, поэтому такая форма за-  
писи считается в настоящее время предпочтительной.  
Пример использования объединенной инструкции try  
Ниже приводится демонстрационный пример использования объединенной   
формы инструкции try. В следующем файле mergedexc.py представлены четы-

942   
Глава 33. Особенности использования исключений   
ре типичных варианта с инструкциями print, описывающими значение каж-  
дого из них:  
sep = ‘-’ \* 32 + ‘\n’  
print(sep + ‘EXCEPTION RAISED AND CAUGHT’)  
try:  
 x = ‘spam’[99]  
except IndexError:  
 print(‘except run’)  
finally:  
 print(‘finally run’)  
print(‘after run’)  
   
print(sep + ‘NO EXCEPTION RAISED’)  
try:  
 x = ‘spam’[3]  
except IndexError:  
 print(‘except run’)  
finally:  
 print(‘finally run’)  
print(‘after run’)  
   
print(sep + ‘NO EXCEPTION RAISED, WITH ELSE’)  
try:  
 x = ‘spam’[3]  
except IndexError:  
 print(‘except run’)  
else:  
 print(‘else run’)  
finally:  
 print(‘finally run’)  
print(‘after run’)  
   
print(sep + ‘EXCEPTION RAISED BUT NOT CAUGHT’)  
try:  
 x = 1 / 0  
except IndexError:  
 print(‘except run’)  
finally:  
 print(‘finally run’)  
print(‘after run’)  
После запуска под управлением Python 3.0 этот пример выводит на экран сле-  
дующий ниже текст (фактически при запуске под управлением Python 2.6 вы-  
водятся те же результаты, потому что каждый вызов функции print выводит   
единственный элемент). Исследуйте программный код, чтобы понять, как ра-  
ботает каждый из вариантов:  
c:\misc> C:\Python30\python mergedexc.py  
------------------------------  
EXCEPTION RAISED AND CAUGHT  
except run  
finally run  
after run  
------------------------------  
NO EXCEPTION RAISED  
finally run  
after run

Инструкция raise   
943  
------------------------------  
NO EXCEPTION RAISED, WITH ELSE  
else run  
finally run  
after run  
------------------------------  
EXCEPTION RAISED BUT NOT CAUGHT  
finally run  
   
Traceback (most recent call last):  
 File “mergedexc.py”, line 36, in <module>  
 x = 1 / 0  
ZeroDivisionError: int division or modulo by zero  
(ZeroDivisionError: целочисленное деление или деление по модулю на ноль)  
Этот пример для возбуждения исключений в основном действии использует   
встроенные операции и полагается на тот факт, что интерпретатор всегда опре-  
деляет появление ошибок во время выполнения программного кода. В следую-  
щем разделе будет показано, как возбуждать исключения вручную.  
Инструкция raise  
Чтобы явно возбудить исключение, можно использовать инструкцию raise.   
В общем виде она имеет очень простую форму записи – инструкция raise со-  
стоит из слова raise, за которым может следовать имя класса или экземпляр   
возбуждаемого исключения:  
raise <instance> # Возбуждает экземпляр класса-исключения  
raise <class> # Создает и возбуждает экземпляр класса-исключения  
raise # Повторно возбуждает самое последнее исключение  
Как уже упоминалось ранее, исключение в Python 2.6 и 3.0 – это всегда эк-  
земпляр класса. Следовательно, первая форма инструкции raise является наи-  
более типичной – ей непосредственно передается экземпляр класса, который   
создается перед вызовом инструкции raise или внутри нее. Если инструкции   
передается класс, интерпретатор вызовет конструктор класса без аргументов,   
а полученный экземпляр передаст инструкции raise – если после имени класса   
добавить круглые скобки, мы получим эквивалентную форму. Третья форма   
инструкции raise просто повторно возбуждает текущее исключение – это удоб-  
но, когда возникает необходимость передать перехваченное исключение друго-  
му обработчику.  
Чтобы лучше понять вышесказанное, рассмотрим несколько примеров. Сле-  
дующие две формы возбуждения встроенных исключений эквивалентны – они   
обе возбуждают экземпляр по имени класса, но первая из них создает экзем-  
пляр неявно:  
raise IndexError # Класс (экземпляр создается неявно)  
raise IndexError() # Экземпляр (создается в инструкции)  
Мы также можем создать экземпляр заранее – инструкция raise принимает   
ссылки на объекты любых типов, поэтому следующие два примера точно так   
же возбуждают исключение IndexError, как и предыдущие:  
exc = IndexError() # Экземпляр создается заранее  
raise exc

944   
Глава 33. Особенности использования исключений   
excs = [IndexError, TypeError]  
raise excs[0]  
При возбуждении исключения интерпретатор отправляет возбужденный эк-  
земпляр вместе с исключением. Если инструкция try включает предложение   
вида except name as X:, переменной X будет присвоен экземпляр, переданный ин-  
струкции raise:  
try:  
 ...  
except IndexError as X: # Переменной X будет присвоен экземпляр исключения  
 ...  
Ключевое слово as является необязательным в обработчиках инструкции try   
(если оно опущено, интерпретатор просто не будет присваивать экземпляр   
переменной), но с его помощью можно получить доступ к данным экземпляра   
и методам класса исключения.  
Точно так же действуют и исключения, определяемые пользователем в виде   
классов. Ниже приводится пример передачи аргумента конструктору класса   
исключения, значение которого становится доступным в обработчике через эк-  
земпляр, присвоенный переменной:  
class MyExc(Exception): pass  
...  
raise MyExc(‘spam’) # Вызов конструктора класса с аргументом  
...  
try:  
 ...  
except MyExc as X: # Атрибуты экземпляра доступны в обработчике  
 print(X.args)  
Однако это описание пересекается с темой следующей главы, поэтому я пока   
отложу описание дополнительных подробностей.  
Независимо от того, какие исключения будут использованы, они всегда иден-  
тифицируются обычными объектами и только одно исключение может быть   
активным в каждый конкретный момент времени. Как только исключение   
перехватывается предложением except, находящимся в любом месте програм-  
мы, исключение деактивируется (то есть оно не будет передано другой инструк-  
ции try), если не будет повторно возбуждено при помощи инструкции raise или   
в результате ошибки.  
Пример: возбуждение и обработка   
собственных исключений  
Программы на языке Python с помощью инструкции raise могут возбуждать   
как встроенные, так и собственные исключения. В настоящее время собствен-  
ные исключения в программе должны быть представлены объектами экзем-  
пляров классов, как, например, MyBad в следующем примере:  
class MyBad: pass  
   
def stuff():  
 raise MyBad() # Возбудить исключение вручную  
try:  
 stuff() # Возбуждает исключение

Инструкция raise   
945  
except MyBad:  
 print ‘got it’ # Здесь выполняется обработка исключения  
... # С этого места продолжается выполнение программы  
На этот раз исключение происходит внутри функции, но в действительности   
это не имеет никакого значения – управление немедленно передается блоку   
except. Обратите внимание, что инструкция try перехватывает собственные ис-  
ключения программы точно так же, как и встроенные исключения.  
Пример: повторное возбуждение исключений   
с помощью инструкции raise  
Инструкция raise, в которой отсутствует имя исключения или дополнитель-  
ные данные, просто повторно возбуждает текущее исключение. В таком виде   
она обычно используется, когда необходимо перехватить и обработать исклю-  
чение, но при этом не требуется деактивировать исключение:  
>>> try:  
... raise IndexError(‘spam’) # Исключения сохраняют аргументы  
... except IndexError:  
... print(‘propagating’)  
... raise # Повторное возбуждение последнего исключения  
...  
propagating  
Traceback (most recent call last):  
 File “<stdin>”, line 2, in <module>  
IndexError: spam  
При таком использовании инструкция raise повторно возбуждает исключение,   
которое затем передается обработчику более высокого уровня (или обработчи-  
ку по умолчанию, который останавливает выполнение программы и выводит   
стандартное сообщение об ошибке). Обратите внимание, как отображается   
значение аргумента в тексте сообщения об ошибке, который был передан кон-  
структору класса, – почему это происходит, вы узнаете в следующей главе.  
Изменения в Python 3.0: raise from  
В Python 3.0 (но не в 2.6) инструкция raise может также включать дополни-  
тельное предложение from:  
raise exception from otherexception  
При использовании предложения from второе выражение определяет еще один   
класс исключения или экземпляр, который будет присвоен атрибуту \_\_cause\_\_   
возбуждаемого исключения. Если возбужденное исключение не будет перехва-  
чено, интерпретатор выведет информацию об обоих исключениях:  
>>> try:  
... 1 / 0  
... except Exception as E:  
... raise TypeError(‘Bad!’) from E  
...  
Traceback (most recent call last):  
 File “<stdin>”, line 2, in <module>  
ZeroDivisionError: int division or modulo by zero

946   
Глава 33. Особенности использования исключений   
The above exception was the direct cause of the following exception:  
(Исключение выше стало прямой причиной следующего исключения:)  
   
Traceback (most recent call last):  
 File “<stdin>”, line 4, in <module>  
TypeError: Bad!  
Когда исключение возбуждается в обработчике исключения, подобная проце-  
дура выполняется неявно: предыдущее исключение присваивается атрибуту   
\_\_context\_\_ нового исключения и также выводится в стандартный поток оши-  
бок, если исключение не будет перехвачено. Это достаточно сложное и малопо-  
нятное расширение языка, поэтому за дополнительной информацией обращай-  
тесь к руководствам по языку Python.  
Примечание,  касающееся  различий  между  версиями: В версии   
Python 3.0 больше не поддерживается форма инструкции raise   
Exc, Args, которая все еще сохраняется в Python 2.6. Вместо нее   
в версии 3.0 следует использовать описанную выше форму raise   
Exc(Args), которая создает экземпляр исключения. Эквивалент-  
ная форма с запятой, сохранившаяся в версии 2.6, считается   
устаревшей. В Python 2.6 она предоставляется для обратной со-  
вместимости с ныне недействующей моделью исключений, осно-  
ванной на операциях со строками, и не рекомендуется к исполь-  
зованию. При использовании этой формы в версии 2.6 она авто-  
матически преобразуется в форму с вызовом конструктора клас-  
са исключения, используемую в версии 3.0. Как и в предыдущих   
версиях, допускается использовать форму raise Exc – в обеих   
версиях она автоматически будет преобразована в форму raise   
Exc(), вызывающую конструктор класса без аргументов.  
Инструкция assert  
Язык Python включает инструкцию assert в качестве особого случая возбуж-  
дения исключений на этапе отладки. Это сокращенная форма типичного ша-  
блона использования инструкции raise, которая представляет собой условную   
инструкцию raise. Инструкция вида:  
assert <test>, <data> # Часть <data> является необязательной  
представляет собой эквивалент следующего фрагмента:  
if \_\_debug\_\_:  
 if not <test>:  
 raise AssertionError(<data>)  
Другими словами, если условное выражение возвращает ложное значение,   
интерпретатор возбуждает исключение: элемент данных (если присутствует)   
играет роль аргумента конструктора исключения. Как и все исключения, ис-  
ключение AssertionError приводит к завершению программы, если не будет   
перехвачено инструкцией try, и в этом случае элемент данных отображается   
как часть сообщения об ошибке.  
Существует дополнительная возможность удалить все инструкции assert из   
скомпилированного байт-кода программы за счет использования флага ко-

Инструкция assert   
947  
мандной строки -O при запуске интерпретатора и тем самым оптимизировать   
программу. Исключение AssertionError является встроенным исключением,   
а имя \_\_debug\_\_ – встроенным флагом, который автоматически получает значе-  
ние True (истина), когда не используется флаг -O. Используйте команду вида py-  
thon –O main.py, чтобы запустить программу в оптимизированном режиме и от-  
ключить все инструкции assert.  
Пример: проверка соблюдения ограничений   
(но не ошибок)  
Обычно инструкция assert используется для проверки условий выполнения   
программы во время разработки. При отображении в текст сообщений об ошиб-  
ках, полученных в результате выполнения инструкции assert, автоматически   
включается информация из строки исходного программного кода и значения,   
перечисленные в инструкции. Рассмотрим файл asserter.py:  
def f(x):  
 assert x < 0, ‘x must be negative’  
 return x \*\* 2  
   
% python  
>>> import asserter  
>>> asserter.f(1)  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
 File “asserter.py”, line 2, in f  
 assert x < 0, ‘x must be negative’  
AssertionError: x must be negative  
Важно не забывать, что инструкция assert главным образом предназначена   
для проверки соблюдения ограничений, накладываемых программистом, а не   
для перехвата настоящих ошибок. Так как интерпретатор Python в состоянии   
сам выявлять ошибки во время выполнения программы, обычно нет необходи-  
мости использовать assert для выявления таких проблем, как выход индекса   
за допустимые пределы, несоответствие типов или деление на ноль:  
def reciprocal(x):  
 assert x != 0 # Бесполезная инструкция assert!  
 return 1 / x # Интерпретатор автоматически проверит на равенство нулю  
Такие инструкции assert являются лишними, потому что встретив ошибку,   
интерпретатор автоматически возбудит исключение, и вы вполне можете по-  
ложиться в этом на него.1 Еще один пример типичного использования инструк-  
ции assert приводится в примере абстрактного суперкласса в главе 28 – там   
инструкция assert использовалась для того, чтобы вызов неопределенных ме-  
тодов приводил к исключению с определенным текстом сообщения.  
1   
По крайней мере, в большинстве случаев. Как предлагалось ранее в этой книге, про-  
верка на наличие ошибки может использоваться в функции, выполняющей необра-  
тимые действия или производящей длительные вычисления. Но даже в этом случае   
старайтесь не использовать чрезмерно специализированные или чрезмерно ограни-  
чительные проверки, т.к. в противном случае это ограничит область применения ва-  
шего программного кода.

948   
Глава 33. Особенности использования исключений   
Контекстные менеджеры with/as  
В версии Python 2.6 и 3.0 появилась новая инструкция, имеющая отношение   
к исключениям – with, с необязательным предложением as. Эта инструкция   
предназначена для работы с объектами контекстных менеджеров, которые   
поддерживают новый протокол взаимодействия, основанный на использова-  
нии методов. Данная особенность доступна также в Python 2.5 в виде необяза-  
тельного расширения, которое можно активировать инструкцией:  
 from \_\_future\_\_ import with\_statement  
В двух словах, инструкция with/as может использоваться как альтернатива   
известной идиомы try/finally� подобно этой инструкции она предназначена   
для выполнения заключительных операций независимо от того, возникло ли   
исключение на этапе выполнения основного действия. Однако, в отличие от   
инструкции try/finally, инструкция with поддерживает более богатый возмож-  
ностями протокол, позволяющий определять как предварительные, так и за-  
ключительные действия для заданного блока программного кода.  
Язык Python дополняет некоторые встроенные средства контекстными менед-  
Python дополняет некоторые встроенные средства контекстными менед-  
 дополняет некоторые встроенные средства контекстными менед-  
жерами, например файлы, которые закрываются автоматически, или блоки-  
ровки потоков выполнения, которые автоматически запираются и отпирают-  
ся. Однако программист также может создавать с классами и свои контекст-  
ные менеджеры.  
Основы использования  
Основная форма инструкции with выглядит, как показано ниже:  
with выражение [as переменная]:  
 блок with  
Здесь предполагается, что выражение возвращает объект, поддерживающий   
протокол контекстного менеджера (вскоре я расскажу об этом протоколе под-  
робнее). Этот объект может возвращать значение, которое будет присвоено   
переменной, если присутствует необязательное предложение as.  
Обратите внимание, что переменной  необязательно будет присвоен  результат   
выражения – результатом выражения является объект, который поддерживает кон-  
текстный протокол, а переменной может быть присвоено некоторое другое значе-  
ние, предназначенное для использования внутри инструкции. Объект, возвра-  
щаемый выражением, может затем выполнять предварительные действия перед   
тем, как будет запущен блок with, а также завершающие действия после того,   
как этот блок будет выполнен, независимо от того, было ли возбуждено исклю-  
чение при его выполнении.  
Некоторые встроенные объекты языка Python были дополнены поддержкой   
протокола управления контекстом и потому могут использоваться в инструк-  
ции with. Например, объекты файлов снабжены менеджером контекста, кото-  
рый автоматически закрывает файл после выполнения блока with независимо   
от того, было ли возбуждено исключение при его выполнении:  
with open(r’C:\misc\data’) as myfile:  
 for line in myfile:  
 print(line)  
 ...остальной программный код...

Контекстные менеджеры with/as   
949  
Здесь вызываемая функция open возвращает объект файла, который присваива-  
ется имени myfile. Применительно к переменной myfile мы можем использовать   
обычные средства, предназначенные для работы с файлами, – в данном случае   
с помощью итератора выполняется чтение строки за строкой в цикле for.  
Однако данный объект поддерживает протокол управления контекстом, ис-  
пользуемый инструкцией with. После того как инструкция with начнет выпол-  
нение, механизм управления контекстом гарантирует, что объект файла, на   
который ссылается переменная myfile, будет закрыт автоматически, даже если   
в цикле for во время обработки файла произойдет исключение.  
Объекты файлов закрываются автоматически в момент их утилизации сбор-  
щиком мусора, однако нет никакой возможности узнать заранее, когда это   
произойдет. Инструкция with, используемая в таком качестве, представляет   
альтернативное решение, позволяющее гарантировать, что файл будет закрыт   
сразу после выполнения определенного блока программного кода. Как мы уже   
видели выше, аналогичного эффекта можно добиться с помощью более универ-  
сальной и более явной инструкции try/finally, но в данном случае для этого   
потребуется написать четыре строки программного кода вместо одного:  
myfile = open(r’C:\misc\data’)  
try:  
 for line in myfile:  
 print(line)  
 ...остальной программный код...  
finally:  
 myfile.close()  
Мы не будем рассматривать в этой книге многопоточную модель выполнения   
в языке Python (за дополнительной информацией по этой теме вам следует об-  
Python (за дополнительной информацией по этой теме вам следует об-  
 (за дополнительной информацией по этой теме вам следует об-  
ращаться к книгам, посвященным прикладному программированию, таким   
как «Программирование на Python»), но блокировка и средства синхрониза-  
Python»), но блокировка и средства синхрониза-  
»), но блокировка и средства синхрониза-  
ции посредством условных переменных также поддерживаются инструкцией   
with за счет обеспечения поддержки протокола управления контекстом:  
lock = threading.Lock()  
with lock:  
 # Критическая секция программного кода  
 ...доступ к совместно используемым ресурсам...  
Здесь механизм управления контекстом гарантирует, что блокировка автома-  
тически будет приобретена до того, как начнет выполняться блок, и освобож-  
дена по завершении работы блока независимо от того, было ли возбуждено ис-  
ключение при его выполнении.  
Модуль decimal, представленный в главе 5, также использует менеджеры кон-  
текста для упрощения сохранения и восстановления текущего контекста вы-  
числений, определяющего параметры точности и округления, используемые   
в вычислениях:  
with decimal.localcontext() as ctx:  
 ctx.prec = 2  
 x = decimal.Decimal(‘1.00’) / decimal.Decimal(‘3.00’)  
После выполнения этой инструкции менеджер локального контекста текущего   
потока выполнения автоматически восстановит его в прежнее состояние, пред-  
шествовавшее началу выполнения инструкции. Чтобы реализовать то же са-

950   
Глава 33. Особенности использования исключений   
мое с помощью инструкции try/finally, нам потребовалось бы предварительно   
сохранить контекст, а затем восстановить его вручную.  
Протокол управления контекстом  
Некоторые встроенные типы данных уже содержат реализацию менеджеров   
контекста, однако точно так же мы можем сами добавлять менеджеры кон-  
текста в свои собственные классы. Для реализации менеджеров контекста   
используются специальные методы классов, которые относятся к категории   
методов перегрузки операторов и обеспечивают взаимодействие с инструкци-  
ей with. Интерфейс, который должны реализовать объекты для использования   
совместно с инструкцией with, достаточно сложен, хотя большинству програм-  
мистов достаточно лишь знать, как используются существующие контексты   
менеджеров. Однако разработчикам программных инструментов может потре-  
боваться знание правил создания новых менеджеров, поэтому коротко рассмо-  
трим основные принципы.   
Ниже описывается, как в действительности работает инструкция with:  
1. Производится вычисление выражения, возвращающего объект, извест-  
ный как менеджер  контекста, который должен иметь методы \_\_enter\_\_   
и \_\_exit\_\_.  
2. Вызывается метод \_\_enter\_\_ менеджера контекста. Возвращаемое значение   
метода присваивается переменной в предложении as, если оно имеется,   
в противном случае значение просто уничтожается.  
3. Затем выполняется блок программного кода, вложенный в инструкцию   
with.  
4. Если при выполнении блока возбуждается исключение, вызывается ме-  
тод \_\_exit\_\_(тип, значение, диагностическая\_информация), которому передается   
подробная информация об исключении. Обратите внимание, что это те же   
самые значения, которые возвращает функция sys.exec\_info, описываемая   
в руководстве по языку Python и далее в этой книге. Если этот метод возвра-  
щает ложное значение, исключение возбуждается повторно, в противном   
случае исключение деактивируется. Обычно исключение следует возбуж-  
дать повторно, чтобы оно могло выйти за пределы инструкции with.  
5. Если в блоке with исключение не возникает, метод \_\_exit\_\_ все равно вызы-  
вается, но в аргументах тип, значение и диагностическая\_информация ему пере-  
дается значение None.  
Рассмотрим небольшой пример, демонстрирующий работу протокола. Следу-  
ющий фрагмент определяет объект менеджера контекста, который сообщает   
о входе и выходе из блока программного кода любой инструкции with, с кото-  
рой он используется:  
class TraceBlock:  
 def message(self, arg):  
 print(‘running’, arg)  
 def \_\_enter\_\_(self):  
 print(‘starting with block’)  
 return self  
 def \_\_exit\_\_(self, exc\_type, exc\_value, exc\_tb):  
 if exc\_type is None:  
 print(‘exited normally\n’)

Контекстные менеджеры with/as   
951  
 else:  
 print(‘raise an exception!’, exc\_type)  
 return False # повторное возбуждение  
   
with TraceBlock() as action:  
 action.message(‘test 1’)  
 print(‘reached’)  
   
with TraceBlock() as action:  
 action.message(‘test 2’)  
 raise TypeError  
 print(‘not reached’)  
Обратите внимание, что метод \_\_exit\_\_ должен возвращать False, чтобы раз-  
решить дальнейшее распространение исключения – отсутствие инструкции   
return обеспечивает тот же самый эффект, потому что в этом случае по умолча-  
нию возвращается значение None, которое по определению является ложным.   
Кроме того, следует заметить, что метод \_\_enter\_\_ возвращает сам объект self,   
который присваивается переменной в предложении as� при желании этот ме-  
тод может возвращать совершенно другой объект.  
При запуске этого фрагмента менеджер контекста с помощью своих методов \_\_  
enter\_\_ и \_\_exit\_\_ отмечает моменты входа и выхода из блока инструкции with.   
Ниже демонстрируется работа этого сценария под управлением Python 3.0 (он   
также будет работать под управлением Python 2.6, но в выводе появятся допол-  
Python 2.6, но в выводе появятся допол-  
 2.6, но в выводе появятся допол-  
нительные круглые скобки):  
% python withas.py  
starting with block  
running test 1  
reached  
exited normally  
   
starting with block  
running test 2  
raise an exception! <class ‘TypeError’>  
Traceback (most recent call last):  
 File “withas.py”, line 20, in <module>  
 raise TypeError  
TypeError  
Менеджеры контекста являются новейшими механизмами, предназначенны-  
ми для разработчиков инструментальных средств, поэтому мы не будем рас-  
сматривать здесь дополнительные подробности (за полной информацией обра-  
щайтесь к стандартным руководствам по языку� например, новый стандарт-  
ный модуль contextlib содержит дополнительные средства, которые могут ис-  
пользоваться при создании менеджеров контекстов). В более простых случаях   
инструкция try/finally обеспечивает достаточную поддержку для выполнения   
завершающих действий.  
В грядущей версии Python 3.1 в инструкции with можно будет   
также определять сразу несколько (иногда их называют «вло-  
женными») менеджеров контекста через запятую. В следующем   
фрагменте, например при выходе из блока инструкции with ав-  
томатически выполняются заключительные операции для обо-  
их файлов независимо от наличия исключений:

952   
Глава 33. Особенности использования исключений   
with open(‘data’) as fin, open(‘res’, ‘w’) as fout:  
 for line in fin:  
 if ‘some key’ in line:  
 fout.write(line)  
В одной инструкции with может быть перечислено любое коли-  
чество менеджеров контекста, которые будут действовать как   
вложенные инструкции with. Вообще говоря, реализация в вер-  
сии 3.1 (и выше):  
with A() as a, B() as b:  
 ...инструкции...  
эквивалентна следующей реализации, которая будет работать   
в 3.1, 3.0 и 2.6:  
with A() as a:  
 with B() as b:  
 ...инструкции...  
Дополнительные подробности вы найдете в примечаниях к вы-  
пуску Python 3.1.  
В заключение  
В этой главе мы приступили к более глубокому изучению вопросов обработки   
исключений и к исследованию инструкций, связанных с исключениями в язы-  
ке Python: инструкция try используется для перехвата исключений, raise ис-  
пользуется для их возбуждения, assert используется для возбуждения исклю-  
чений по условию и with используется для обертывания программного кода   
менеджерами контекстов, определяющими действия на входе и выходе.  
Пока исключения выглядят достаточно простым инструментом, впрочем, та-  
ковым они и являются – единственная сложность заключается в их идентифи-  
кации. Следующая глава продолжит наши исследования описанием реализа-  
ции наших собственных объектов исключений, где будет показано, что классы   
позволяют создавать исключения более полезные, чем простые строки. Одна-  
ко, прежде чем двинуться вперед, ответьте на контрольные вопросы по темам,   
охваченным в этой главе.  
Закрепление пройденного  
Контрольные вопросы  
1. Для чего служит инструкция try?  
2. Какие две основные разновидности инструкции try существуют?  
3. Для чего служит инструкция raise?  
4. Для чего служит инструкция assert, и какую другую инструкцию она на-  
поминает?  
5. Для чего служит инструкция with/as, и какие другие инструкции она напо-  
минает?

Закрепление пройденного   
953  
Ответы  
1. Инструкция try служит для перехвата исключений и проведения восста-  
новительных действий после них. Она определяет блок выполняемого про-  
граммного кода и один или более обработчиков исключений, которые могут   
возникнуть в ходе выполнения блока.  
2. Существует две основные разновидности инструкции try – это try/except/  
else (используется для перехвата исключений) и try/finally (используется   
для указания завершающих действия, которые должны быть выполнены   
независимо от того, возникло ли исключение или нет). В версии Python   
2.4 это две отдельные инструкции, которые можно объединить вложением   
друг в друга. В версии 2.5 и выше блоки except и finally могут смешиваться   
в одной и той же инструкции, то есть две формы инструкции объединены   
в одну. В объединенной форме блок finally по-прежнему выполняется при   
выходе из инструкции try независимо от того, было обработано исключение   
или нет.  
3. Инструкция raise возбуждает (запускает) исключение. Интерпретатор по-  
средством внутренних механизмов возбуждает встроенные исключения,   
а ваши сценарии с помощью инструкции raise могут возбуждать как встро-  
енные, так и свои собственные исключения.  
4. Инструкция assert возбуждает исключение AssertionError, когда условное   
выражение возвращает ложное значение. Она напоминает инструкцию   
raise, обернутую инструкций if.  
5. Инструкция with/as предназначена для автоматического запуска программ-  
ного кода, выполняющего предварительные и завершающие действия пе-  
ред входом и после выхода из обернутого блока программного кода. Она   
в общих чертах напоминает инструкцию try/finally, так как тоже выполня-  
ет действия на выходе независимо от того, возникло исключение или нет, но   
в отличие от последней, позволяет определять действия на входе и на выхо-  
де, используя для этого протокол, основанный на использовании объектов.

Глава 34.  
   
Объекты исключений  
До сих пор я преднамеренно умалчивал о том, чем в действительности являют-  
ся исключения. Как уже упоминалось в предыдущей главе, в Python 2.6 и 3.0   
встроенные и пользовательские исключения идентифицируются объектами   
экземпляров классов. Хотя это и означает, что вы будете вынуждены исполь-  
зовать приемы объектно-ориентированного программирования, чтобы опре-  
делять новые исключения в своих программах, тем не менее, следует иметь   
в виду, что классы и ООП вообще обладают многими преимуществами.  
Ниже перечислены некоторые преимущества, которыми обладают исключе-  
ния, основанные на классах:   
 •  
Они могут быть организованы в категории. Классы исключений поддержи-  
вают возможность изменения в будущем – добавление новых исключений   
в будущем вообще не будет требовать изменений в инструкциях try.  
 •  
Они могут нести в себе информацию о состоянии. Классы исключений пре-  
доставляют естественное место для хранения информации, доступной для   
обработчиков в инструкции try. Они могут включать как информацию о со-  
стоянии, так и методы, доступные через экземпляры класса.  
 •  
Поддерживают наследование. Исключения на основе классов могут при-  
нимать участие в иерархиях наследования с целью обладания общим по-  
ведением – наследовать методы отображения, например, чтобы обеспечить   
единый стиль сообщений об ошибках.  
Обладая этими преимуществами, исключения на основе классов лучше под-  
держивают возможность развития программ и крупных систем. В действи-  
тельности, по причинам, что указаны выше, все встроенные исключения иден-  
тифицируются классами и организованы в виде дерева наследования. Вы мо-  
жете избрать такой же подход при создании своих собственных исключений.  
В Python 3.0 пользовательские исключения наследуют суперклассы встро-  
Python 3.0 пользовательские исключения наследуют суперклассы встро-  
 3.0 пользовательские исключения наследуют суперклассы встро-  
енных исключений. Эти суперклассы, как мы увидим далее, предоставляют   
удобные средства по умолчанию для вывода и сохранения информации, поэто-  
му для успешного создания собственных исключений вам необходимо пони-  
мать назначение встроенных исключений.

Исключения: назад в будущее   
955  
Примечание,  касающееся  различий  между  версиями: В обеих   
версиях Python, 2.6 и 3.0, исключения должны определяться   
как классы. Кроме того, в версии 3.0 требуется, чтобы классы   
исключений наследовали суперкласс встроенного исключения   
BaseException, прямо или косвенно. Как мы увидим далее, в боль-  
шинстве случаев пользовательские исключения наследуют под-  
класс Exception этого класса для поддержки универсальных об-  
работчиков исключений обычных типов – при указании имени   
этого класса в обработчике большинство программ будут пере-  
хватывать все исключения, которые они должны перехваты-  
вать. Классические классы в Python 2.6 также могут играть   
роль исключений, но если класс исключения наследует класс   
встроенного исключения, он автоматически становится классом   
нового стиля, каковыми являются все классы в версии 3.0.  
Исключения: назад в будущее  
Когда-то давно (до выхода версий Python 2.6 и 3.0) было возможно определять   
исключения двумя разными способами. Это усложняло использование ин-  
струкций try и raise, да и сам язык Python. На сегодняшний день существует   
только один способ определения исключений. Это была отличная идея – уда-  
лить из языка всякий хлам, накопившийся из-за необходимости сохранять   
обратную совместимость. Поскольку знакомство со старым способом поможет   
понять, почему исключения стали такими, какие они есть сейчас, а также по-  
тому что невозможно стереть полностью все то, что использовалось миллиона-  
ми программистов на протяжении почти двух десятилетий, мы начнем наше   
исследование с беглого обзора прошлого.  
Строковые исключения ушли в прошлое!   
До выхода версий Python 2.6 и 3.0 имелась возможность определять исключе-  
ния в виде экземпляров классов и в виде объектов строк. Строковые исклю-  
чения генерировали предупреждения о нежелательности их использования   
в Python 2.5 и были удалены в Python 2.6 и 3.0, поэтому в настоящее время   
допускается использовать только исключения на основе классов, которые рас-  
сматриваются в этой книге. Тем не менее если вам приходится сопровождать   
старые программы, вы все еще можете столкнуться со строковыми исключе-  
ниями. Кроме того, они могут встретиться вам в Интернете и в разнообразных   
руководствах, написанных несколько лет тому назад (что воспринимается как   
вечность с точки зрения Python!).  
Строковые исключения были достаточно просты в использовании – допуска-  
лось использовать любые строки, а сопоставление выполнялось по идентично-  
сти объекта, но не по значению (то есть с помощью оператора is, а не ==).  
C:\misc> C:\Python25\python  
>>> myexc = “My exception string” # Неужели мы были когда-то молодыми?  
>>> try:  
... raise myexc  
... except myexc:  
... print(‘caught’)  
...  
caught

956   
Глава 34. Объекты исключений   
Эта форма исключений была ликвидирована, потому что строковые исключе-  
ния не так удобны в крупных программах, как классы, и сложнее в сопрово-  
ждении. Хотя в настоящее время вы и не можете использовать строковые ис-  
ключения, тем не менее, они обеспечивают естественный переход к представ-  
лению модели исключений на основе классов.  
Исключения на основе классов  
Строки обеспечивают самый простой способ определения исключений. Одна-  
ко, как описывалось ранее, классы предоставляют дополнительные преиму-  
щества, которые заслуживают, чтобы познакомиться с ними. Наиболее важ-  
ное преимущество заключается в том, что классы позволяют организовать   
исключения в категории и они обладают большей гибкостью, чем простые   
строки. Кроме того, классы обеспечивают естественный способ присоединения   
к исключениям дополнительной информации и поддерживают наследование.   
Они обеспечивают лучшее решение и в настоящее время представляют един-  
ственную возможность определения новых исключений.  
Помимо отличий в программном коде главное различие между строковыми ис-  
ключениями и исключениями на базе классов заключается в способе иденти-  
фикации возбужденных исключений в предложениях except инструкции try:   
 •  
Строковые исключения идентифицируются по идентичности  объекта:   
идентификация возбужденного исключения в предложении except выпол-  
няется с помощью оператора is.  
 •  
Исключения на основе классов идентифицируются отношением наследова-  
ния: возбужденное исключение считается соответствующим предложению   
except, если в данном предложении указан класс исключения или любой из   
его суперклассов.  
То есть, когда в инструкции try предложение except содержит суперкласс, оно   
будет перехватывать экземпляры этого суперкласса, а также экземпляры всех   
его подклассов, расположенных ниже в дереве наследования. Благодаря этому   
исключения на основе классов поддерживают возможность создания иерархий   
исключений: суперклассы превращаются в имена категорий, а подклассы со-  
ответствуют различным видам исключений внутри категории. Используя имя   
общего суперкласса, предложение except сможет перехватывать целую катего-  
рию исключений – каждый конкретный подкласс будет соответствовать этому   
предложению.  
Строковые исключения не поддерживают эту концепцию: поскольку иденти-  
фикация строковых исключений производится простой операцией проверки   
идентичности объектов, нет никакого простого способа организовать строко-  
вые исключения в более гибкие категории или группы. В результате обработ-  
чики были жестко связаны с наборами исключений, из-за чего существенно   
усложнялась возможность их модификации.  
В дополнение к этой идее исключения на основе классов обеспечивают лучшую   
поддержку информации  о  состоянии (присоединенной к экземплярам) и по-  
зволяют исключениям принимать участие в иерархиях наследования (с цель ю   
обрести общие черты поведения). Благодаря тому, что они обладают всеми   
преимуществами классов и ООП в целом, они представляют собой более мощ-  
ную альтернативу ныне отсутствующим строковым исключениям при незна-  
чительном увеличении объемов программного кода.

Исключения на основе классов   
957  
Создание классов исключений  
Давайте рассмотрим на примере программного кода, как работают классы ис-  
ключений. В следующем файле classexc.py определяется суперкласс с именем   
General и два подкласса с именами Specific1 и Specific2. Этот пример иллюстри-  
рует понятие категорий исключений, где General – это имя категории, а два под-  
класса – это определенные типы исключений внутри категории. Обработчики,   
которые перехватывают исключение General, так же будут перехватывать и все   
его подклассы, в том числе Specific1 и Specific2:  
class General(Exception): pass  
class Specific1(General): pass  
class Specific2(General): pass  
   
def raiser0():  
 X = General() # Возбуждает экземпляр суперкласса исключения  
 raise X  
   
def raiser1():  
 X = Specific1() # Возбуждает экземпляр подкласса исключения  
 raise X  
   
def raiser2():  
 X = Specific2() # Возбуждает экземпляр другого подкласса исключения  
 raise X  
   
for func in (raiser0, raiser1, raiser2):  
 try:  
 func()  
 except General: # Перехватывает исключения General и любые его подклассы  
 import sys  
 print(‘caught:’, sys.exc\_info()[0])  
   
C:\python30> python classexc.py  
caught: <class ‘\_\_main\_\_.General’>  
caught: <class ‘\_\_main\_\_.Specific1’>  
caught: <class ‘\_\_main\_\_.Specific2’>  
Этот фрагмент достаточно прост для понимания, однако у меня имеется не-  
сколько примечаний к реализации:   
Суперкласс Exception  
К классам, используемым для построения дерева категорий исключений,   
предъявляется не так много требований. Фактически все классы в этом   
примере – пустые. Тела этих классов не содержат ничего, кроме инструк-  
 этих классов не содержат ничего, кроме инструк-  
этих классов не содержат ничего, кроме инструк-  
 классов не содержат ничего, кроме инструк-  
классов не содержат ничего, кроме инструк-  
 не содержат ничего, кроме инструк-  
не содержат ничего, кроме инструк-  
 содержат ничего, кроме инструк-  
содержат ничего, кроме инструк-  
 ничего, кроме инструк-  
ничего, кроме инструк-  
, кроме инструк-  
кроме инструк-  
 инструк-  
инструк-  
ции pass. Однако обратите внимание, что здесь класс верхнего уровня на-  
Однако обратите внимание, что здесь класс верхнего уровня на-  
следует встроенный класс Exception. Это является обязательным требова-  
нием в Python 3.0 – классические классы в Python 2.6 также могут играть   
роль исключений, но если класс исключения наследует класс встроенного   
исключения, он автоматически становится классом нового стиля, каковы-  
ми являются все классы в версии 3.0. Класс Exception предоставляет нема-  
ло полезных особенностей, как мы увидим далее. Мы не использовали их   
в этом примере, тем не менее, идею наследовать его можно считать удачной   
в любой версии Python.

958   
Глава 34. Объекты исключений   
Возбуждение экземпляров  
В этом примере мы создаем экземпляры классов в инструкциях raise. В мо-  
дели исключений, основанной на классах, мы всегда возбуждаем и пере-  
хватываем объекты экземпляров классов. Если в инструкции raise имена   
классов указываются без круглых скобок, интерпретатор будет автомати-  
чески создавать экземпляры, вызывая их конструкторы без аргументов.   
Экземпляры исключений могут создаваться до вызова инструкции raise,   
как сделано в данном примере, или внутри нее.  
Перехватывание категорий  
В этом примере определены функции, которые возбуждают экземпляры   
всех трех классов исключений, а на верхнем уровне модуля определена   
инструкция try, которая вызывает функции и перехватывает исключения   
класса General. Та же инструкция try перехватывает и два более специфи-  
ческих исключения, потому что они являются подклассами класса General.  
Информация об исключении  
В этом примере обработчик исключений использует функцию sys.exc\_info –   
как мы узнаем в следующей главе, эта функция обеспечивает обобщенный   
способ получить последнее возбужденное исключение. В двух словах, пер-  
вый элемент в полученном результате – это класс возбужденного исклю-  
чения, а второй – фактический экземпляр исключения. В подобных пред-  
ложениях except, перехватывающих все исключения, принадлежащие не-  
которой категории, как в данном примере, функция sys.exc\_info является   
единственным способом точно определить, что произошло. В данном случае   
ее можно рассматривать, как эквивалент обращения к атрибуту \_\_class\_\_   
экземпляра. Как мы увидим в следующей главе, функция sys.exc\_info часто   
используется в обработчиках пустых предложений except, которые пере-  
хватывают все возможные исключения.  
Последний пункт требует дополнительных пояснений. Внутри обработчика   
можно быть уверенным, что возбужденный экземпляр является экземпляром   
класса, указанного в предложении except, или одного из его подклассов. Благо-  
даря этому тип исключения можно также получить из атрибута \_\_class\_\_ эк-  
земпляра. Ниже приводится фрагмент, который действует точно так же, как   
и предыдущий пример:  
class General(Exception): pass  
class Specific1(General): pass  
class Specific2(General): pass  
   
def raiser0(): raise General()  
def raiser1(): raise Specific1()  
def raiser2(): raise Specific2()  
   
for func in (raiser0, raiser1, raiser2):  
 try:  
 func()  
 except General as X: # X – возбужденный экземпляр  
 print(‘caught:’, X.\_\_class\_\_) # То же, что и sys.exc\_info()[0]  
Так как тип исключения можно определить с помощью атрибута \_\_class\_\_ воз-  
бужденного экземпляра исключения, как в данном примере, функцию sys.exc\_

В чем преимущества иерархий исключений?   
959  
info удобнее использовать в обработчиках пустых предложений except, где нет   
другого способа получить доступ к экземпляру или к его классу. Кроме того,   
в действующих программах обычно не приходится беспокоиться о конкретном   
типе возбужденного исключения – вызывая методы экземпляра, мы автома-  
тически получаем поведение, присущее возбужденному исключению. Подроб-  
нее об этом и о функции sys.exc\_info будет рассказываться в следующей главе.   
Кроме того, если вы забыли назначение атрибута \_\_class\_\_ в экземплярах, об-  
ращайтесь к главе 28 и ко всей шестой части в целом.  
В чем преимущества иерархий исключений?   
Поскольку в примере предыдущего раздела имеется всего три возможных ис-  
ключения, он действительно не может продемонстрировать все преимущества   
применения классов исключений. На самом деле мы могли бы достичь того же   
эффекта, указав в предложении except список имен исключений в круглых   
скобках. Как это можно сделать, показано в файле stringexc.py:  
try:  
 func()  
except (General, Specific1, Specific2): # Перехватывает все эти исключения  
 ...  
Такой подход мог применяться и при использовании ныне отсутствующих   
строковых исключений. Однако в случае разветвленных или глубоких иерар-  
хий исключений, может оказаться гораздо проще перехватывать категории,   
используя классы, чем перечислять в предложении except все исключения,   
входящие в категорию. Кроме того, иерархии категорий можно расширять, до-  
бавляя новые подклассы, не ломая при этом существующий программный код.  
Предположим, что вы занимаетесь разработкой библиотеки, реализующей   
функции обработки числовой информации, которая используется широким   
кругом людей. Во время работы над библиотекой вы обнаруживаете две ситуа-  
ции, которые могут приводить к таким ошибкам, как деление на ноль и пере-  
полнение. Вы описываете эти ошибки как исключения, которые могут возбуж-  
даться библиотекой:  
# mathlib.py  
   
class Divzero(Exception): pass  
class Oflow(Exception): pass  
   
def func():  
 ...  
 raise Divzero()  
Теперь те, кто будет использовать вашу библиотеку, станут стремиться оберты-  
вать вызовы ваших функций или классов инструкцией try, чтобы перехваты-  
вать два ваших исключения (если они не будут перехватывать их, эти исключе-  
ния будут приводить к аварийному завершению программ):  
# client.py  
   
import mathlib  
   
try:  
 mathlib.func(...)

960   
Глава 34. Объекты исключений   
except (mathlib.Divzero, mathlib.Oflow):  
 ...обработка и восстановление после ошибки...  
Все работает просто замечательно и многие начинают использовать вашу би-  
блиотеку. Однако шесть месяцев спустя вы, просматривая программный код   
(программисты обычно склонны делать это), обнаруживаете еще одну ситуа-  
цию, которая может приводить к другой ошибке – потере значимых разрядов,   
после чего добавляете новое исключение:  
# mathlib.py  
   
class Divzero(Exception): pass  
class Oflow(Exception): pass  
class Uflow(Exception): pass  
К сожалению, выпуском новой версии своей библиотеки вы создаете пробле-  
му для тех, кто ею пользуется. Если они явно указывали имена ваших исклю-  
чений, теперь им придется вернуться к своим программам и внести соответ-  
ствующие изменения везде, где производятся обращения к вашей библиотеке,   
чтобы включить вновь добавленное имя исключения:  
# client.py  
   
import mathlib  
   
try:  
 mathlib.func(...)  
except (mathlib.Divzero, mathlib.Oflow, mathlib.Uflow):  
 ...обработка и восстановление после ошибки...  
Вероятно, это не конец света. Если ваша библиотека предназначена исключи-  
тельно для внутреннего использования, вы могли бы внести все необходимые   
изменения самостоятельно. Вы могли бы также написать сценарий, который   
попытается ликвидировать проблему автоматически (едва ли такой сценарий   
будет насчитывать более дюжины строк и на его создание уйдет совсем немного   
времени). Однако если многим людям придется изменять все инструкции try   
всякий раз, когда вы изменяете свой набор исключений, такая политика обнов-  
ления определенно не будет расцениваться как самая вежливая.  
Ваши пользователи могут попытаться избежать этой ловушки, определяя пу-  
стые предложения except, которые перехватывают все исключения:  
# client.py  
   
try:  
 mathlib.func(...)  
except: # Перехватывать все исключения  
 ...обработка и восстановление после ошибки...  
Но при таком решении могут перехватываться посторонние исключения, даже   
такие, которые вызваны опечатками в именах переменных, ошибками работы   
с памятью, прерываниями работы программы с клавиатуры (Ctrl-C) и исклю-  
чения, генерируемые программой при завершении, а для вас было бы нежела-  
тельно, чтобы перехваченные исключения ошибочно классифицировались как   
ошибки в библиотеке.   
И действительно, в подобных ситуациях пользователи стремятся перехва-  
тывать и обрабатывать только определенные исключения, возбуждаемые би-

В чем преимущества иерархий исключений?   
961  
блиотекой, как описывается в документации к ней – если в ходе работы би-  
блиотечной функции возникает какое-то другое исключение, они чаще всего   
расценивают это как ошибку в самой библиотеке (и обычно сообщают об этом   
разработчику!). Как правило, при обработке исключений чем больше опреде-  
ленности, тем лучше (к этой идее мы еще вернемся в разделе с описанием ти-  
пичных проблем, в следующей главе).1  
Так как же быть? Исключения на основе классов полностью ликвиди руют эту   
проблему. Вместо того чтобы определять библиотечные ис ключения как про-  
стой набор независимых классов, их можно оформить в виде дерева классов   
с одним общим суперклассом, охватывающим целую катего рию исключений:  
# mathlib.py  
   
class NumErr(Exception): pass  
class Divzero(NumErr): pass  
class Oflow(NumErr): pass  
...  
def func():  
 ...  
 raise Divzero()  
При таком подходе пользователям вашей библиотеки достаточно бу дет указать   
общий суперкласс (то есть категорию), чтобы перехваты вать все исключения,   
возбуждаемые библиотекой, причем, как суще ствующие, так и те, что появят-  
ся в будущем:  
# client.py  
   
import mathlib  
...  
try:  
 mathlib.func(...)  
except mathlib.NumErr:  
 ...вывод сообщения и восстановление после ошибки...  
Когда вы опять вернетесь к работе над библиотекой, новые исключения можно   
будет добавлять как новые подклассы от общего суперкласса:  
# mathlib.py  
   
...  
class Uflow(NumErr): pass  
1   
Как было предложено одним моим сообразительным студентом, в модуле библиоте-  
ки можно было бы определить кортеж, содержащий все исключения, которые могут   
быть возбуждены библиотекой. Тогда клиент мог бы импортировать этот кортежи   
и использовать его имя в предложении except, чтобы перехватывать все библиотеч-  
ные исключения (вспомните, что при использовании кортежа в предложении except   
будут перехватываться все перечисленные в нем исключения). Когда позднее в би-  
блиотеку добавится новое исключение, можно просто расширить экспортируемый   
кортеж. Такой прием будет работать, но тогда вам придется постоянно обновлять   
кортеж с именами исключений внутри модуля библиотеки. Кроме того, исключения,   
основанные на классах, несут в себе гораздо больше преимуществ, по сравнению со   
строковыми исключениями, чем простое деление на категории – они поддерживают   
возможность присоединять информацию о состоянии, обладают методами, использу-  
ют механизм наследования и поддерживают возможность адаптации, чего лишены   
отдельные исключения.

962   
Глава 34. Объекты исключений   
В результате программный код пользователей, перехватывающий ис ключения   
вашей библиотеки, останется работоспособным без каких-либо изменений. Вы   
свободно сможете добавлять, удалять и изменять исключения произвольным   
образом – пока клиенты используют имя суперкласса, они могут не беспокоить-  
ся об изменениях в вашем наборе исключений. Другими словами, исключения   
на основе классов лучше отвечают требованиям сопровождения, чем строки.   
Кроме того, ис ключения на основе классов могут поддерживать хранение   
информа ции о состоянии и наследование, что идеально подходит для крупных   
программ. Однако чтобы разобраться в этих концепциях, нам сначала нужно   
понять, как соотносятся классы исключений, опреджеляемые пользователем,   
с классами встроенных исключений, которые они наследуют.  
Классы встроенных исключений  
Примеры в предыдущем разделе возникли не на пустом месте. Все встроенные   
исключения, которые могут возбуждаться интерпретатором, являются объек-  
тами предопределенных классов. Кроме того, они организованы в неглубокую   
иерархию с общими суперклас сами категорий и подклассами определенных   
типов исключений, практически так же, как в примере выше.  
В Python 3.0 все знакомые исключения, с которыми нам уже приходилось   
встре чаться (например, SyntaxError), в действительности являются обычными   
классами, доступными в виде встроенных имен в модуле builtins (в Python 2.6   
этот модуль называется \_\_builtin\_\_) и в виде атрибутов модуля exceptions, вхо-  
дящего в состав стандартной библиотеки. Кроме того, в языке Python встро-  
енные исключения орга низованы в иерархию с целью поддержки различных   
режимов пере хвата исключений. Например:   
BaseException  
Корневой суперкласс исключений. Этот класс не предназначен для непо-  
средственного наследования пользовательскими классами (для этого сле-  
дует использовать класс Exception). Он содержит реализацию по умолчанию   
вывода сообщений и обеспечивает сохранение информации о состоянии.   
Если встроенной функции str передать экземпляр этого класса (например,   
с помощью функции print), класс вернет строку с аргументами, которые   
передавались конструктору при создании экземпляра (или пустую строку,   
если конструктор вызывался без аргументов). Кроме того, если подклассы   
не переопределяют конструктор этого класса, все аргументы, передаваемые   
ему при создании экземпляра, сохраняются в атрибуте args экземпляра   
в виде кортежа.  
Exception  
Корневой суперкласс всех прикладных исключений. Это прямой потомок   
суперкласса BaseException и суперкласс для всех других встроенных исклю-  
чений, кроме классов, связанных с событиями завершения программы (Sys-  
temExit, KeyboardInterrupt и GeneratorExit). Почти все пользовательские клас-  
сы исключений должны наследовать этот класс, а не BaseException. При со-  
блюдении этого соглашения предложения except инструкции try, в которых   
указано исключение Exception, будут перехватывать все исключения, кроме   
событий завершения программы, которые обычно обрабатывать не требует-  
ся. В результате использование имени Exception в инструкции try обеспечи-  
вает более точную избирательность, чем пустое предложение except.

Классы встроенных исключений   
963  
ArithmeticError  
Суперкласс всех арифметических ошибок (и подкласс класса Exception).  
OverflowError  
Подкласс класса ArithmeticError, идентифицирующий конкретную арифме-  
тическую ошибку.  
И так далее. Подробнее познакомиться с этой структурой можно либо в спра-  
вочных руководствах, таких как «Pocket Reference», или в руководстве по   
библиотеке Python. Обратите внимание, что деревья классов исключений   
в Python 3.0 и 2.6 немного отличаются. Следует также заметить, что структуру   
дерева классов можно посмотреть в тексте справки для модуля exceptions толь-  
ко в Python 2.6 (этот модуль был исключен из Python 3.0). Описание функции   
help приводится в главах 4 и 15:  
>>> import exceptions  
>>> help(exceptions)  
...объемный текст справки опущен...  
Категории встроенных исключений  
Дерево встроенных классов позволяет определять, насколько конкрет ными   
или универсальными будут ваши обработчики исключений. На пример, встро-  
енное исключение ArithmeticError – это суперкласс для таких более конкретных   
исключений, как OverflowError и ZeroDivisionError. Указав имя ArithmeticError   
в инструкции try, вы будете перехва тывать все арифметические ошибки, а ука-  
зав имя OverflowError, вы бу дете перехватывать только ошибки определенного   
типа и никакие другие.  
Точно так же можно использовать исключение Exception – суперкласс всех   
прикладных исключений в Python 3.0, чтобы организовать обработку всех ис-  
ключений. Использование этого класса по своему действию напоминает пустое   
предложение except, но позволяет игнорировать исключения, связанные с за-  
вершением программы:  
try:  
 action()  
except Exception:  
 ...обработать все прикладные исключения...  
else:  
 ...обработать ситуацию отсутствия исключений...  
Однако данный прием не настолько универсален в Python 2.6, потому что   
исключения, определяемые пользователем как классические классы, не яв-  
ляются подклассами корневого класса Exception. Этот способ обладает более   
высокой надежностью в Python 3.0, так как в этой версии требуется, чтобы все   
классы исключений наследовали какие-либо встроенные исключения. Однако   
даже в Python 3.0 этому приему свойственны те же потенциальные ловушки,   
характерные для пустого предложения except. Как описывалось в предыдущей   
главе – в этом случае могут перехватываться исключения, которые предпола-  
гается обрабатывать в другом месте, что может скрыть подлинные ошибки про-  
граммирования. Так как эта проблема встречается достаточно часто, мы вер-  
немся к ней, когда будем обсуждать проблемы исключений в следующей главе.

964   
Глава 34. Объекты исключений   
Неважно, будете вы использовать категории в дереве встроенных классов или   
нет, этот подход служит отличным примером� при исполь зовании подобных   
методов к созданию своих собственных исключений вы сможете реализовать   
гибкие наборы исключений, которые легко можно изменять.  
Операция вывода по умолчанию   
и сохранение информации  
Встроенные исключения предоставляют реализацию вывода сообщения, ис-  
пользуемую по умолчанию, и сохранение информации о состоянии, что обеспе-  
чивает значительную долю логики работы исключений, в которых нуждаются   
пользовательские классы. Если, наследуя встроенные классы исключений, вы   
не переопределяете конструктор суперкласса, любые аргументы, передавае-  
мые конструктору, будут сохраняться в атрибуте args экземпляра и автомати-  
чески включаться в текст сообщения при выводе экземпляра (если конструк-  
тор вызывался без аргументов, будут выведены пустой кортеж и стандартная   
строка сообщения).  
Это объясняет, почему аргументы, передаваемые конструкторам классов встро-  
енных исключений, включаются в текст сообщения об ошибке – все аргументы   
присоединяются конструктором к экземпляру и отображаются при попытке   
вывести его:  
>>> raise IndexError # То же, что и IndexError(): нет аргументов  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
IndexError  
   
>>> raise IndexError(‘spam’) # Конструктор присоединит аргумент  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
IndexError: spam  
   
>>> I = IndexError(‘spam’) # Аргументы доступны в виде атрибута  
>>> I.args  
(‘spam’,)  
То же относится и к пользовательским исключениям, потому что они наследу-  
ют от встроенных суперклассов конструктор и методы вывода:  
>>> class E(Exception): pass  
...  
>>> try:  
... raise E(‘spam’)  
... except E as X:  
... print(X, X.args) # Выведет аргументы, сохраненные конструктором  
...  
spam (‘spam’,)  
   
>>> try:  
... raise E(‘spam’, ‘eggs’, ‘ham’)  
... except E as X:  
... print(X, X.args)  
...  
(‘spam’, ‘eggs’, ‘ham’) (‘spam’, ‘eggs’, ‘ham’)

Определение текста исключения   
965  
Обратите внимание, что объекты экземпляров исключений сами по себе не яв-  
ляются строками, но они используют протокол перегрузки операторов, кото-  
рый мы изучали в главе 29, и реализуют метод \_\_str\_\_, обеспечивающий преоб-  
разование экземпляра в строку. Чтобы выполнить конкатенацию экземпляра   
с настоящей строкой, его необходимо вручную преобразовать в строковое пред-  
ставление: str(X) + “string”.  
Автоматическая поддержка вывода и сохранения информации удобна сама по   
себе, тем не менее, чтобы организовать сохранение дополнительной информа-  
ции и вывод специфических сообщений, вы всегда можете переопределить уна-  
следованные методы, такие как \_\_str\_\_ и \_\_init\_\_, в подклассах класса Excep-  
tion, о чем рассказывается в следующем разделе.  
Определение текста исключения  
Как мы видели в предыдущем разделе, по умолчанию исключения на основе   
классов выводят значения всех аргументов, которые были переданы конструк-  
тору класса, если их перехватить и вывести:  
>>> class MyBad(Exception): pass  
...  
>>> try:  
... raise MyBad(‘Sorry--my mistake!’)  
... except MyBad as X:  
... print(X)  
...  
Sorry--my mistake!  
Эта же унаследованная модель отображения используется, когда исключение   
отображается в составе сообщения об ошибке, если оно не будет перехвачено   
программой:  
>>> raise MyBad(‘Sorry--my mistake!’)  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
\_\_main\_\_.MyBad: Sorry--my mistake!  
Во многих случаях этого вполне достаточно. Однако, чтобы улучшить сообще-  
ние, необходимо переопределить в классе ис ключения один из двух методов   
перегрузки операции вывода (\_\_repr\_\_ или \_\_str\_\_), чтобы возвращалась же-  
лаемая строка, которая будет отображаться при выводе исключения. Строка,   
возвращаемая методом, будет отображаться при выводе экземпляра исключе-  
ния вручную или когда исключение будет перехвачено обработчиком по умол-  
чанию:  
>>> class MyBad(Exception):  
... def \_\_str\_\_(self):  
... return ‘Always look on the bright side of life...’  
...  
>>> try:  
... raise MyBad()  
... except MyBad as X:  
... print(X)  
...  
Always look on the bright side of life...

966   
Глава 34. Объекты исключений   
>>> raise MyBad()  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
\_\_main\_\_.MyBad: Always look on the bright side of life...  
Важно отметить, что в подобных ситуациях предпочтительнее переопределять   
метод \_\_str\_\_, потому что встроенные суперклассы уже имеют метод \_\_str\_\_,   
который в большинстве случаев (включая операцию вывода) пользуется пре-  
имуществом перед методом \_\_repr\_\_. Если переопределить метод \_\_repr\_\_, опе-  
рация вывода благополучно вызовет метод \_\_str\_\_ вместо вашей версии метода   
\_\_repr\_\_! Подробнее об этих специальных методах рассказывается в главе 29.  
Независимо от того, что вернет ваш метод, это значение будет включено в текст   
сообщения об ошибке для неперехваченных исключений и использовано при   
попытке вывести экземпляр явно. В данном примере метод возвращает жест-  
ко определенную строку, однако он может выполнять любую обработку текста   
и использовать информацию о состоянии, присоединенную к объекту экзем-  
пляра. В следующем разделе мы рассмотрим возможности, которые можно ис-  
пользовать при работе с этой информацией.  
Передача данных в экземплярах   
и реализация поведения   
Помимо поддержки гибких иерархий классы исключений также яв ляются   
удобным местом для хранения дополнительной информации в виде атрибутов   
экземпляров. Как мы уже видели выше, суперклассы встроенных исключе-  
ний реализуют конструктор по умолчанию, который автоматически сохраня-  
ет аргументы в экземпляре, в виде кортежа, в атрибуте с именем args. И хотя   
конструктора по умолчанию в большинстве случаев вполне достаточно, иногда   
возникает необходимость определить собственный конструктор. Кроме того,   
классы могут определять дополнительные методы для использования в обра-  
ботчиках, предоставляя тем самым предопределенную логику обработки ис-  
ключений.  
Передача дополнительной информации об исключении  
Когда возбуждается исключение, оно способно пересекать границы модулей –   
инструкция raise, запускающая исключение, и инструкция try, перехваты-  
вающая его, могут находиться в разных модулях. В общем случае глобальные   
переменные не подходят для сохранения дополнительной информации, потому   
что программный код в инструкции try может не знать, в каком модуле на-  
ходятся эти переменные. Передача дополнительной информации внутри само-  
го экземпляра исключения обеспечивает более надежный способ получить ее   
в инструкции try.  
При использовании классов это может происходить почти автоматически. Как   
мы уже видели, при возбуждении исключения вместе с ним интерпретатор   
передает экземпляр класса. Обработчики в инструкциях try могут получить   
доступ к экземплярам возбужденных исключений, если в предложении except   
указать имя переменной после ключевого слова as. Этот прием обеспечивает   
естественный способ передачи данных и поведения обработчику.

Передача данных в экземплярах и реализация поведения   
967  
Программа, выполняющая анализ файлов, может, например, со общать об ошиб-  
ке форматирования, возбуждая экземпляр исключе ния, который заполняется   
дополнительной информацией об ошибке:  
>>> class FormatError(Exception):  
... def \_\_init\_\_(self, line, file):  
... self.line = line  
... self.file = file  
...  
>>> def parser():  
... raise FormatError(42, file=’spam.txt’) # Если обнаружена ошибка  
...  
>>> try:  
... parser()  
... except FormatError as X:  
... print(‘Error at’, X.file, X.line)  
...  
Error at spam.txt 42  
В этом примере переменной X в предложении except присваивается ссылка на   
экземпляр, который был сгенерирован во время возбужде ния исключения.1   
Благодаря этой переменной мы получаем доступ к атрибутам, присоединен-  
ным к экземпляру нашей реализацией конструктора. Конечно, мы могли бы   
положиться на реализацию сохранения информации, имеющуюся во встроен-  
ных суперклассах, но такой способ хуже подходит для нашего приложения:  
>>> class FormatError(Exception): pass # Наследует конструктор по умолчанию  
...  
>>> def parser():  
... raise FormatError(42, ‘spam.txt’) # Именованные аргументы недопустимы!  
...  
>>> try:  
... parser()  
... except FormatError as X:  
... print(‘Error at:’, X.args[0], X.args[1]) # Не так удобно для   
... # данного приложения  
Error at: 42 spam.txt  
Предоставление методов исключений  
Помимо возможности передавать дополнительную информацию о состоянии   
адаптированные классы могут использоваться для реализации специфиче-  
ского поведения объектов исключений. То есть класс исключения может опре-  
делять дополнительные методы для использования в обработчиках. Ниже   
приводится пример класса исключения, который реализует дополнительный   
1   
Как уже говорилось выше, доступ к объекту экземпляра клас са исключения обе-  
спечивается также вторым элементом кортежа, возвра щаемого функцией sys.exc\_  
info() – инструментом, который возвращает ин формацию о самом последнем исклю-  
чении. Если в предложении except не указано имя переменной, можно использовать   
эту функцию, когда возни кает необходимость обратиться к исключению за получе-  
нием присоединен ных к нему данных или для вызова его методов. Подробнее о функ-  
ции sys.exc\_info рассказывается в следующей главе.

968   
Глава 34. Объекты исключений   
метод, использующий информацию о состоянии для регистрации ошибки   
в файле:   
class FormatError(Exception):  
 logfile = ‘formaterror.txt’  
 def \_\_init\_\_(self, line, file):  
 self.line = line  
 self.file = file  
 def logerror(self):  
 log = open(self.logfile, ‘a’)  
 print(‘Error at’, self.file, self.line, file=log)  
   
def parser():  
 raise FormatError(40, ‘spam.txt’)  
   
try:  
 parser()  
except FormatError as exc:  
 exc.logerror()  
Если запустить этот сценарий, в ответ на вызов метода внутри обработчика ис-  
ключения он запишет сообщение об ошибке в файл:  
C:\misc> C:\Python30\python parse.py  
C:\misc> type formaterror.txt  
Error at spam.txt 40  
При использовании подобных классов методы (такие, как loggeror) могут   
наследо ваться подклассами, а атрибуты экземпляра (такие, как line и file)   
предоставляют возможность сохранения информации о состоянии, обеспечи-  
вая дополнительный контекст для последующих вызовов ме тодов. Кроме того,   
классы исключений легко могут адаптироваться и расширяться благодаря   
наследованию. Другими словами, так как исключения определяются в виде   
классов, все преимущества ООП, о которых мы узнали в шестой части книги,   
доступны и при работе с исключениями.  
В заключение  
В этой главе мы занялись созданием собственных исключений. Здесь мы узна-  
ли, что в Python 2.6 и 3.0 исключения могут быть реализованы как экземпля-  
ры классов (альтернативная модель исключений на основе строковых объек-  
тов, которая была доступна в предыдущих версиях Python, теперь не должна   
использоваться). Классы исключений поддерживают концепцию создания   
иерархий исключений (что положительно сказывается на удобстве сопрово-  
ждения), позволяют присоединять к исключениям дополнительные данные   
и поведение в виде атрибутов и методов экземпляров, а также обеспечивают   
насле дование атрибутов и методов от суперклассов.  
Мы видели, что перехватывая суперкласс в инструкции try, мы пере хватываем   
этот класс, а также все его подклассы, расположенные ни же в дереве наследо-  
вания� суперклассы начинают играть роль назва ний категорий, а подклассы   
становятся определенными типами ис ключений в этих категориях. Мы также   
видели, что суперклассы встроенных исключений, которые должны наследо-  
ваться нашими классами, обеспечивают реализации вывода по умолчанию

Закрепление пройденного   
969  
и сохранения дополнительной информации, которые мы можем переопреде-  
лять в случае необходимости.  
Следующая глава завершает эту часть книги ис следованием некоторых типич-  
ных случаев использования исключе ний и рассмотрением инструментов, наи-  
более часто используемых программистами на языке Python. Однако прежде   
чем двинуться дальше, ответьте на контрольные вопросы к этой главе.  
Закрепление пройденного  
Контрольные вопросы  
1. Какие два новых ограничения пользовательских исключений появились   
в Python 3.0?  
2. Как определяется соответствие исключений на основе классов и обработ-  
чиков?  
3. Назовите два способа присоединения контекстной информации к объектам   
исключени й.  
4. Назовите два способа, позволяющих определить текст сообщения об ошиб-  
ке в объектах исключений.  
5. Почему в настоящее время вы не должны использовать исключе ния на   
основе строк?  
Ответы  
1. В Python 3.0 исключения могут определяться только в виде классов (то   
есть возбуждаются и перехватываются экземпляры классов). Кроме того,   
классы исключений должны наследовать встроенный класс BaseException   
(на практике в большинстве случаев исключения наследуют его подкласс   
Exception, благодаря которому обеспечивается возможность определять об-  
работчики, перехватывающие все обычные исключения).  
2. Соответствие исключений на основе классов определяется отноше нием к су-  
перклассу: при использовании имени суперкласса в обра ботчике исключе-  
ния будут перехватываться экземпляры этого класса, а также экземпляры   
всех его подклассов, расположенных ниже в дереве наследования. Благода-  
ря этому суперклассы можно интерпретировать как категории исключений,   
а подклассы – как более специфичные типы исключений в этих категориях.  
3. Присоединение дополнительной информации к исключениям на ос нове   
классов производится путем заполнения атрибутов объекта эк земпляра ис-  
ключения, часто внутри конструкторов классов. Для самых простых слу-  
чаев суперклассы встроенных исключений предоставляют конструктор,   
который автоматически сохраняет свои аргументы в экземпляре (в атри-  
буте args). В обра ботчиках исключений указывается переменная, которой   
присваи вается экземпляр исключения, после этого имя переменной может   
использоваться для доступа к присоединенной информации и для вызова   
любых унаследованных методов класса.  
4. Текст сообщения об ошибках в исключениях на основе классов мож но   
определить с помощью метода перегрузки \_\_str\_\_. Для самых простых слу-  
чаев суперклассы встроенных исключений обеспечивают автоматическое

970   
Глава 34. Объекты исключений   
отображение всех аргументов, переданных конструктору класса. Объект   
исключения автоматически преобразуется в отображаемую строку, когда   
выполняются явные операции вывода, такие как print и str, или когда сам   
интерпретатор выводит сообщение об ошибке.   
5. Потому что, как заявил Гвидо, они были ликвидированы в Python 2.6 и 3.0.   
На самом деле, для этого есть весьма серьезные основания: строковые ис-  
ключения не поддерживают де ление на категории, не позволяют присоеди-  
нять информацию о со стоянии или наследовать поведение, как исключения   
на основе классов. С практической точки зрения, строковые исключения   
про ще в использовании на первых порах, пока программы достаточно ма-  
ленькие, но их становится сложно использовать, как только про граммы   
становятся больше.

Глава 35.  
   
Использование исключений  
Данная глава завершает эту часть книги рассмотрением некоторых тем, свя-  
занных с проектированием исключений, и примеров их исполь зования. Далее   
следует раздел с описанием типичных проблем и уп ражнения. Поскольку эта   
глава к тому же является последней главой книги, здесь приводится краткий   
обзор средств разработки, которые помогут вам пройти путь от начинающего   
программиста до разработ чика приложений на языке Python.   
Вложенные обработчики исключений  
До сих пор в наших примерах для перехвата исключений использова лась   
единственная инструкция try, но что произойдет, если одну инст рукцию try   
вложить внутрь другой? И, раз уж на то пошло, что про изойдет, если внутри   
инструкции try вызывается функция, которая выпол няет другую инструкцию   
try? С технической точки зрения инструк ции могут вкладываться друг в друга   
как синтаксически, так и по пути следования потока управления через про-  
граммный код.  
Оба эти варианта проще будет понять, если вы узнаете, что интерпре татор   
складывает инструкции try стопкой  во время выполнения. Ко гда возника-  
ет исключение, интерпретатор возвращается к самой по следней инструкции   
try, содержащей соответствующее предложение except. Поскольку каждая   
инструкция try оставляет метку, интерпре татор может возвращаться к более   
ранним инструкциям try, двигаясь по стопке меток. Такое вложение активных   
обработчиков и есть то, что подразумевается, когда мы говорим о распростра-  
нении исключе ний вверх, к обработчикам «более высокого уровня». Эти обра-  
ботчики являются обычными инструкциями try, в которые поток управления   
ходом выполнения программы вошел раньше.  
Рисунок 35.1 иллюстрирует, что происходит, когда возникает вложе ние ин-  
струкций try/except во время выполнения. Объем программного кода, кото-  
рый выполняется в инструкции try, может оказаться весьма существенным   
(например, он может содержать вызовы функций) и не редко вызывает другой   
программный код, который готов перехватить те же самые исключения. Когда   
исключение наконец возбуждается, интерпретатор переходит к самой послед-

972   
Глава 35. Использование исключений   
ней инструкции try, в кото рой указано имя исключения, запускает блок except   
и продолжает вы полнение программы ниже этой инструкции try.  
Как только такое исключение будет перехвачено, его жизнь заканчива ется –   
управление не передается всем соответствующим инструкциям try, содержа-  
щим имя исключения, – только первая из них получает возможность обрабо-  
тать исключение. Например, на рис. 35.1 инструк ция raise в функции func2   
возвращает управление обработчику в функ ции func1, после чего программа   
продолжает выполнение внутри func1.  
Напротив, когда исключение возникает во вложенных инструкциях try/final-  
ly, выполняется каждый блок finally по очереди – интерпре татор продолжает   
передавать исключение вверх по цепочке вложен ных инструкций try, пока не   
будет достигнут обработчик по умолча нию верхнего уровня (который выводит   
стандартные сообщения об ошибках). Как показано на рис. 35.2, предложения   
finally не останав ливают распространение исключений – они лишь определя-  
ют про граммный код, который должен выполняться на выходе из инструк ции   
try:  
 func1()  
except E:  
 ...  
def func1():  
 try:  
 func2()  
 except E:  
 ...  
def func2():  
 ...  
 raise E  
 ...  
Рис. 35.1. Вложенные инструкции try/except: когда возбуждается исключение   
(программой или интерпретатором), происходит возврат к самой последней   
инструкции try с соответствующим предложением except и программа про-  
должает выполнение после этой инструкции try. Предложения except пере-  
хватывают и останавливают дальнейшее распространение исключений –   
это место, где выполняются восстановительные операции после исключения  
try:  
 func1()  
finally:  
 ...  
def func1():  
 try:  
 func2()  
 finally:  
 ...  
def func2():  
 ...  
 raise E  
 ...  
Рис. 35.2. Вложенные инструкции try/finally: когда возбуждается исключение,   
управление возвращается самой последней инструкции try и выполняется   
ее блок finally, после этого исключение продолжит свое движение по блокам   
finally во всех активных инструкциях try, пока в конечном счете не будет   
достигнут обработчик по умолчанию, где производится вывод сообщения об   
ошибке. Предложения finally перехватывают (но не останавливают) исключе-  
ния – они определяют действия, которые должны выполняться «на выходе»

Вложенные обработчики исключений   
973  
try в процессе движения исключения. Если к моменту возникно вения исклю-  
чения имелось несколько активных инструкций try/fi nally, они все будут вы-  
полнены, если только где-то на пути исключения не встретится инструкция   
try/except, которая перехватит его.  
Другими словами, куда будет выполнен переход при возникновении исключе-  
ния, полностью зависит от того, где оно возникло, – это опре деляется ходом вы-  
полнения программы, а не только синтаксисом. Распространение исключения,   
по сути, происходит в порядке, обрат ном порядку вхождения в инструкции try.   
Это движение останавливается, когда управление переходит к соответствую-  
щему блоку except, и про должается когда управление проходит через предло-  
жения finally.  
Пример: вложение в потоке управления  
Обратимся к примеру, чтобы рассмотреть этот тип вложения более кон кретно.   
В следующем файле модуля nestexc.py определяются две функ ции. Функция   
action2 возбуждает исключение (нельзя складывать чис ла и последовательно-  
сти), функция action1 обертывает вызов функции action2 в инструкцию try, ко-  
торая перехватывает исключение:  
def action2():  
 print(1 + []) # Возбуждает исключение TypeError  
   
def action1():  
 try:  
 action2()  
 except TypeError: # Самая последняя соответствующая инструкция try  
 print(‘inner try’)  
   
try:  
 action1()  
except TypeError: # Этот обработчик будет выполнен, только если  
 print(‘outer try’) # action1 повторно возбудит исключение  
   
% python nestexc.py  
inner try  
Обратите внимание, что на верхнем уровне модуля, внизу файла, вы зов функ-  
ции action1 также обернут инструкцией try. В тот момент, когда функция ac-  
tion2 возбуждает исключение TypeError, существуют две активные ин струкции   
try – одна в функции action1 и одна в программном коде на верхнем уровне мо-  
дуля. Интерпретатор выбирает и запускает самую последнюю инструкцию try   
с соответствующим предложением except, которой в данном случае является   
инструкция try в функции action1.  
Как уже говорилось, место, куда будет выполнен переход в случае ис ключения,   
зависит от того, в каком месте программы находится поток управления. Поэто-  
му, чтобы знать, куда будет выполнен переход, не обходимо знать место, где на-  
ходится поток управления. В данном случае вы бор места, где будет обработано   
исключение, больше зависит от того, где находится поток управления, чем от   
синтаксиса. Однако мы мо жем организовать синтаксическое вложение обра-  
ботчиков – эквива лентный случай рассматривается в следующем разделе.

974   
Глава 35. Использование исключений   
Пример: синтаксическое вложение  
В главе 33, когда рассматривалась новая объединенная инструкция try/except/  
finally, я уже говорил, что вполне возможно вкладывать ин струкции try син-  
таксически, задавая вложение в программном коде:  
try:  
 try:  
 action2()  
 except TypeError: # Самая последняя соответствующая инструкция try  
 print(‘inner try’)  
except TypeError: # Этот обработчик будет выполнен, только если  
 print(‘outer try’) # вложенный обработчик повторно возбудит исключение  
Этот программный код задает ту же структуру вложенных обработчи ков, что   
и предыдущий пример (и ведущую себя точно так же). Факти чески инструк-  
ции, вложенные синтаксически, работают точно так же, как показано на   
рис. 35.1 и 35.2� единственное отличие заключа ется в том, что вложенные обра-  
ботчики физически объединены в бло ке инструкции try, а не находятся в раз-  
ных функциях. Например, ис ключение пройдет через все блоки finally незави-  
симо от того, вложе ны они синтаксически или в ходе выполнения программы   
происходит вложение физически отдельных фрагментов программного кода:  
>>> try:  
... try:  
... raise IndexError  
... finally:  
... print(‘spam’)  
... finally:  
... print(‘SPAM’)  
...  
spam  
SPAM  
Traceback (most recent call last):  
 File “<stdin>”, line 3, in <module>  
IndexError  
Графическая иллюстрация порядка выполнения этого фрагмента по казана на   
рис. 35.2 – результат получается тот же самый, но сама ло гика выполнения   
в данном случае образована вложенными инструк циями. Более интересный   
пример синтаксического вложения в дейст вии приводится в следующем файле   
except-finally.py:  
def raise1(): raise IndexError  
def noraise(): return  
def raise2(): raise SyntaxError  
   
for func in (raise1, noraise, raise2):  
 print(‘\n’, func, sep=’ ‘)  
 try:  
 try:  
 func()  
 except IndexError:  
 print(‘caught IndexError’)  
 finally:  
 print(‘finally run’)

Идиомы исключений   
975  
Этот фрагмент перехватывает исключение, если оно будет возбужде но, и вы-  
полняет завершающие действия в блоке finally независимо от того, возникло   
исключение или нет. Чтобы понять это, может потребо ваться некоторое время   
на изучение фрагмента, но результат очень на поминает объединение предло-  
жений except и finally в единственной инструкции try в версии Python 2.5 или   
выше:  
% python except-finally.py  
<function raise1 at 0x026ECA98>  
caught IndexError  
finally run  
   
<function noraise at 0x026ECA50>  
finally run  
   
<function raise2 at 0x026ECBB8>  
finally run  
   
Traceback (most recent call last):  
 File “except-finally.py”, line 9, in <module>  
 func()  
 File “except-finally.py”, line 3, in raise2  
 def raise2(): raise SyntaxError  
SyntaxError: None  
Как мы видели в главе 33, начиная с версии Python 2.5, появилась воз можность   
использовать предложения except и finally в одной инструк ции try. Это делает   
описанный здесь прием синтаксического вложения ненужным, однако он по-  
прежнему работает, его можно встретить в программном коде, написанном до   
выхода версии Python 2.5, и он может использоваться для реализации альтер-  
нативных конструкций обработки исключений.  
Идиомы исключений  
Мы рассмотрели внутренний механизм исключений. Теперь рассмот рим неко-  
торые другие типичные способы их использования.  
Исключения не всегда являются ошибками  
В языке Python все ошибки являются исключениями, но не все исклю чения   
являются ошибками. Например, в главе 9 мы видели, что по дос тижении кон-  
ца файла метод чтения объекта файла возвращает пустую строку. Напротив,   
встроенная функция input (с которой мы впер вые встретились в главе 3 и кото-  
рую использовали в интерактивном цикле в главе 10) читает по одной строке   
текста при каждом вызове из стандартного потока ввода sys.stdin и возбуждает   
исключение EOFError по достижении конца файла (в Python 2.6 эта функция на-  
зывается raw\_input).   
В отличие от методов объекта файла, дан ная функция не возвращает пустую   
строку� пустая строка, полученная от функции input, означает всего лишь пу-  
стую строку. Несмотря на свое название, исключение EOFError в данном кон-  
тексте – это всего лишь сигнал, а не ошибка. По этой причине чтобы избежать   
прежде временного завершения работы сценария, функцию input обертыва ют   
инструкцией try, которую вкладывают в цикл, как показано ниже:

976   
Глава 35. Использование исключений   
while 1:  
 try:  
 line = input() # Прочитать строку из потока stdin  
 except EOFError:  
 break # Выход по достижении конца файла  
 else:  
 ...обработка следующей строки...  
Существуют и другие встроенные исключения, которые являются сиг налами,   
а не ошибками, – вызов функции sys.exit() и нажатие комбинации клавиш   
Ctrl-C, например, возбуждают исключение SystemExit и KeyboardInterrupt соот-  
ветственно. В языке Python имеется также ряд встроен ных исключений, ко-  
торые являются скорее предупреждениями, чем ошибками. Некоторые из них   
применяются, чтобы сообщить о неже лательности использования некоторых   
особенностей языка (которые вскоре будут удалены). За дополнительной ин-  
формацией по предупре ждениям обращайтесь к описанию встроенных исклю-  
чений в руково дстве по стандартной библиотеке и к модулю warnings.  
Передача сигналов из функций по условию  
Исключения, определяемые программой, также могут служить сигна лами об   
условиях, которые не являются ошибками. Например, проце дура поиска мо-  
жет предусматривать возбуждение исключения в слу чае нахождения соответ-  
ствия вместо того, чтобы возвращать флаг со стояния, который должен интер-  
претироваться вызывающей програм мой. В следующем примере инструкция   
try/except/else играет роль инструкции if/else, предназначенной для проверки   
возвращаемого значения:  
class Found(Exception): pass  
   
def searcher():  
 if ...успех...:  
 raise Found()  
 else:  
 return  
   
try:  
 searcher()  
except Found: # Исключение, если элемент найден  
 ...успех...  
else: # иначе: элемент не найден  
 ...неудача...  
В более широком смысле такая организация программного кода мо жет с успе-  
хом использоваться для любой функции, которая не может вернуть специ-  
альный признак, свидетельствующий об успехе или не удаче. Например, если   
любое возвращаемое значение является допус тимым, невозможно выбрать   
какое-то одно значение, которое сигна лизировало бы о необычных состояни-  
ях. Исключения обеспечивают способ подать сигнал, не возвращая значение:  
class Failure(Exception): pass  
   
def searcher():  
 if ...успех...:  
 return ...найденный\_элемент...  
 else:

Идиомы исключений   
977  
 raise Failure()  
   
try:  
 item = searcher()  
except Failure:  
 ...сообщение о неудаче...  
else:  
 ...обработка найденного элемента...  
Поскольку язык Python является динамически типизированным и в сво ей   
основе поддерживает полиморфизм, исключения, а не возвращение специаль-  
ного признака являются более предпочтительным способом сообщать о таких   
состояниях.  
Закрытие файлов и соединений с сервером  
Мы уже сталкивались с похожими примерами в главе 33. Тем не менее повторю   
еще раз, что инструменты обработки исключений также часто используются   
с целью обеспечить освобождение системных ресурсов независимо от того, воз-  
никло исключение в процессе работы или нет.  
Например, некоторые серверы требуют, чтобы по завершении сеанса работы   
соединение было закрыто. Аналогично, после операции вывода в файл может   
потребоваться закрыть его, чтобы вытолкнуть содержимое буферов на диск,   
а неиспользуемые файлы, открытые для чтения, могут понапрасну занимать   
файловые дескрипторы – объекты файлов автоматически закрываются сбор-  
щиком мусора, но иногда бывает очень сложно знать, когда это произойдет на   
самом деле.  
Наиболее простой и очевидный способ гарантировать выполнение заключи-  
тельных операций для какого-то конкретного блока программного кода за-  
ключается в использовании инструкции try/finally:  
myfile = open(r’C:\misc\script’, ‘w’)  
try:  
 ...обработать myfile...  
finally:  
 myfile.close()  
Как мы видели в главе 33, некоторые объекты в Python 2.6 и 3.0 еще больше   
упрощают такую возможность, предоставляя менеджеры контекстов, которые   
могут использоваться совместно с инструкцией with/as, позволяющие автома-  
тически выполнять заключительные операции:  
with open(r’C:\misc\script’, ‘w’) as myfile:  
 ...обработать myfile...  
Так какой же вариант лучше? Как обычно, это зависит от вашей программы.   
В сравнении с инструкцией try/finally менеджеры контекста менее очевидны,   
что противоречит общей философии языка Python. Кроме того, менеджеры   
контекста менее универсальны – они доступны лишь для некоторых типов   
объектов, к тому же создание собственных менеджеров контекста, реализую-  
щих заключительные операции, вообще является более трудоемкой задачей,   
чем использование инструкции try/finally.   
С другой стороны, использование существующих менеджеров контекста тре-  
бует меньше программного кода, чем применение инструкции try/finally, как

978   
Глава 35. Использование исключений   
видно из предыдущих примеров. Кроме того, протокол менеджеров контекста   
кроме заключительных операций предусматривает также возможность реали-  
зации начальных операций, выполняемых на входе. Инструкция try/finally,   
вероятно, используется более широко, однако менеджеры контекста могут ока-  
заться предпочтительнее там, где они доступны или где сложность их создания   
оправдывается удобством использования.  
Отладка с помощью внешних инструкций try  
Обработчики исключений можно также использовать как замену об работчика   
по умолчанию. Обернув всю программу (или вызов ее) во внешнюю инструк-  
цию try, можно перехватывать любые исключения, которые только будут воз-  
никать во время работы программы, отменяя тем самым способ завершения   
программы, заданный по умолчанию.  
В следующем фрагменте пустое предложение except перехватывает любые не-  
обработанные исключения, возникшие в ходе выполнения программы. Чтобы   
получить доступ непосредственно к самому исклю чению, вызовите встроен-  
ную функцию sys.exc\_info из модуля sys – она возвращает кортеж, в котором   
первые два элемента содержат имя исключения и экземпляр класса возбуж-  
денного исключения (вскоре мы подробнее рассмотрим функцию sys.exc\_info):  
try:  
 ...запуск программы...  
except: # Сюда попадут все необработанные исключения  
 import sys  
 print(‘uncaught!’, sys.exc\_info()[0], sys.exc\_info()[1])  
Этот прием часто используется во время разработки, так как он позво ляет   
сохранить программу активной даже после ошибки – с его помощью можно   
производить дополнительные проверки без необходимости перезапус кать про-  
грамму. Это прием может также использоваться для тестиро вания другого   
программного кода, как описано в следующем разделе.  
Запуск тестов в рамках единого процесса  
Некоторые из приемов, которые мы только что рассмотрели, можно было бы   
объединить в тестовом приложении, которое позволяет тести ровать другой   
программный код в рамках одного и того же процесса:  
import sys  
log = open(‘testlog’, ‘a’)  
from testapi import moreTests, runNextTest, testName  
def testdriver():  
 while moreTests():  
 try:  
 runNextTest()  
 except:  
 print(‘FAILED’, testName(), sys.exc\_info()[:2], file=log)  
 else:  
 print(‘PASSED’, testName(), file=log)  
   
testdriver()  
Здесь функция testdriver выполняет в цикле серию тестов (модуль te stapi – не-  
кая абстракция в этом примере). Поскольку в обычной ситуа ции необработан-

Идиомы исключений   
979  
ное исключение приводило бы к завершению самого тестового приложения,   
можно обернуть вызовы очередного теста инст рукцией try, чтобы обеспечить   
продолжение процесса тестирования по сле неудачного завершения любого из   
тестов. Здесь, как обычно, пустое предложение except перехватывает любые не-  
обработанные исключения, возникшие в ходе выполнения теста, и регистриру-  
ет в файле информа цию об исключении, полученную с помощью функции sys.  
exc\_info. Предложение else выполняется в случае отсутствия исключений –   
когда тест завершился благополучно.  
Такой подход типичен для систем, которые тестируют функции, моду ли   
и классы, запуская их в рамках того же самого процесса, что и само тестовое   
приложение. Однако на практике тестирование может ока заться процедурой   
гораздо более сложной, чем показано здесь. Напри мер, чтобы протестировать   
внешнюю программу, может потребоваться проверять коды состояния или   
вывод, создаваемый такими средствами запуска программ, как os.system и os.  
popen, описания которых вы най дете в стандартном руководстве по библиотеке   
(такие инструменты во обще не возбуждают исключений в случае появления   
ошибок во внеш ней программе – фактически тест выполняется параллельно   
с програм мой, выполняющей тестирование).  
В конце этой главы мы познакомимся с некоторыми законченными платфор-  
мами тестирования, предоставляемыми интерпретатором Python, такими как   
doctest и PyUnit, которые обеспечивают возможность сравнения ожи даемого   
вывода с фактическими результатами.  
Подробнее о функции sys.exc\_info  
Функция sys.exc\_info, результаты которой использовались в послед них двух   
разделах, позволяет обработчикам исключений получить доступ к по следнему   
возбужденному исключению. Ее особенно удобно использовать в пустых пред-  
ложениях except, которые перехватывают все исключения, чтобы определить,   
что именно произошло:  
try:  
 ...  
except:  
 # sys.exc\_info()[0:2] – класс исключения и экземпляр  
Если в момент ее вызова ника кое исключение не обрабатывается, функция воз-  
вращает кортеж с тре мя объектами None. В противном случае возвращаются   
(тип, значение, трассировочная\_информация), где:  
 •  
Тип – это класс обрабатываемого исключения.  
 •  
Значение – это экземпляр класса возбужденного исключения.  
 •  
Трассировочная информация – это объект, который представляет стек вы-  
зовов в точке, где возникло исключение (в документации к модулю traceback   
описываются инструменты, которые могут ис пользоваться вместе с этим   
объектом для создания сообщений об ошибках вручную).  
Как мы уже видели в предыдущей главе, иногда функция sys.exc\_info может   
также использоваться, чтобы определить конкретный тип исключения, когда   
выполняется перехват по имени суперкласса категории исключений. Однако,   
как мы видели, в подобном случае тип исключения можно также определить

980   
Глава 35. Использование исключений   
с помощью атрибута \_\_class\_\_ экземпляра, который можно получить с помо-  
щью ключевого слова as, поэтому функция sys.exc\_info чаще всего использует-  
ся в пустых предложениях except:  
try:  
 ...  
except General as instance:  
 # instance.\_\_class\_\_ – класс исключения  
Кроме того, используя интерфейсы объектов экземпляров и опираясь на по-  
лиморфизм, часто бывает лучше просто использовать методы класса исключе-  
ния, чем проверять его тип:  
try:  
 ...  
except General as instance:  
 # instance.method() выполнит действия, ожидаемые от этого экземпляра  
Как обычно, проверять типы объектов в языке Python означает ограничивать   
гибкость программного кода. Реализации, основанные на использовании по-  
лиморфизма, как в последнем примере, обычно обеспечивают лучшую под-  
держку возможных изменений в будущем.  
Примечание,  касающееся  различий  между  версиями: В версии   
Python 2.6 для извлечения типа и значения самого последнего   
исключения можно использовать более старые инструменты, та-  
кие как sys.exc\_type и sys.exc\_value, но они могут использо ваться   
только применительно к единственному исключению, глобаль-  
ному для всего процесса. Эти две функции были удалены   
в Python 3.0. Более новая и более предпочтительная функция   
sys.exc\_info(), доступная в обеих версиях Python, 2.6 и 3.0, запо-  
минает информацию об исключениях в каж дом потоке выполне-  
ния. Конечно, это имеет значение только при ис пользовании не-  
скольких потоков выполнения в программах на языке Python   
(тема, которая выходит далеко за рамки этой книги), однако   
в Python 3.0 она является единственным доступным инструмен-  
том. За допол нительной информацией обращайтесь к справоч-  
ному руководству по библиотеке языка Python и к другим спе-  
циализированным книгам.  
Советы по применению   
и типичные проблемы исключений  
В этой главе я решил объединить советы по применению и описание типичных   
проблем в один раздел, потому что проблемы с исключениями чаще всего тесно   
связаны с особенностями их применения. Вообще говоря, исключения в язы-  
ке Python очень просты в обраще нии. Настоящее искусство их использования   
заключается в принятии решения, насколько универсальными должны быть   
предложения ex cept и какой объем программного кода должен быть обернут   
инструк циями try. Рассмотрим сначала вторую проблему.

Советы по применению и типичные проблемы исключений   
981  
Что должно быть обернуто  
В принципе, можно было бы обернуть каждую инструкцию в сценарии в свою   
собственную инструкцию try, но это будет выглядеть достаточ но глупо (тогда   
инструкции try тоже следовало бы обернуть в инструк ции try!). Это настоящая   
проблема проектирования, которая никак не связана с конкретным языком   
программирования и становится более очевидной на практике. Однако ниже   
приводится несколько правил, выработанных на практике:  
 •  
В инструкции try следует заворачивать операции, для которых неудача не   
является чем-то необычным. Например, операции, взаимодействующие   
с систе мой (открытие файлов, взаимодействия с сокетами и т. д.), являют ся   
первыми кандидатами для заключения их в инструкции try.  
 •  
При этом из первого правила есть исключение – в простых сценари ях быва-  
ет желательно, чтобы подобные неудачи приводили к завер шению работы   
программы. Это особенно верно, когда неудачи ожи даемы. Неудачи в языке   
Python приводят к выводу полезных сооб щений (только не в случае краха   
программы), и они часто представ ляют собой лучший результат, на кото-  
рый только можно надеяться.  
 •  
Завершающие операции должны заключаться в инструкции try/finally,   
чтобы гарантировать их выполнение. Эта форма инструк ции позволяет вы-  
полнять программный код независимо от того, возникло исключение или   
нет.  
 •  
Иногда более удобно завернуть вызов крупной функции в единст венную ин-  
струкцию try, чем засорять эту функцию несколькими инструкциями try.   
При таком подходе все исключения, возникшие в функции, будут перехва-  
чены инструкцией try, окружающей вы зов, за счет чего можно уменьшить   
объем программного кода внут ри самой функции.  
Влияние на количество обработчиков исключений нередко оказывает тип про-  
граммы. Например, серверные программы должны работать постоянно, и поэ-  
тому в них инструкции try наверняка будут необходи мы, чтобы перехватывать   
исключения и выполнять восстановитель ные операции после них. В програм-  
мах тестирования, как мы видели в этой главе, также необходимо выполнять   
обработку исключе ний. Однако в более простых сценариях часто можно вооб-  
ще игнори ровать исключения, потому что неудача на любом этапе выполнения   
требует прекращения работы сценария.  
Не перехватывайте слишком много:   
избегайте пустых предложений except  
К вопросу о степени универсальности обработчика. Язык Python по зволяет   
явно указывать, какие исключения должны перехватываться, и иногда бывает   
необходимо проявлять осторожность, чтобы не пере хватывать слишком много.   
Например, вы уже знаете, что пустое пред ложение except перехватывает все   
исключения, которые только могут возникнуть в блоке try.  
Сделать это несложно и иногда даже желательно, но это может привес ти   
к тому, что будет перехвачена ошибка, обработка которой преду смотрена в ин-  
струкции try на более высоком уровне вложенной струк туры. В примере ниже   
обработчик исключения перехватыва ет и деактивирует все исключения, кото-

982   
Глава 35. Использование исключений   
рые достигнут его, независимо от того, ожидает ли какие-либо исключения об-  
работчик уровнем выше:  
def func():  
 try:  
 ... # Здесь возбуждается исключение IndexError   
 except:  
 ... # Но все исключения попадают сюда!  
   
try:  
 func()  
except IndexError: # Исключение должно обрабатываться здесь  
 ...  
Что еще хуже, такой программный код может перехватывать исклю чения, ко-  
торые вообще не имеют никакого отношения к программе. Даже такие ситуа-  
ции, как ошибки работы с памятью, настоящие ошибки в программном коде,   
прекращение итераций, прерывание с клавиатуры и выход из програм мы, воз-  
буждают исключения. Обычно такие исключения не должны перехватывать-  
ся.  
Например, сценарии обычно завершают работу, когда поток управле ния дости-  
гает конца главного файла. Однако в языке Python имеется специальная функ-  
ция sys.exit(statuscode), с помощью которой можно завершить работу програм-  
мы. Чтобы завершить программу, эта функ ция в действительности возбуждает   
исключение SystemExit, благодаря чему имеется возможность реализовать вы-  
полнение завершающих операций в инструкции try/finally, а в специализиро-  
ванных программах – перехватить это событие.1 По этой причине инструкция   
try с пустым предложением except может непреднамерен но перехватить такое   
важное исключение, как показано в следующем файле (exiter.py):  
import sys  
   
def bye():  
 sys.exit(40) # Серьезная ошибка: завершить работу немедленно!  
   
try:  
 bye()  
except:  
 print(‘got it’) # Ой! Мы проигнорировали команду на завершение  
print(‘continuing...’)  
   
% python exiter.py  
got it  
continuing...  
Вы просто не сможете предугадать все исключения, которые могут произойти   
во время выполнения операции. Решить проблему в данном конкретном слу-  
чае можно с помощью использования встроенных классов исключений, пред-  
1   
Похожая функция os.\_exit также завершает работу программы, но делает это непо-  
средственно – она пропускает этап выполнения завершающих дей ствий и не может   
быть перехвачена с помощью инструкций try/except или try/finally. Обычно эта   
функция используется в дочерних процессах, опи сание которых выходит далеко за   
рамки этой книги. За дополнительной информацией обращайтесь к справочному ру-  
ководству по библиотеке язы ка Python и к другим специализированным книгам.

Советы по применению и типичные проблемы исключений   
983  
ставленных в предыдущей главе, благодаря тому, что суперкласс Exception не   
наследуется классом SystemExit:  
try:  
 bye()  
except Exception: # Не будет препятствовать завершению программы,   
 ... # но БУДЕТ перехватывать массу других исключений  
Однако в других случаях такой подход ничуть не лучше использования пусто-  
го предложения except – так как Exception является суперклассом всех встро-  
енных исключений, кроме исключений завершения программы, при его ис-  
пользовании будут перехватываться все исключения, обрабатывать которые,   
возможно, предполагается где-то в другом месте в программе.  
Вероятно, хуже всего то, что пустое предложение except может пере хватить на-  
стоящие ошибки в программном коде, которым желательно было бы позволить   
пройти дальше. Фактически пустые предложения except могут отключать ме-  
ханизм интерпретатора, предназначенный для вывода сообщений об ошибках,   
скрывая возможные ошибки в программном коде. Например, рассмотрим та-  
кой фрагмент:  
mydictionary = {...}  
...  
try:  
 x = myditctionary[‘spam’] # Ой: опечатка  
except:  
 x = None # А мы предполагаем, что получили KeyError  
...продолжение работы с x...  
Здесь программист предполагает, что в данной ситуации возможен единствен-  
ный тип ошибки – это ошибка отсутствующего ключа. Но поскольку в имени   
словаря myditctionary была допущена опечатка (должно быть mydictionary), ин-  
терпретатор возбуждает исключение NameError, встретив ссылку на неопреде-  
ленное имя, которое благопо лучно будет перехвачено и проигнорировано об-  
работчиком. Обработ чик неправильно запишет в переменную значение по   
умолчанию, за маскировав ошибку в программе. Кроме того, использование   
имени Exception в предложении except даст тот же эффект, что и использование   
пустого предложения except. Если этот программный код будет находиться до-  
статочно далеко от места, где используется выбранное значение, его отладка   
превратится в весьма захватывающую задачу!  
Возьмите за правило специализировать свои обработчики, насколько это воз-  
можно – пустые предложения except удобны в использовании, но они потенци-  
ально опасны. Так, в последнем примере было бы луч ше использовать предло-  
жение except KeyError:, чтобы более явно обо значить свои намерения и избежать   
возможности перехвата посторон них событий. В более простых сценариях по-  
добные проблемы могут иметь не такое существенное значение, чтобы переве-  
сить удобство ис пользования, но в общем случае универсальные обработчики   
обычно достав ляют массу неприятностей.  
Не перехватывайте слишком мало:   
используйте категории  
С другой стороны, было бы нежелательно делать обработчики слиш ком узко-  
специализированными. Когда в инструкции try перечисля ются конкретные

984   
Глава 35. Использование исключений   
исключения, перехватываться будут только те ис ключения, которые были пе-  
речислены. Это не обязательно плохо, но если в процессе развития программы   
появится новое исключение, вам может потребоваться вернуться и добавить   
это исключение в список обрабатываемых в своем программном коде.  
Мы сталкивались с этой проблемой в предыдущей главе. Например, следую-  
щий обработчик интерпретирует исключения MyExcept1 и MyExcept2 как нор-  
мальную ситуацию, а все остальные – как ошибку. Если в будущем будет до-  
 – как ошибку. Если в будущем будет до-  
– как ошибку. Если в будущем будет до-  
бавлено исключение MyExcept3, оно будет обра батываться как ошибка, если не   
добавить его в список исключений:  
try:  
 ...  
except (MyExcept1, MyExcept2): # Работает неправильно при добавлении MyExcept3  
 ... # Нет ошибки  
else:  
 ... # Рассматривается как ошибка  
К счастью, при осторожном использовании исключений на основе классов, об-  
суждавшихся в главе 33, можно полностью избавиться от этой ловушки. Как   
мы уже видели, если перехватывать общий супер класс, в будущем можно бу-  
дет добавлять и возбуждать более конкрет ные подклассы исключений без не-  
обходимости изменять список ис ключений в предложении except – суперкласс   
становится легко расши ряемой категорией исключений:  
try:  
 ...  
except SuccessCategoryName: # Работает правильно при добавлении MyExcept3  
 ... # Нет ошибки  
else:  
 ... # Рассматривается как ошибка  
Другими словами, порой придется пройти длинный путь, чтобы найти опти-  
мальное решение. Мораль этой истории состоит в том, что вам следует с осо-  
бым тщанием подходить к выбору степени детализации, чтобы обработчики   
исклю чений не были ни слишком универсальными, ни слишком узкоспе-  
циализированными. Политика исключений должна быть составной частью   
общего дизайна, особенно в крупных программах.   
Заключение по основам языка  
Поздравляю! Этим разделом заканчивается ваше изучение основ язы ка про-  
граммирования Python. Если вы забрались так далеко, что чи таете эти строки,   
можете смело считать себя Официальным Програм мистом на языке Python   
(и можете не стесняться упоминать о знании этого языка в своих резюме). Вы   
уже видели почти все, что можно уви деть в самом языке, и получили знания   
более глубокие, чем имели многие практикующие программисты на язы-  
ке Python в начале своего пути. Вы изучили встроенные типы, инструкции   
и исключения, а так же инструменты, которые используются для создания   
крупных эле ментов программ (функции, модули и классы). Кроме того, вы   
иссле довали ряд важных проблем, связанных с проектированием, ООП, ар-  
хитектуру программы и многое другое.

Заключение по основам языка   
985  
Набор инструментальных средств языка Python  
Начиная с этого момента, ваша будущая карьера программиста на языке Python   
в значительной степени будет состоять из овладения ин струментальными   
средствами, доступными для прикладного про граммирования на языке   
Python. Это может занять немало времени. Стандартная библиотека, напри-  
мер, содержит сотни модулей, а разра ботчиками сообщества предлагается еще   
больше. Чтобы познакомить ся со всеми этими инструментами, может потребо-  
ваться лет десять, а то и больше, особенно если учесть, что постоянно появля-  
ются новые (можете мне поверить!).  
Вообще говоря, Python обеспечивает следующую иерархию инстру ментальных   
средств:  
Встроенные  
Встроенные типы, такие как строки, списки и словари, помогают быстро   
создавать несложные программы.  
Расширения на языке Python  
Для решения более сложных задач вы можете расширить возможно сти   
Python своими собственными функциями, модулями и классами.  
Компилируемые расширения  
Хотя мы и не касались данной темы в этой книге, тем не менее, воз можности   
Python можно расширять с помощью модулей, написан ных на других язы-  
ках программирования, таких как C или C++.  
Благодаря такой многоуровневой организации инструментальных средств вы   
можете выбирать, насколько глубоко погружаться в эту ие рархию при созда-  
нии своих программ, – для простых сценариев доста точно будет встроенных   
средств, для крупных программ могут потре боваться дополнительные расши-  
рения на языке Python, а компили руемые расширения – для решения необыч-  
ных задач. В этой книге мы охватили первые две категории, и этого вполне   
достаточно, чтобы на чать писать на языке Python серьезные программы.  
В табл. 35.1 приводятся некоторые из встроенных и других функцио нальных   
возможностей, доступных в языке Python, исследованием которых вы будете   
заниматься остаток вашей карьеры программиста на языке Python. До на-  
стоящего момента наши примеры были очень маленькими и самостоятель-  
ными. Главная их цель состояла в том, чтобы помочь вам освоить основы. Но   
теперь, когда вы узнали все о ба зовом языке, настало время учиться использо-  
вать встроенные интер фейсы Python, чтобы быть в состоянии выполнять на-  
стоящую работу. Вы обнаружите, что такой простой язык, как Python, делает   
решение наиболее распространенных задач намного более легким делом, чем   
можно было бы ожидать.  
Таблица 35.1. Категории инструментальных средств в языке Python  
Категория  
Примеры  
Типы объектов  
Списки, словари, файлы, строки  
Функции  
len, range, open  
Исключения  
IndexError, KeyError

986   
Глава 35. Использование исключений   
Категория  
Примеры  
Модули  
os, tkinter, pickle, re  
Атрибуты  
\_\_dict\_\_, \_\_name\_\_, \_\_class\_\_  
Внешние инструменты  
NumPy, S�IG, Jython, IronPython, Django и др.  
Инструменты разработки крупных проектов  
Как только вы овладеете основами языка, вы обнаружите, что ваши програм-  
мы становятся существенно больше, чем примеры, с которы ми вы эксперимен-  
тировали до сих пор. Для разработки крупных про грамм и в Python, и в общем   
доступе имеется целый набор инструмен тов разработки. Некоторые из них вы   
видели в действии, некоторые я только упомянул. Чтобы помочь вам на вашем   
нелегком пути, я при веду краткое описание некоторых наиболее часто исполь-  
зуемых инст рументов:  
PyDoc и строки документирования  
Функция help и HTML-интерфейсы модуля PyDoc были представле ны в гла-  
ве 15. Модуль PyDoc реализует систему документирования для модулей   
и объектов и интегрирован со строками документиро вания. Это стандарт-  
ная часть системы Python, поэтому за дополни тельными подробностями об-  
ращайтесь к справочному руководству по библиотеке. Кроме того, в главе   
4 даются подсказки с указани ем на источники документации и другие ин-  
формационные ресурсы по языку Python.  
PyChecker  
Python – это динамический язык программирования, поэтому не которые   
ошибки сложно обнаружить, пока программа не будет запущена (напри-  
мер, синтаксические ошибки можно выявить при запуске или во время им-  
портирования файла). Это не такой боль шой недостаток – как и для боль-  
шинства языков программирова ния, это лишь означает, что прежде чем   
распространять свой про граммный код, его необходимо тестировать. При   
использовании языка Python этап компиляции замещается этапом началь-  
Python этап компиляции замещается этапом началь-  
 этап компиляции замещается этапом началь-  
ного тестирования. Кроме того, динамиче ская природа языка Python, авто-  
матический вывод сообщений об ошибках и модель исключений позволяют   
быстрее и проще отыскивать и исправлять ошибки, чем в других языках   
программирования (например, в отличие от языка C, интерпрета тор Python   
не вызывает крах системы при появлении ошибок).  
Системы PyChecker и PyLint обеспечивают возможность выявления   
широко го круга наиболее часто встречающихся ошибок еще до того, как   
сценарий будет запущен. Они играют роль, похожую на ту, какую игра-  
ет программа lint в разработке на языке C. Некоторые кол лективы разра-  
ботчиков проверяют свой программный код на языке Python с помощью   
PyChecker еще до его тестирования или распро странения, чтобы выявить   
все скрытые проблемы. В действитель ности даже стандартная библиоте-  
ка языка Python регулярно прове ряется с помощью PyChecker перед вы-  
пуском. PyChecker и PyLint – это сто ронние пакеты, распространяемые   
с открытыми исходными текста ми. Найти их можно по адресу http://www.  
python.org, на веб-сайте проекта PyPI или с помощью поисковой системы.  
Таблица 35.1. (продолжение)

Заключение по основам языка   
987  
PyUnit (он же unittest)  
В главе 24 мы видели, как в файлы модулей добавляется программный   
код самопроверки, который использует результат проверки \_\_name\_\_ ==   
‘\_\_main\_\_’. Дополнительно для нужд тестиро вания в состав Python входят   
два инструмента. Первый, PyUnit (в руководстве по библиотеке называет-  
ся unittest), обеспечивает комплект классов, с помощью которых можно   
определить и настро ить варианты тестов и указать ожидаемые результаты.   
Он напоми нает библиотеку JUnit в языке Java. Это сложная основанная на   
классах система, подробное описание которой вы найдете в спра вочном ру-  
ководстве по библиотеке Python.  
doctest  
Модуль doctest, входящий в состав стандартной библиотеки, реали зует   
второй и более простой подход к регрессивному тестированию. Он основан   
на использовании строк документирования в языке Py thon. В первом при-  
ближении, чтобы воспользоваться модулем doctest, следует скопировать   
результаты тестирования в интерак тивном сеансе в строки документиро-  
вания в файле с исходным тек стом. После этого модуль doctest извлечет   
эти строки документиро вания, вычленит из них описание тестов с ожи-  
даемыми результата ми и повторно выполнит тесты, чтобы сравнить полу-  
ченные резуль таты с ожидаемыми. Функциональные возможности doctest   
могут использоваться разными способами, о чем подробнее рассказывает ся   
в справочном руководстве по стандартной библиотеке Python.   
Интегрированные среды разработки  
В главе 3 мы уже рассматривали интегрированные среды разработки для   
языка Python. Такие интегрированные среды, как IDLE, обеспе чивают гра-  
фический интерфейс для редактирования, запуска, от ладки и просмотра   
программ на языке Python. Некоторые мощные интегрированные среды   
разработки (такие как Eclipse, Komodo, NetBeans и �ing) поддерживают   
решение дополнительных задач разработки, включая интеграцию с систе-  
мами контроля версий, интерактивные построи тели графического интер-  
фейса, создание файлов проектов и многих других. Список интегрирован-  
ных сред разработки и построителей графических интерфейсов для языка   
Python вы найдете в главе 3, а также на сайтах http://www.python.org и с по-  
мощью поисковых систем.  
Профилировщики  
Поскольку язык Python является высокоуровневым и динамиче ским язы-  
ком программирования, интуитивные представления о производительно-  
сти, которые следуют из опыта работы с другими языками программирова-  
ния, неприменимы к программному коду на языке Python. Чтобы выявить   
узкие места в программе, вам не обходимо добавить логику, выполняющую   
замеры временных ин тервалов с помощью инструментов, определяемых   
в модулях time или timeit, или запустить свой программный код под управ-  
лением модуля profile. Мы уже видели модули измерения времени в дейст-  
вии, когда сравнивали скорость работы итерационных инструмен тов в гла-  
ве 20. Профилирование – это обычно самый первый шаг, который выполня-  
ется на этапе оптимизации, позволяющий выявить узкие места, после чего   
можно начинать поиск альтернативных, более производительных решений.

988   
Глава 35. Использование исключений   
Модуль profile – это модуль стандартной библиотеки, который реа лизует   
профилирование исходных текстов программ на языке Py thon. Он выпол-  
няет строку, которую вы ему передадите (например, импорт файла или вы-  
зов функции), и затем по умолчанию выводит в стандартный поток отчет,   
в котором собрана информация о производительности – количество вызовов   
каждой функции, вре мя, потраченное каждой функцией, и многое другое.   
Модуль profile может запускаться как самостоятельный сценарий, импор-  
тироваться и допускает возможность дополнительной настройки – напри-  
мер, он может сохранять полученную информацию в файле для последую-  
щего анализа с помощью модуля pstats. Чтобы выполнить профилирование   
в интерактивном режиме, импортируйте модуль profile и вызовите функ-  
цию profile.run(‘code’), передав ей строку с программным кодом (например,   
вызов функции или инструкцию импортирования целого модуля), произво-  
дительность которого требуется оценить. Чтобы выполнить профилирова-  
ние из системной командной оболочки, можно воспользоваться командой   
вида python -m profile main.py args... (подробнее формат этой команды описы-  
(подробнее формат этой команды описы-  
подробнее формат этой команды описы-  
 формат этой команды описы-  
формат этой команды описы-  
 этой команды описы-  
этой команды описы-  
 команды описы-  
команды описы-  
 описы-  
описы-  
вается в приложении A).   
Описание других инструментов профилирования вы найдете в руковод-  
стве по стандартной библиотеке Python – модуль cProfile, например, имеет   
практически тот же интерфейс, что и модуль profile, но более низкие на-  
кладные расходы, вследствие чего он лучше подходит для профилирования   
программ, выполнение которых занимает длительное время.   
Отладчики  
В главе 3 мы также обсуждали возможные способы отладки (смотрите врез-  
ку «Отладка программ на языке Python»). Многие интегрированные среды   
разработки поддерживают отладку с использованием графического интер-  
фейса. Кроме того, стандартная библиотека языка Python включает модуль   
отладчика исходных текстов с именем pdb. Этот модуль работает подобно от-  
ладчику командной строки в языке C (например, dbx, gdb):   
Подобно профилировщику отладчик pdb может запускаться в интерактив-  
ном режиме, из системной командной строки, импортироваться как модуль   
или вызываться, как функ ция из модуля pdb (например, pdb.run(“main()”)),   
после чего можно вводить ко манды отладчика в интерактивном режиме   
pdb. Чтобы запустить pdb из системной командной строки, можно восполь-  
зоваться командой вида python -m pdb main.py args... (подробнее формат этой   
команды описывается в приложении A). Кроме того, отладчик pdb включа-  
ет полезную функцию для проведения послеаварийного анализа – pdb.pm(),   
которая позволяет выполнять отладку после появ ления исключения.   
Многие интегрированные среды разработки, такие как IDLE, включают   
интерфейсы «указал и щелкнул», поэто му pdb в наши дни используется от-  
носительно редко, за исключением случаев, когда графический интерфейс   
недоступен или когда требуется более полный контроль над процессом от-  
ладки. Советы по ис пользованию отладчика с графическим интерфейсом   
в IDLE вы найдете в главе 3. Честно говоря, отладчик pdb и интегриро-  
ванные среды разработки тоже используются не слишком часто. Как от-  
мечалось в главе 3, большинство практикующих программистов предпочи-  
тают отлаживать свой программный код вставкой инструкций print в кри-

Заключение по основам языка   
989  
тических точках или просто просматривают сообщения об ошибках (не   
самый современный способ, зато самый быстрый!).  
Варианты распространения  
В главе 2 были представлены инструменты, используемые для упа ковки   
программ на языке Python. Такие инструменты, как py2exe, PyInstaller   
и freeze, могут упаковывать байт-код и виртуальную ма шину с интерпрета-  
тором Python в «фиксированные двоичные фай лы», способные выполнять-  
ся, как самостоятельные про граммы. Они не требуют установки Python   
и полностью скрывают про граммный код. Кроме того, в главе 2 мы узна ли,   
что программы на языке Python могут распространяться в виде исходных   
текстов (.py) или в виде байт-кода (.pyc), а также о суще ствовании программ-  
ных ловушек, обеспечивающих возможность реализации специализиро-  
ванных приемов работы с пакетами, та ких как автоматическое извлечение   
файлов из архивов в формате .zip и шифрование байт-кода.   
Мы также познакомились с модулем distutils, входящим в состав стан-  
дартной библиотеки, который обеспечивает упаковку модулей и пакетов   
на языке Python и рас ширений, написанных на языке C, – за дополни-  
тельной информа цией обращайтесь к справочным руководствам по языку   
Python. Недавно появившаяся в языке Python сторонняя система подготов-  
ки дистри бутивов «eggs» представляет собой другую альтернативу, которая   
позволяет учитывать зависимости, – дополнительную информацию о ней   
ищите в Сети.  
Способы оптимизации  
Существует два основных инструмента оптимизации программ на языке   
Python. В главе 2 была описана система Psyco, позволяющая оптимизиро-  
вать программы по скорости выполнения. Она предоставляет дина мический   
компилятор, выполняющий трансляцию байт-кода в дво ичный машинный   
код, и Shedskin – транслятор исходных текстов с языка Python на язык   
C++. Иногда вам могут встретиться файлы .pyo с оптимизи рованным байт-  
кодом, которые создаются при запуске интерпрета тора Python с ключом   
командной строки –O (обсуждается в главах 21 и 33), но так как этот способ   
обеспечивает весьма скромное увеличение производительности, он обычно   
не используется.   
Наконец, для по вышения производительности можно отдельные части сво-  
их про грамм перенести на компилирующий язык программирования, та-  
кой как C, – подробнее о расширениях на языке C рассказывается в книге   
«Программирование на Python» и в стандартных руководствах по языку   
Python. Вообще говоря, скорость работы интерпретатора Py thon постоянно   
увеличивается, поэтому старайтесь использовать самую свежую его вер-  
сию, когда это возможно.  
Другие советы по разработке крупных проектов  
Кроме всего прочего, в этой книге мы познакомились с различными особен-  
ностями языка, удобство которых особенно ярко проявляется при работе   
с крупными проектами. Среди них: пакеты модулей (глава 23)� исключе-  
ния на основе классов (глава 33)� псевдочастные ат рибуты класса (гла-  
ва 30)� строки документирования (глава 15)� фай лы с настройками пути   
поиска модулей (глава 21)� сокрытие имен, импортируемых инструкцией

990   
Глава 35. Использование исключений   
from \* с помощью списков \_\_all\_\_ и имен в формате \_X (глава 24)� добавле-  
ние программного кода само проверки с использованием приема \_\_name\_\_ ==   
‘\_\_main\_\_’ (глава 24)� использование общих правил проектирования при соз-  
дании функ ций и модулей (главы 17, 19 и 24), использование шаблонов ООП   
(глава 30 и другие) и т. д.  
Узнать о других общедоступных разнообразных инструментах разра ботки   
можно на страницах веб-сайта PyPI, по адресу http://www.python.org и на дру-  
гих ресурсах в Сети.  
В заключение  
Эта глава завершила часть книги, описывающую исключения, кратким об-  
зором типичных случаев использования ис ключений и инструментальных   
средств разработки.   
Кроме того, этой главой завершается изучение основ языка программирования   
Python. К настоящему моменту вы познакомились с полным набором возмож-  
ностей языка Python, которые используются большинством программистов.   
Фактически коль скоро вы забрались так далеко, что чи таете эти строки, мо-  
жете смело считать себя Официальным Програм мистом на языке Python. Обя-  
зательно приобретите футболку, когда в следующий раз посетите сайт проекта   
Python.   
Следующая, заключительная часть книги представляет собой сборник глав,   
описывающих более сложные темы, которые так или иначе относятся к осно-  
вам языка. Все эти главы не являются обязательными к прочтению, потому   
что далеко не всем программистам на языке Python приходится сталкивать-  
Python приходится сталкивать-  
 приходится сталкивать-  
ся с обсуждаемыми в них проблемами, – на самом деле, большинство из вас   
может остановиться на этой главе и начать исследовать возможности языка   
Python в сфере прикладного программирования. Честно говоря, знание при-  
кладных библиотек на практике нередко оказывается важнее, чем знание до-  
полнительных (иногда весьма необычных) особенностей языка.  
С другой стороны, если вам действительно необходимы знания, которые при-  
годятся при работе со строками Юникода, с двоичными данными, при исполь-  
зовании таких инструментов конструирования прикладных интерфейсов, как   
дескрипторы, декораторы и метаклассы, или вам просто хочется изучить воз-  
можности языка еще глубже, то следующая часть книги поможет вам в этом   
начинании. Объемные примеры в заключительной части дадут вам возмож-  
ность увидеть, как на практике применять уже знакомые вам концепции.  
Поскольку этой главой завершается изучение основ языка программирования   
Python, она заканчивается единственным контрольным вопросом. Как обыч-  
но, обязательно выполните упражнения к этой части, чтобы закрепить знания,   
полученные в последних нескольких главах. Поскольку следующая часть не   
является обязательной к прочтению, в конце нее уже не будет упражнений для   
самостоятельного решения. Если вы захотите увидеть некоторые примеры,   
как концепции, которые мы обсуждали в этой книге, объединяются в реаль-  
ных программах, загляните в «решение» упражнения 4 в приложении B.

Закрепление пройденного   
991  
Закрепление пройденного  
Контрольные вопросы  
1. (Этот вопрос уже задавался в контрольных вопросах к главе 1 – ви дите,   
я же говорил, что это будет просто.) Почему слово «spam» так часто появля-  
ется в примерах в этой книге?  
Ответы  
1. Язык Python получил свое название в честь английской комик-группы Мон-  
ти Пайтона (Monty Python) (согласно опросам, которые я проводил среди   
своих студентов, это «самая большая тайна» в мире Python). Слово «spam»   
взято из пародии Монти Пайтона (Monty Python), где герои сериала пыта-  
ются заказать блюдо в кафе терии, а их заглушает хор викингов, поющих   
песню о консервах (spam). И если бы я мог вставить сюда аудиофрагмент из   
этой песни в качестве заключительных титров, я бы сделал это.  
Упражнения к седьмой части  
Мы достигли конца этой части книги, поэтому настало время выпол нить не-  
сколько упражнений на применение исключений, чтобы по практиковаться   
в основах. Исключения действительно являются очень простым инструмен-  
том – если вы пользуетесь ими, значит вы владеете ими в полной мере.  
Решения вы найдете в приложении B в разделе «Часть VII. Исключе ния и ин-  
струменты».  
1. try/except. Напишите функцию с именем oops, которая при вызове явно   
возбуждает исключение IndexError. Затем напишите другую функцию, вы-  
зывающую функцию oops внутри инструкции try/except, которая перехва-  
тывает ошибку. Что произойдет, если изме нить функцию oops так, чтобы   
вместо IndexError она возбуждала ис ключение KeyError? Где располагаются   
имена KeyError и IndexError? (Подсказка: вспомните, что все простые неква-  
лифицированные имена находят ся в одной из четырех областей видимости   
согласно правилу LEGB.)  
2. Объекты исключений и списки. Измените функцию oops, которую вы толь-  
ко что написали так, чтобы она возбуждала ваше собствен ное исключение   
с именем MyError. Определите свое исключение в виде класса. Затем расширь-  
те инструкцию try в функции, которая вызывает функцию oops, так чтобы   
кроме исключения IndexError она перехватывала бы еще и это исключе ние   
и выводила бы перехваченный экземпляр на эк ран.   
3. Обработка ошибок. Напишите функцию safe(func, \*args), которая запуска-  
ет указанную функцию func, передавая ей произвольное количество аргу-  
ментов с использованием синтаксиса \*name, перехватывает любые исклю-  
чения, возникающие в ходе выполнения этой функции и выводит инфор-  
мацию об исключении с использованием функции exc\_info из модуля sys.   
Затем с помощью своей функции safe запус тите функцию oops из упраж-  
нения 1 или 2. Поместите функцию safe в модуль с именем tools.py и пе-  
редайте ей функцию oops в инте рактивном режиме. Какие сообщения об   
ошибках вы получили? Наконец, расширьте свою функцию safe так, чтобы   
при возникно вении исключения она выводила содержимое стека вызовов

992   
Глава 35. Использование исключений   
с помо щью встроенной функции print\_exc, расположенной в стандартном   
модуле traceback (за дополнительной информацией обращайтесь к руковод-  
ству по библиотеке языка Python).  
4. Примеры  для  самостоятельного  изучения. В конец приложения B я до-  
бавил несколько примеров сценариев, разработанных в ходе выполнения   
упражнений моими студентами, чтобы вы могли само стоятельно изучить   
и опробовать их, попутно изучая набор стан дартных руководств по языку   
Python. Эти примеры не содержат описаний, и в них используются инстру-  
менты из стандартной биб лиотеки языка Python, которые вам потребуется   
исследовать самостоятельно. Для многих читателей эти примеры помогут   
увидеть, как концепции, которые мы обсуждали в этой книге, объединя-  
ются в реальных программах. Если эти примеры возбудят у вас интерес,   
вы сможете отыскать множество более реалистичных примеров программ   
на языке Python в последующих книгах, таких как «Программирование на   
Py thon», и в Сети.

Часть   
VIII.  
Расширенные возможности

Глава 36.  
   
Юникод и строки байтов  
В главе, посвященной строкам (глава 7), я преднамеренно ограничил круг   
тем, касающихся строк, освещением только тех особенностей, которые нужны   
большинству программистов, использующих язык Python. Поскольку пода-  
Python. Поскольку пода-  
. Поскольку пода-  
вляющему большинству программистов приходится иметь дело только с про-  
стыми формами текста, такими как текст ASCII, они вполне могут ограничить-  
ASCII, они вполне могут ограничить-  
, они вполне могут ограничить-  
ся использованием базового строкового типа str языка Python и связанными   
с ним операциями и не нуждаются в знании расширенных строковых концеп-  
ций. Фактически эти программисты вообще могут игнорировать изменения   
в Python 3.0, коснувшиеся строк, и продолжать использовать приемы работы   
со строками, какие они использовали раньше.  
C другой стороны, некоторым программистам приходится иметь дел со спе-  
 другой стороны, некоторым программистам приходится иметь дел со спе-  
циализированными типами данных: с наборами символов, не входящих в диа-  
пазон ASCII, с файлами изображений и так далее. Для таких программистов   
(и других, которые могут в будущем примкнуть к ним), в этой главе восполня-  
ется недостаток информации о строках и рассматриваются некоторые расши-  
ренные концепции модели строк в языке Python.  
В частности, здесь мы исследуем основы поддержки Юникода в языке Python –   
строк, состоящих из многобайтовых символов, используемых в интернациона-  
лизированных приложениях, а также двоичных данных – строк, представля-  
ющих байты по их абсолютным значениям. Как мы увидим далее, реализация   
представления строк изменилась в последних версиях Python:  
 •  
В Python 3.0 имеется альтернативный строковый тип для представления   
двоичных данных, а поддержка Юникода обеспечивается обычным строко-  
вым типом (символы ASCII интерпретируются как разновидность символов   
Юникода).  
 •  
В Python 2.6 для представления Юникода используется альтернативный   
строковый тип, а поддержка простых текстовых строк и двоичных данных   
обеспечивается обычным строковым типом.  
Кроме того, так как модель представления строк в языке Python напрямую   
определяет порядок обработки файлов с данными, не являющимися символа-  
ми ASCII, мы также исследуем здесь фундаментальные принципы, имеющие   
отношение к этой теме. Наконец, мы коротко рассмотрим некоторые дополни-

996   
Глава 36. Юникод и строки байтов   
тельные инструменты для работы со строками и двоичными данными, такие   
как поиск по шаблону, сериализация объектов, упаковка двоичных данных   
и синтаксический анализ разметки XML, а также влияние на них изменений   
в Python 3.0, связанных со строками.  
Эта глава относится к разделу, где рассматриваются дополнительные, расши-  
ренные темы, потому что далеко не всем программистам требуется вникать   
в тонкости кодирования символов Юникода или в особенности работы с дво-  
ичными данными. Однако если вам когда-либо потребуется заниматься обра-  
боткой таких данных, вы увидите, что строковая модель в языке Python предо-  
Python предо-  
 предо-  
ставляет всю необходимую для этого поддержку.  
Изменения в Python 3.0, касающиеся строк  
Одним из наиболее значительных изменений в версии 3.0 является изменение   
в объектах строкового типа. Если говорить кратко, типы str и unicode, имев-  
шиеся в версии 2.X, в версии 3.0 были преобразованы в типы str и bytes, и по-  
явился новый изменяемый тип bytearray. Строго говоря, тип bytearray досту-  
пен и в Python 2.6 (но не доступен в более ранних версиях), однако реализация   
поддержки этого типа данных является результатом переноса из версии 3.0,   
и в версии 2.6 этот тип данных не делает четких различий между текстовыми   
и двоичными данными.  
Эти изменения могут оказать весьма существенное влияние на программный   
код, особенно если вам приходится заниматься обработкой Юникода или дан-  
ных, по своей природе являющихся двоичными. Фактически степень значи-  
мости этой темы для вас зависит от того, какие из следующих задач вам при-  
ходится решать:  
 •  
Если вам приходится иметь дело с текстом Юникода, например в интерна-  
ционализированных приложениях или в результатах, возвращаемых не-  
которыми парсерами разметки XML, вы обнаружите, что поддержка коди-  
XML, вы обнаружите, что поддержка коди-  
, вы обнаружите, что поддержка коди-  
ровок символов в версии 3.0 изменилась, но при этом стала более простой   
и прозрачной, чем в Python 2.6.  
 •  
Если вам приходится иметь дело с двоичными данными, например в фор-  
ме изображений или аудиоданных, или обрабатывать упакованные данные   
с помощью модуля struct, вам потребуется поближе познакомиться с новым   
типом данных bytes, появившимся в Python 3.0, а также понять, что в вер-  
Python 3.0, а также понять, что в вер-  
 3.0, а также понять, что в вер-  
сии 3.0 текстовые строки и двоичные данные различаются более четко.  
 •  
Если решаемые вами задачи не относятся ни к одной из двух предыдущих   
категорий, то вы можете просто использовать строки Python 3.0 практиче-  
Python 3.0 практиче-  
 3.0 практиче-  
ски так же, как в 2.6: использовать обобщенный строковый тип str, тексто-  
вые файлы и все, уже знакомые вам, строковые операции, изученные нами   
ранее. Ваши строки будут кодироваться и декодироваться с учетом коди-  
ровки, используемой в системе по умолчанию (в США это ASCII или UTF-8   
в �indows� кодировку по умолчанию можно определить с помощью функ-  
�indows� кодировку по умолчанию можно определить с помощью функ-  
� кодировку по умолчанию можно определить с помощью функ-  
ции sys.getdefaultencoding()), но для вас это, скорее всего, просто пройдет   
незамеченным.  
Другими словами, если вы всегда работаете с текстом, состоящим только из   
символов ASCII, вы можете использовать обычные строковые объекты и тек-  
ASCII, вы можете использовать обычные строковые объекты и тек-  
, вы можете использовать обычные строковые объекты и тек-  
стовые файлы и избежать необходимости вникать в подробности, о которых   
рассказывается далее. Как мы вскоре увидим, ASCII – это простейшая разно-  
ASCII – это простейшая разно-  
 – это простейшая разно-

Основы строк   
997  
видность Юникода и подмножество других кодировок, поэтому строковые опе-  
рации и текстовые файлы «просто работают», если программа обрабатывает   
текст, состоящий только из символов ASCII.  
Однако даже если решаемые вами задачи относятся к последней из трех преды-  
дущих категорий, понимание основ модели строк в Python 3.0 поможет вам не   
только прояснить для себя особенности поведения строк, но и быстрее овладеть   
особенностями работы с Юникодом или двоичными данными, когда это потре-  
буется.  
Поддержка Юникода и двоичных данных присутствует также в Python 2.6,   
хотя и имеет несколько иные формы. Основное наше внимание в этой главе бу-  
дет сосредоточено на строковых типах, имеющихся в Python 3.0, тем не менее,   
попутно мы исследуем некоторые отличия от версии 2.6. Независимо от того,   
какую версию вы используете, инструменты, исследуемые здесь, могут иметь   
большое значение для программ самых разных типов.  
Основы строк  
Прежде чем перейти к программному коду, давайте прежде познакомимся   
с моделью строк, реализованной в языке Python. Чтобы понять, почему в Py-  
Py-  
thon 3.0 изменился подход к работе со строками, для начала нужно узнать, как   
представляются символы в компьютерах.  
Кодировки символов  
Большинство программистов представляют себе текстовые строки как после-  
довательности символов. Однако в памяти компьютера символы могут пред-  
ставляться различными способами, в зависимости от того, какой набор симво-  
лов используется.  
Стандарт ASCII был выработан в США и для многих американских програм-  
мистов определяет понятие текстовых строк. Стандарт ASCII определяет набор   
символов с кодами в диапазоне от 0 до 127, что позволяет сохранять каждый   
символ в одном 8-битовом байте (в котором фактически используется только 7   
младших битов). Например, согласно стандарту ASCII символу латиницы ‘a’   
соответствует целочисленное значение 97 (0x61 – в шестнадцатеричном пред-  
x61 – в шестнадцатеричном пред-  
61 – в шестнадцатеричном пред-  
ставлении), которое занимает единственный байт в памяти компьютера и в   
файлах. Если вам интересно узнать, как действует этот стандарт, можете поэк-  
спериментировать со встроенной функцией ord, которая возвращает целочис-  
ленный код символа, и с функцией chr, возвращающей символ по указанному   
целочисленному коду:  
>>> ord(‘a’) # ‘a’ – байт с целочисленным значением 97 в ASCII  
97  
>>> hex(97)  
‘0x61’  
>>> chr(97) # Целочисленному значению 97 соответствует символ ‘a’  
‘a’  
Однако иногда одного байта бывает недостаточно для представления отдель-  
ных символов. Например, в диапазон символов, определяемых стандартом   
ASCII, не попадают различные специальные символы и буквы с диакритиче-  
скими знаками. Некоторые стандарты позволяют использовать все возможные

998   
Глава 36. Юникод и строки байтов   
значения 8-битных байтов, от 0 до 255, чтобы обеспечить возможность пред-  
ставления специальных символов, отображая их в диапазон значений от 128   
до 255 (за пределами диапазона ASCII). Один из таких стандартов, известный   
под названием Latin-1, широко используется в Западной Европе. В стандарте   
Latin-1 коды со значениями выше 127 присвоены символам с диакритически-  
-1 коды со значениями выше 127 присвоены символам с диакритически-  
ми знаками и другим специальным символам. Например, значению 196 в этом   
стандарте соответствует специальный символ, не входящий в набор ASCII:  
>>> 0xC4  
196  
>>> chr(196)  
‘Ä’  
Этот стандарт включает в себя множество дополнительных специальных сим-  
волов. И все равно в некоторых алфавитах так много символов, что нет ника-  
кой возможности представить каждый из них одним байтом. Стандарт Юникод   
(Unicode) обеспечивает более гибкие возможности. Строки Юникода иногда на-  
Unicode) обеспечивает более гибкие возможности. Строки Юникода иногда на-  
) обеспечивает более гибкие возможности. Строки Юникода иногда на-  
зывают строками «многобайтовых символов», потому что каждый символ в та-  
ких строках может быть представлен несколькими байтами. Юникод обычно   
используется в интернационализированных программах, чтобы обеспечить   
возможность представления символов из европейских и азиатских алфавитов,   
которые содержат гораздо больше символов, чем можно было бы представить   
с помощью 8-битных байтов.  
Чтобы хранить такой текст в памяти компьютера, его необходимо транслиро-  
вать в последовательность простых байтов и обратно, используя определенную   
кодировку – набор правил преобразования строк, состоящих из символов Юни-  
кода, в последовательность байтов и извлечения строк из последовательностей   
байтов. Говоря техническим языком, такие преобразования между последова-  
тельностями байтов и строками обозначаются двумя терминами:  
 •  
Кодирование – процесс преобразования строки символов в последователь-  
ность простых байтов в соответствии с желаемой кодировкой.  
 •  
Декодирование – процесс преобразования последовательности байтов в стро-  
ку символов в соответствии с желаемой кодировкой.  
То есть мы кодируем строки в последовательности байтов и декодируем после-  
довательности байтов в строки. Для некоторых кодировок процесс преобразо-  
вания тривиально прост – в кодировках ASCII и Latin-1, например, каждому   
символу соответствует единственный байт, поэтому фактически никакого пре-  
образования не требуется. Для других кодировок процедура отображения мо-  
жет оказаться намного сложнее и порождать по несколько байтов для каждого   
символа.  
Широко используемая кодировка UTF-8, например, позволяет представить   
широкий диапазон символов, используя схему с переменным числом байтов.   
Символы с кодами ниже 128 представляются одним байтом� символы с кодами   
в диапазоне от 128 до 0x7ff (2047) преобразуются в двухбайтовые последова-  
x7ff (2047) преобразуются в двухбайтовые последова-  
7ff (2047) преобразуются в двухбайтовые последова-  
ff (2047) преобразуются в двухбайтовые последова-  
 (2047) преобразуются в двухбайтовые последова-  
тельности, где каждый байт имеет значение от 128 до 255� а символы с кодами   
выше 0x7ff преобразуются в трех- и четырехбайтовые последовательности, где   
каждый байт имеет значение от 128 до 255. Благодаря этой схеме строки с сим-  
волами ASCII остаются компактными, ликвидируются проблемы с порядком   
следования байтов и исключается необходимость использовать байты с нуле-  
вым значением, которые могут вызывать проблемы при работе с библиотеками   
языка C и при организации сетевых взаимодействий.

Основы строк   
999  
Поскольку для сохранения совместимости различные кодировки отображают   
символы в один и тот же диапазон кодов, набор символов ASCII является под-  
ASCII является под-  
 является под-  
множеством обеих кодировок, Latin-1 и UTF-8, – то есть допустимые строки   
символов ASCII также будут считаться допустимыми строками символов в ко-  
ASCII также будут считаться допустимыми строками символов в ко-  
 также будут считаться допустимыми строками символов в ко-  
дировках Latin-1 и UTF-8. То же относится и к данным в файлах: все текстовые   
файлы, состоящие из символов ASCII, будут считаться допустимыми тексто-  
ASCII, будут считаться допустимыми тексто-  
, будут считаться допустимыми тексто-  
выми файлами, с точки зрения кодировки UTF-8, потому что ASCII – это под-  
UTF-8, потому что ASCII – это под-  
-8, потому что ASCII – это под-  
ASCII – это под-  
 – это под-  
множество 7-битных символов в кодировке UTF-8.  
И наоборот, кодировка UTF-8 сохраняет двоичную совместимость с ASCII для   
всех символов с кодами ниже 128. Кодировки Latin-1 и UTF-8 просто включают   
дополнительные символы: Latin-1 включает дополнительные символы с кода-  
Latin-1 включает дополнительные символы с кода-  
-1 включает дополнительные символы с кода-  
ми в диапазоне от 128 до 255, отводя для каждого символа один байт, а коди-  
ровка UTF-8 включает символы, которые могут быть представлены несколь-  
UTF-8 включает символы, которые могут быть представлены несколь-  
-8 включает символы, которые могут быть представлены несколь-  
кими байтами. Другие кодировки обеспечивают расширение наборов символов   
похожими способами, но все эти наборы – ASCII, Latin-1, UTF-8 и многие дру-  
ASCII, Latin-1, UTF-8 и многие дру-  
, Latin-1, UTF-8 и многие дру-  
Latin-1, UTF-8 и многие дру-  
-1, UTF-8 и многие дру-  
UTF-8 и многие дру-  
-8 и многие дру-  
гие – рассматриваются как Юникод.  
С точки зрения программиста на языке Python, кодировки определяются как   
строки, содержащие названия кодировок. Язык Python поддерживает при-  
Python поддерживает при-  
 поддерживает при-  
мерно 100 различных кодировок – полный список вы найдете в справочни-  
ке по стандартной библиотеке Python. Если импортировать модуль encodings   
и вызвать функцию help(encodings), кроме всего прочего можно также увидеть   
список из множества названий кодировок� некоторые из них реализованы на   
языке Python, некоторые – на языке C. Отдельные кодировки имеют несколько   
названий, например: latin-1 и iso8859\_1 – это синонимы одной и той же коди-  
ровки Latin-1. Мы еще вернемся к кодировкам, ниже в этой главе, когда будем   
рассматривать способы записи строк Юникода в сценариях.  
За дополнительной информацией о Юникоде обращайтесь к стандартному на-  
бору руководств по языку Python. В разделе «Python HO�TOs» вы найдете до-  
Python. В разделе «Python HO�TOs» вы найдете до-  
. В разделе «Python HO�TOs» вы найдете до-  
Python HO�TOs» вы найдете до-  
 HO�TOs» вы найдете до-  
HO�TOs» вы найдете до-  
» вы найдете до-  
кумент «Unicode HO�TO», где приводятся дополнительные сведения, опущен-  
Unicode HO�TO», где приводятся дополнительные сведения, опущен-  
 HO�TO», где приводятся дополнительные сведения, опущен-  
HO�TO», где приводятся дополнительные сведения, опущен-  
», где приводятся дополнительные сведения, опущен-  
ные здесь для экономии места.  
Типы строк в Python  
Если говорить более конкретно, язык программирования Python предоставля-  
Python предоставля-  
 предоставля-  
ет строковые типы для представления текстовых данных в сценариях. Какие   
именно строковые типы вы будете использовать, зависит от версии Python.   
В Python 2.X имеется общий строковый тип, обеспечивающий возможность   
представления двоичных данных и 8-битных символов, таких как символы   
ASCII, а также более специализированный тип, предназначенный для пред-  
, а также более специализированный тип, предназначенный для пред-  
ставления строк, состоящих из многобайтовых символов Юникода:  
 •  
str – для представления текстовых строк из 8-битных символов и двоичных   
данных  
 •  
unicode – для представления текстовых строк из многобайтовых символов   
Юникода  
Эти два строковых типа в Python 2.X отличаются между собой (тип unicode   
может представлять многобайтовые символы и обладает дополнительной под-  
держкой операций кодирования и декодирования), однако они имеют схожие   
наборы операций. Строковый тип str в Python 2.X используется для представ-  
Python 2.X используется для представ-  
 2.X используется для представ-  
X используется для представ-  
 используется для представ-  
ления текстовых данных, которые могут быть представлены 8-битными бай-

1000   
Глава 36. Юникод и строки байтов   
тами, а также двоичных данных, которые представлены абсолютными значе-  
ниями байтов.  
Напротив, в Python 3.X имеется три строковых типа – один служит для пред-  
ставления текстовых данных и два – для представления двоичных данных:  
 •  
str – для представления текстовых строк, состоящих из символов Юникода   
(как 8-битных, так и многобайтовых)  
 •  
bytes – для представления двоичных данных  
 •  
bytearray – изменяемая версия типа bytes  
Как уже упоминалось выше, тип bytearray также имеется в Python 2.6, однако   
реализация поддержки этого типа данных является результатом переноса из   
версии 3.0 – он не делает четких различий для содержимого и вообще считает-  
ся типом данных, присущим версии 3.0.   
Все три строковых типа в версии 3.0 поддерживают похожие наборы операций,   
но действуют они по-разному. Основная цель изменений в версии 3.X состоя-  
X состоя-  
 состоя-  
ла в том, чтобы объединить строковые типы, используемые в версиях 2.X, для   
представления обычного текста и текста, состоящего из символов Юникода,   
в один строковый тип, поддерживающий обычные символы и символы Юнико-  
да: разработчики хотели устранить разделение строк в 2.X и сделать работу со   
строками Юникода более естественной. Учитывая, что символы ASCII и дру-  
ASCII и дру-  
 и дру-  
гие 8-битные символы в действительности являются лишь подмножеством   
символов Юникода, такое объединение выглядит вполне логичным.  
Для этого тип str в версии 3.0 определен как неизменяемая  последователь-  
ность символов (не обязательно байтов), которая может содержать обычный   
текст, состоящий из символов ASCII, по одному байту на символ, или текст,   
состоящий из многобайтовых символов Юникода, например из набора UTF-8.   
При обработке в сценариях строки этого типа по умолчанию кодируются в со-  
ответствии с настройками системы, однако имеется возможность явно указы-  
вать название кодировки для преобразования объектов типа str как находя-  
щихся в памяти, так и при чтении/записи текстовых файлов.  
Несмотря на то, что в версии 3.0 в новом строковом типе str было достигну-  
то желаемое слияние обычных строк со строками Юникода, тем не менее, во   
многих программах сохраняется потребность выполнять обработку простых   
двоичных данных, которые не должны преобразовываться в текстовое пред-  
ставление. В эту категорию попадают программы, выполняющие обработку   
файлов изображений и аудиофайлов, а также упакованных данных, исполь-  
зуемых для обмена информацией с устройствами или с программами на языке   
C при помощи модуля struct. Поэтому для поддержки возможности обработки   
двоичных данных был введен новый тип данных bytes.  
В версии 2.X потребность в обработке двоичных данных удовлетворял общий   
тип str, потому что в этой версии строки рассматривались просто как последо-  
вательности байтов (для представления строк, состоящих из многобайтовых   
символов, использовался отдельный тип unicode). В версии 3.0 тип bytes опреде-  
лен как неизменяемая последовательность 8-битных целых чисел, представ-  
ляющих абсолютные значения байтов. Кроме того, в версии 3.0 тип bytes под-  
держивает практически тот же набор операций, что и тип str, включая стро-  
ковые методы, операции над последовательностями и даже поиск совпадений   
с помощью модуля re, но не поддерживает операцию форматирования строк.

Основы строк   
1001  
В версии 3.0 объект типа bytes в действительности является последовательно-  
стью коротких целых чисел, каждое из которых имеет значение в диапазоне   
от 0 до 255. Операция извлечения элемента по индексу из последовательности   
типа bytes возвращает объект типа int, операция извлечения среза возвращает   
новый объект типа bytes, а применение встроенной функции list к такому объ-  
екту дает в результате список целых чисел, а не строк. Однако при выполнении   
операций, которые применяются к символьным данным, содержимое объектов   
типа bytes интерпретируется как байты в кодировке ASCII (например, метод   
isalpha будет интерпретировать каждый байт как код символа ASCII). Кроме   
того, для удобства объекты типа bytes выводятся как строки символов, а не как   
последовательности целых чисел.  
Дополнительно разработчики Python добавили в версию 3.0 новый тип bytear-  
ray. Тип bytearray – это разновидность типа bytes, допускающая возможность   
непосредственного изменения объектов в памяти. Он поддерживает обычные   
строковые операции, которые поддерживаются типами str и bytes, а также   
множество операций, изменяющих сам объект, как списки (например, методы   
append и extend и операцию присваивания по индексу). Учитывая, что строки   
могут интерпретироваться как последовательности простых байтов, тип byte-  
array обеспечивает возможность непосредственного изменения строковых дан-  
ных в памяти, что невозможно в Python 2 без преобразования строк в объекты   
изменяемого типа и не поддерживается типами str и bytes в Python 3.0.  
Несмотря на то, что Python 2.6 и 3.0 предлагают практически одни и те же   
функциональные возможности, тем не менее, доступ к ним организован совер-  
шенно по-разному. Фактически строковые типы в Python 2.6 не имеют прямо-  
Python 2.6 не имеют прямо-  
 2.6 не имеют прямо-  
го соответствия в версии 3.0 – тип str в версии 2.6 совмещает в себе типы str   
и bytes из версии 3.0, а тип str в версии 3.0 совмещает в себе типы str и unicode   
из версии 2.6. Кроме того, изменяемость типа bytearray в версии 3.0 вообще яв-  
ляется уникальной.  
Однако на практике эти различия вовсе не выглядят такими кардинальными,   
как может показаться. Все различия сводятся к следующему: в 2.6 тип str ис-  
пользуется для представления текстовых и двоичных данных, а тип unicode –   
для других форм представления текста� в 3.0 тип str используется для пред-  
ставления всех видов текстовых данных (состоящих как из простых символов,   
так и из символов Юникода), а типы bytes и bytearray – для представления дво-  
ичных данных. На практике выбор типа нередко определяется используемы-  
ми инструментами – особенно это относится к инструментам обработки фай-  
лов, о которых рассказывается в следующем разделе.  
Текстовые и двоичные файлы  
Операции ввода-вывода над файлами также претерпели изменение в Py-  
Py-  
thon 3.0. Они учитывают различия между типами str/bytes и автоматически   
поддерживают кодирование символов Юникода. Теперь в языке Python тексто-  
Python тексто-  
 тексто-  
вые и двоичные файлы имеют более существенные различия, не зависящие от   
платформы:  
Текстовые файлы  
Когда файл открывается в текстовом режиме, данные из него при чтении   
декодируются автоматически (с использованием кодировки по умолчанию,   
в зависимости от настроек системы или с использованием явно указанной

1002   
Глава 36. Юникод и строки байтов   
кодировки) и возвращаются в виде объекта типа str. Операции записи при-  
нимают объекты типа str и автоматически кодируют содержащиеся в них   
данные перед записью в файл. Текстовые файлы также поддерживают   
универсальный механизм преобразования символов, обозначающих конец   
строки, и дополнительные аргументы, определяющие порядок кодирова-  
ния. В зависимости от имени используемой кодировки, текстовые файлы   
могут также автоматически обрабатывать последовательности, являющи-  
еся маркерами порядка следования байтов, находящиеся в начале файла   
(подробнее об этом рассказывается чуть ниже).  
Двоичные файлы  
Когда файл открывается в двоичном режиме, добавлением в строку режима   
символа ‘b’ (только в нижнем регистре) в аргументе встроенной функции   
open, данные при чтении не декодируются, а просто возвращаются в своем   
неизменном виде, в виде объекта типа bytes. Операция записи точно так   
же принимает объект bytes и записывает его в файл, не выполняя никаких   
преобразований. Операции над двоичными файлами могут также прини-  
мать объекты типа bytearray и записывать их содержимое в файлы.  
Поскольку теперь типы str и bytes разграничиваются более четко, вам придет-  
ся заранее определять, какую природу имеют ваши данные – текстовую или   
двоичную, и использовать для их представления объекты либо типа str, либо   
bytes. И наконец, режим, в котором открывается файл, определяет тип объ-  
ектов, который должен использоваться для представления содержимого этого   
файла:  
 •  
Если сценарий обрабатывает файлы с изображениями, упакованными дво-  
ичными данными, созданными другими программами, или потоки данных   
от каких-либо устройств, велика вероятность, что вам потребуется исполь-  
зовать объекты типа bytes и открывать файлы в двоичном режиме. У вас   
также имеется дополнительная возможность использовать тип bytearray,   
когда потребуется обновлять данные, не создавая промежуточные копии   
в памяти.  
 •  
Если сценарий обрабатывает данные, имеющие текстовую природу, такие   
как вывод другой программы, разметка HTML, интернационализирован-  
HTML, интернационализирован-  
, интернационализирован-  
ный текст или файлы в форматах CSV и XML, вам наверняка потребуется   
использовать объекты типа str и открывать файлы в текстовом режиме.  
Обратите внимание, что аргумент со строкой режима встроенной функции open   
(второй ее аргумент) имеет особое значение в Python 3.0 – он не только опреде-  
Python 3.0 – он не только опреде-  
 3.0 – он не только опреде-  
ляет режим работы с файлом, но и тип объектов. Добавляя в строку режима   
символ b, вы определяете двоичный режим работы и будете получать или долж-  
ны передавать объекты типа bytes, представляющие содержимое файла в опера-  
циях чтения и записи. Без символа b файлы будут обрабатываться в текстовом   
режиме, и для представления их содержимого вы будете использовать объекты   
типа str. Например, режимы rb, wb и rb+ подразумевают использование типа   
bytes� r, w+ и rt (по умолчанию) подразумевают использование типа str.  
Кроме того, в текстовом режиме файлы могут обрабатывать последователь-  
ность с маркером порядка следования байтов (byte order marker, BOM), кото-  
рый может присутствовать в начале файла при использовании определенных   
схем кодирования символов. Например, в кодировках UTF-16 и UTF-32 маркер   
BOM определяет прямой или обратный порядок следования байтов (big-endi-  
 определяет прямой или обратный порядок следования байтов (big-endi-  
big-endi-  
-endi-  
endi-

Примеры использования строк в Python 3.0   
1003  
an и little-endian соответственно). По сути, порядок следования байтов имеет   
очень большое значение. Текстовые файлы с данными в кодировке UTF-8 так-  
UTF-8 так-  
-8 так-  
же могут включать маркер BOM, просто чтобы указать, что содержимое фай-  
BOM, просто чтобы указать, что содержимое фай-  
, просто чтобы указать, что содержимое фай-  
ла записано именно в кодировке UTF-8, хотя это и не является обязательным.   
Когда выполняются операции чтения и записи с использованием этих схем   
кодирования, интерпретатор автоматически пропускает или записывает мар-  
кер BOM, если он вообще подразумевается кодировкой или если вы указывае-  
BOM, если он вообще подразумевается кодировкой или если вы указывае-  
, если он вообще подразумевается кодировкой или если вы указывае-  
те более конкретное имя кодировки, явно определяющее порядок следования   
байтов. Например, обработка маркера BOM всегда выполняется при использо-  
BOM всегда выполняется при использо-  
 всегда выполняется при использо-  
вании кодировки «utf-16», более специфическая кодировка «utf-16-le» опреде-  
utf-16», более специфическая кодировка «utf-16-le» опреде-  
-16», более специфическая кодировка «utf-16-le» опреде-  
utf-16-le» опреде-  
-16-le» опреде-  
le» опреде-  
» опреде-  
ляет обратный (little-endian) порядок следования байтов в кодировке UTF-16.   
Аналогично более специфическая кодировка «utf-8-sig» вынуждает интерпре-  
utf-8-sig» вынуждает интерпре-  
-8-sig» вынуждает интерпре-  
sig» вынуждает интерпре-  
» вынуждает интерпре-  
татор пропускать при чтении и записывать маркер BOM при вводе и выводе   
соответственно текста в кодировке UTF-8 (обобщенное имя кодировки «utf-8»   
этого не предполагает).  
Мы еще вернемся к маркерам BOM и к файлам в разделе «Обработка маркера   
BOM в Python 3.0», ниже. Но сперва исследуем новую модель строк Юникода.  
Примеры использования строк в Python 3.0  
Рассмотрим несколько примеров, демонстрирующих особенности использова-  
ния строковых типов в Python 3.0. Предварительное замечание: программный   
код, представленный в этом разделе, может запускаться только под управле-  
нием Python 3.0. Однако основные строковые операции совместимы с разны-  
ми версиями Python. Простые строки символов ASCII, представленные в виде   
объекта типа str, одинаково обрабатываются в версиях 2.6 и 3.0 (и в точности   
так, как мы видели в главе 7). Кроме того, несмотря на то, что в Python 2.6   
отсутствует тип bytes (в этой версии используется один обобщенный тип str),   
программный код, использующий этот тип данных, обычно остается работо-  
способным – в 2.6 вызов bytes(X) интерпретируется как синоним str(X), а но-  
вая форма литералов b’...’ интерпретируется так же, как обычный строковый   
литерал ‘...’. Тем не менее в отдельных случаях вы можете столкнуться с не-  
совместимостью версий – в версии 2.6, например, функция bytes не прини-  
мает второй аргумент (название кодировки), который является обязательным   
в версии 3.0.  
Литералы и основные свойства  
В Python 3.0 строковые объекты создаются, когда вы вызываете встроенную   
функцию, такую как str или bytes, манипулируете объектом файла, создан-  
ным вызовом функции open (описывается в следующем разделе) или опреде-  
ляете строковый литерал в тексте сценария. В последнем случае для создания   
объектов типа bytes, в версии 3.0 была введена новая форма литерала b’xxx’   
(и эквивалентная ей: B’xxx’), а объекты типа bytearray могут создаваться вызо-  
вом функции bytearray с различными аргументами.  
Если говорить более формально, в версии 3.0 все текущие формы определения   
строковых литералов – ‘xxx’, “xxx” и текст в тройных кавычках – генериру-  
ют объект типа str. Добавление символа b или B перед открывающей кавыч-  
кой в любой из этих форм приводит к созданию объекта типа bytes. Эта новая

1004   
Глава 36. Юникод и строки байтов   
форма b’...’ литерала bytes похожа на форму r’...’ «сырой» строки, которая   
использовалась, чтобы избежать необходимости экранировать символы обрат-  
ного слеша. Рассмотрим следующий сеанс работы с интерактивной оболочкой   
Python 3.0:  
C:\misc> c:\python30\python  
>>> B = b’spam’ # Создаст объект bytes (8-битные байты)  
>>> S = ‘eggs’ # Создаст объект str   
 # (символы Юникода, 8-битные или многобайтовые)  
>>> type(B), type(S)  
(<class ‘bytes’>, <class ‘str’>)  
>>> B # Выведет строку символов,   
b’spam’ # в действительности – последовательность целых чисел  
>>> S  
‘eggs’  
В действительности объект типа bytes является последовательностью корот-  
ких целых чисел, однако его содержимое выводится как строка символов, если   
это возможно:  
>>> B[0], S[0] # Операция индексирования возвращает   
(115, ‘e’) # объект типа int для bytes и объект типа str для str  
>>> B[1:], S[1:] # Операция извлечения среза возвращает   
(b’pam’, ‘ggs’) # новый объект типа bytes или str  
>>> list(B), list(S)  
([115, 112, 97, 109], [‘e’, ‘g’, ‘g’, ‘s’]) # Объект bytes в действительности   
 # содержит целые числа  
Тип bytes относится к категории неизменяемых, как и тип str (а тип bytearray,   
описываемый ниже, – нет), – вы не сможете присвоить другой объект типа str,   
bytes или целое число по смещению в объекте bytes. Кроме того, префикс b мо-  
жет применяться к любым строковым литералам:  
>>> B[0] = ‘x’ # Оба типа являются неизменяемыми  
TypeError: ‘bytes’ object does not support item assignment  
   
>>> S[0] = ‘x’  
TypeError: ‘str’ object does not support item assignment  
   
>>> B = B””” # Префикс b может предшествовать апострофам,   
... xxxx # кавычкам или тройным кавычкам  
... yyyy  
... “””  
>>> B  
b’\nxxxx\nyyyy\n’  
Как уже упоминалось выше, в Python 2.6 также может использоваться форма   
литерала b’xxx’, но она интерпретируется точно так же, как форма ‘xxx’ и соз-  
дает объект типа str, а bytes – это всего лишь синоним для str. В версии 3.0, как   
было показано выше, обе эти формы воспроизводят объекты отличного типа   
bytes. Обратите также внимание, что формы u’xxx’ и U’xxx’ литералов строк   
Юникода, имеющиеся в 2.6, были убраны в 3.0 – вместо них следует использо-  
вать форму ‘xxx’, потому что теперь все строки интерпретируются как строки   
Юникода, даже если они состоят только из символов ASCII (подробнее о лите-  
ASCII (подробнее о лите-  
 (подробнее о лите-  
ралах Юникода, содержащих не только символы ASCII, рассказывается в раз-  
ASCII, рассказывается в раз-  
, рассказывается в раз-  
деле «Кодирование строк символов не-ASCII» ниже).

Примеры использования строк в Python 3.0   
1005  
Преобразования  
В Python 2.X допускается смешивать в операциях объекты типов str и unicode   
(если строки содержат только 7-битные символы ASCII). Однако в версии 3.0   
различия между типами str и bytes считаются настолько существенными, что   
смешивание их в выражениях не допускается, и они никогда автоматически   
не преобразуются из одного в другой при передаче в функции. Функция, кото-  
рая ожидает получить аргумент типа str, обычно не принимает аргумент типа   
bytes, и наоборот.  
По этой причине в Python 3.0 требуется, чтобы вы передавали объекты того   
или другого типа или выполняли явное преобразование:  
 •  
str.encode() и bytes(S, encoding) преобразуют строку в последовательность   
простых байтов и на основе объекта типа str создают объект типа bytes.  
 •  
bytes.decode() и str(B, encoding) преобразуют последовательность простых   
байтов в строку и на основе объекта типа bytes создают объект типа str.  
Эти методы, encode и decode (а также объекты файлов, описываемые в следую-  
щем разделе), используют либо кодировку по умолчанию, исходя из настроек   
системы, либо явно указанное название кодировки. Например, в 3.0:  
>>> S = ‘eggs’  
>>> S.encode() # str в bytes: кодирует текст в последовательность байтов  
b’eggs’  
   
>>> bytes(S, encoding=’ascii’) # str в bytes, альтернативный способ  
b’eggs’  
   
>>> B = b’spam’  
>>> B.decode() # bytes в str: декодирует байты в текст  
‘spam’  
   
>>> str(B, encoding=’ascii’) # bytes в str, альтернативный способ  
‘spam’  
Здесь следует сделать два замечания. Прежде всего, кодировку по умолчанию,   
используемую в системе, можно узнать с помощью модуля sys, но аргумент en-  
coding в функции bytes не является необязательным, даже при том, что в функ-  
ции str.encode (и bytes.decode) одноименный аргумент – необязательный.  
Во-вторых, несмотря на то, что функция str не требует указывать аргумент en-  
coding, в отличие от функции bytes, его отсутствие в вызове функции str вовсе   
не означает, что будет использоваться кодировка по умолчанию. Когда функ-  
ция str вызывается без аргумента encoding, она возвращает строковую форму   
объекта bytes, а не преобразует его в объект str (обычно это не совсем то, что   
требуется!). Предположим, что объекты B и S имеют значения, какие они по-  
лучили в предыдущем сеансе:  
>>> import sys  
>>> sys.platform # Тип платформы  
‘win32’  
>>> sys.getdefaultencoding() # Кодировка по умолчанию  
‘utf-8’  
   
>>> bytes(S)  
TypeError: string argument without an encoding

1006   
Глава 36. Юникод и строки байтов   
>>> str(B) # Вызов str без аргумента encoding  
“b’spam’” # Выведет строку, а не выполнит преобразование!  
>>> len(str(B))  
7  
>>> len(str(B, encoding=’ascii’)) # С аргументом encoding преобразует   
4 # в объект типа str  
Кодирование строк Юникода  
Операции кодирования и декодирования приобретут для вас большую зна-  
чимость, когда вы начнете применять их к строкам Юникода, содержащим   
символы, не являющиеся символами ASCII. Чтобы добавить в строковый ли-  
ASCII. Чтобы добавить в строковый ли-  
. Чтобы добавить в строковый ли-  
терал символы Юникода, которые порой даже невозможно ввести с клавиату-  
ры, в языке Python поддерживается возможность указывать экранированные   
значения байтов “\xNN” в шестнадцатеричном виде и экранированные значения   
символов Юникода “�NNNN” и “�NNNNNNNN”. Экранированные значения символов   
Юникода первого вида состоят из четырех шестнадцатеричных цифр и пред-  
ставляют 2-байтовые (16-битные) коды символов, а значения второго вида со-  
стоят из восьми шестнадцатеричных цифр и представляют 4-байтовые (32-бит-  
ные) коды символов.  
Кодирование строк символов ASCII  
Рассмотрим несколько примеров, демонстрирующих основы кодирования   
строк. Как мы уже знаем, строки символов ASCII являются простейшей раз-  
ASCII являются простейшей раз-  
 являются простейшей раз-  
новидностью строк символов Юникода, которые хранятся как последователь-  
ности байтов, представляющих символы:  
C:\misc> c:\python30\python  
>>> ord(‘X’) # В кодировке по умолчанию ‘X’ имеет значение 88   
88  
>>> chr(88) # Код 88 соответствует символу ‘X’  
‘X’  
   
>>> S = ‘XYZ’ # Строка Юникода из символов ASCII  
>>> S  
‘XYZ’  
>>> len(S) # 3 символа  
3  
>>> [ord(c) for c in S] # 3 байта с целочисленными значениями  
[88, 89, 90]  
Обычный текст, состоящий только из 7-битных символов ASCII, как в данном   
примере, представляется как последовательность байтов в любых схемах коди-  
рования Юникода, о чем уже говорилось выше:  
>>> S.encode(‘ascii’) # Значения 0..127 в 1 байте (7 битов) каждое  
b’XYZ’  
>>> S.encode(‘latin-1’) # Значения 0..255 в 1 байте (8 битов) каждое  
b’XYZ’  
>>> S.encode(‘utf-8’) # Значения 0..127 в 1 байте,   
b’XYZ’ # 128..2047 – в 2, другие – в 3 или 4  
Фактически объекты типа bytes, возвращаемые данной операцией кодирова-  
ния строки символов ASCII, в действительности являются последовательно-  
ASCII, в действительности являются последовательно-  
, в действительности являются последовательно-

Кодирование строк Юникода   
1007  
стью коротких целых чисел, которые просто выводятся как символы ASCII,   
когда это возможно:  
>>> S.encode(‘latin-1’)[0]  
88  
>>> list(S.encode(‘latin-1’))  
[88, 89, 90]  
Кодирование строк символов не-ASCII  
Для представления символов, не входящих в диапазон ASCII, можно исполь-  
ASCII, можно исполь-  
, можно исполь-  
зовать шестнадцатеричные экранированные последовательности значений   
байтов и символов Юникода – шестнадцатеричные экранированные последо-  
вательности значений байтов могут представлять только значения отдельных   
байтов, а экранированные последовательности значений символов Юникода   
могут определять символы, состоящие из двух или четырех байтов. Шестнад-  
цатеричные значения 0xCD и 0xE8, например, представляют коды двух специ-  
альных символов с диакритическими знаками, не входящими в диапазон   
7-битных символов ASCII, но мы можем вставлять их в объекты str, потому что   
тип str в Python 3.0 поддерживает символы Юникода:  
>>> chr(0xc4) # 0xC4, 0xE8: символы, не входящие в диапазон ASCII  
‘Ä’  
>>> chr(0xe8)  
‘è’  
   
>>> S = ‘\xc4\xe8’ # Экранированные последовательности шестнадцатеричных  
>>> S # значений байтов  
‘Äè’  
   
>>> S = ‘Äè’ # 16-битные экранированные последовательности   
>>> S # шестнадцатеричных значений символов Юникода  
‘Äè’  
>>> len(S) # 2 символа (это не число байтов!)  
2  
Кодирование и декодирование строк символов не-ASCII  
Если теперь попробовать закодировать строки символов не-ASCII в последо-  
ASCII в последо-  
 в последо-  
вательности простых байтов, используя кодировку ASCII, мы получим сооб-  
ASCII, мы получим сооб-  
, мы получим сооб-  
щение об ошибке. Однако, если указать кодировку Latin-1, ошибки не будет   
и каждому символу в строке будет поставлен в соответствие отдельный байт.   
При использовании кодировки UTF-8 для каждого символа будет выделено по   
2 байта. Если записать такую строку в файл, в нем фактически будет сохранена   
последовательность байтов с учетом использовавшейся кодировки, как показа-  
но ниже:  
>>> S = ‘Äè’  
>>> S  
‘Äè’  
>>> len(S)  
2  
   
>>> S.encode(‘ascii’)  
UnicodeEncodeError: ‘ascii’ codec can’t encode characters in position 0-1:  
ordinal not in range(128)

1008   
Глава 36. Юникод и строки байтов   
>>> S.encode(‘latin-1’) # По одному байту на символ  
b’\xc4\xe8’  
   
>>> S.encode(‘utf-8’) # Два байта на символ  
b’\xc3\x84\xc3\xa8’  
   
>>> len(S.encode(‘latin-1’)) # 2 байта – в latin-1, 4 – в utf-8  
2  
>>> len(S.encode(‘utf-8’))  
4  
Обратите внимание, что можно пойти обратным путем – прочитать последова-  
тельность байтов из файла и декодировать их в строку символов Юникода. Од-  
нако, как будет показано ниже, если в вызове функции open указать название   
кодировки, то операции чтения автоматически будут выполнять декодирова-  
ние прочитанных данных (и помогут избежать ошибок, которые могут явить-  
ся результатом чтения неполных последовательностей символов, когда чтение   
выполняется блоками байтов):  
>>> B = b’\xc4\xe8’  
>>> B  
b’\xc4\xe8’  
>>> len(B) # 2 байта, 2 символа  
2  
>>> B.decode(‘latin-1’) # Декдировать в текст latin-1  
‘Äè’  
   
>>> B = b’\xc3\x84\xc3\xa8’  
>>> len(B) # 4 байта  
4  
>>> B.decode(‘utf-8’)  
‘Äè’  
>>> len(B.decode(‘utf-8’)) # 2 символа Юникода  
2  
Другие способы кодирования строк Юникода  
Некоторые кодировки используют еще более длинные последовательности   
байтов для представления символов. В случае необходимости вы можете ука-  
зывать 16- и 32-битные значения Юникода для символов в строках – в первом   
случае используется форма “�...” с четырьмя шестнадцатеричными цифрами,   
а во втором – форма “�...” с восемью шестнадцатеричными цифрами:  
>>> S = ‘AÄBèC’  
>>> S # A, B, C и 2 не-ASCII символа  
‘AÄBèC’  
>>> len(S) # 5 символов  
5  
   
>>> S.encode(‘latin-1’)  
b’A\xc4B\xe8C’  
>>> len(S.encode(‘latin-1’)) # 5 байтов в кодировке latin-1  
5  
   
>>> S.encode(‘utf-8’)  
b’A\xc3\x84B\xc3\xa8C’  
>>> len(S.encode(‘utf-8’)) # 7 байтов в кодировке utf-8  
7

Кодирование строк Юникода   
1009  
Интересно, что некоторые кодировки могут иметь существенные различия   
в кодах символов. Например, кодировка cp500 EBCDIC даже символы ASCII   
кодирует совсем не так, как кодировки, с которыми мы уже познакомились   
выше (поскольку интерпретатор автоматически выполняет кодирование и де-  
кодирование, нам остается только позаботиться о том, чтобы указать нужное   
имя кодировки):  
>>> S  
‘AÄBèC’  
>>> S.encode(‘cp500’) # Две другие западноевропейские кодировки  
b’\xc1c\xc2T\xc3’  
>>> S.encode(‘cp850’) # 5 байтов в каждой  
b’A\x8eB\x8aC’  
   
>>> S = ‘spam’ # В большинстве кодировок символы ASCII   
>>> S.encode(‘latin-1’) # кодируются одинаково  
b’spam’  
>>> S.encode(‘utf-8’)  
b’spam’  
>>> S.encode(‘cp500’) # Но не в кодировке cp500: IBM EBCDIC!  
b’\xa2\x97\x81\x94’  
>>> S.encode(‘cp850’)  
b’spam’  
С технической точки зрения, вы можете составлять строки Юникода по ча-  
стям, используя функцию chr вместо экранированных шестнадцатеричных   
значений, но это может оказаться весьма утомительным в случае длинных   
строк:  
>>> S = ‘A’ + chr(0xC4) + ‘B’ + chr(0xE8) + ‘C’  
>>> S  
‘AÄBèC’  
Здесь следует сделать два замечания. Во-первых, в Python 3.0 допускается   
в строках типа str кодировать специальные символы с использованием шест-  
надцатеричных экранированных последовательностей значений байтов и сим-  
волов Юникода, но в строках типа bytes могут применяться только шестнад-  
цатеричные экранированные последовательности значений байтов – экрани-  
рованные последовательности значений символов Юникода в строках типа   
bytes будут интерпретироваться буквально, а не как экранированные последо-  
вательности. Фактически строки bytes должны декодироваться в строки str,   
чтобы корректно вывести символы, не являющиеся символами ASCII:  
>>> S = ‘A\xC4B\xE8C’ # str распознает экранированные   
>>> S # значения символов Юникода  
‘AÄBèC’  
   
>>> S = ‘AÄBèC’  
>>> S  
‘AÄBèC’  
   
>>> B = b’A\xC4B\xE8C’ # bytes распознает экранированные   
>>> B # последовательности байтов, но не символов Юникода  
b’A\xc4B\xe8C’  
   
>>> B = b’AÄBèC’ # Экранированные последовательности   
>>> B # интерпретируются буквально!  
b’A\\u00C4B\\U000000E8C’

1010   
Глава 36. Юникод и строки байтов   
>>> B = b’A\xC4B\xE8C’ # В строках bytes используйте экранированные   
 # последовательности байтов  
>>> B # Выведет не-ASCII символы   
b’A\xc4B\xe8C’ # в шестнадцатеричном виде  
>>> print(B)  
b’A\xc4B\xe8C’  
>>> B.decode(‘latin-1’) # Декодировать в кодировку latin-1,   
‘AÄBèC’ # чтобы вывести как текст  
Во-вторых, при определении литералов bytes допускается использовать симво-  
лы ASCII, а для байтов со значениями выше 127 – экранированные последова-  
ASCII, а для байтов со значениями выше 127 – экранированные последова-  
, а для байтов со значениями выше 127 – экранированные последова-  
тельности шестнадцатеричных значений. С другой стороны, в литералах str   
допускается использовать любые символы, имеющиеся в исходной кодировке   
(в качестве которой, как будет рассказываться ниже, по умолчанию использу-  
ется UTF-8, если в исходном файле явно не была объявлена другая кодировка):  
>>> S = ‘AÄBèC’ # Символы из UTF-8, если кодировка не была объявлена  
>>> S  
‘AÄBèC’  
   
>>> B = AÄBèC’  
SyntaxError: bytes can only contain ASCII literal characters.  
   
>>> B = b’A\xC4B\xE8C’ # Допускаются символы ASCII или   
>>> B # экранированные последовательности  
b’A\xc4B\xe8C’  
>>> B.decode(‘latin-1’)  
‘AÄBèC’  
   
>>> S.encode() # Исходная строка закодирована в кодировке UTF-8  
b’A\xc3\x84B\xc3\xa8C’ # Если кодировка не указана, используется   
 # системная кодировка  
>>> S.encode(‘utf-8’)  
b’A\xc3\x84B\xc3\xa8C’  
   
>>> B.decode() # Простые байты не соответствуют кодировке utf-8  
UnicodeDecodeError: ‘utf8’ codec can’t decode bytes in position 1-2: ...  
Преобразования между кодировками  
До сих пор мы использовали операции кодирования и декодирования строк,   
только чтобы исследовать их структуру. В более общем случае мы можем ис-  
пользовать эти операции для преобразования строк в другие кодировки, отли-  
чающиеся от исходной кодировки по умолчанию, но при этом мы должны явно   
указывать название кодировки в операциях кодирования и декодирования:  
>>> S = ‘AÄBèC’  
>>> S  
‘AÄBèC’  
>>> S.encode() # По умолчанию используется кодировка utf-8  
b’A\xc3\x84B\xc3\xa8C’  
   
>>> T = S.encode(‘cp500’) # Преобразовать в кодировку EBCDIC  
>>> T  
b’\xc1c\xc2T\xc3’  
   
>>> U = T.decode(‘cp500’) # Преобразовать обратно в Юникод  
>>> U

Кодирование строк Юникода   
1011  
‘AÄBèC’  
   
>>> U.encode() # По умолчанию снова используется кодировка utf-8  
b’A\xc3\x84B\xc3\xa8C’  
Имейте в виду, что специальные экранированные последовательности не-  
обходимы только для представления символов, не входящих в набор ASCII,   
в литералах строк. На практике такой текст вам часто придется загружать из   
файлов. Как будет показано ниже в этой главе, объект файла в Python 3.0 (соз-  
Python 3.0 (соз-  
 3.0 (соз-  
данный с помощью встроенной функции open) автоматически декодирует тек-  
стовые строки в процессе их чтения и кодирует в процессе записи – благодаря   
этому в своих сценариях вы сможете работать со строками обычным способом,   
не нуждаясь в использовании специальных форм представления символов.  
Далее в этой главе мы также познакомимся с возможностью преобразования   
строк из одной кодировки в другую в процессе чтения и записи в файлы, ис-  
пользуя прием, близко напоминающий тот, что приводится в последнем при-  
мере, – хотя вам и потребуется явно указывать имя кодировки при открытии   
файла, тем не менее, объекты файлов автоматически будут выполнять боль-  
шую часть работы.   
Кодирование строк Юникода в Python 2.6  
Теперь, когда я познакомил вас с основными особенностями строк Юникода   
в Python 3.0, я должен заметить, что практически те же самые операции над   
строками доступны и в Python 2.6, хотя набор инструментов, используемый   
при этом, несколько отличается. В Python 2.6 имеется тип unicode, но он отли-  
чается от типа str, а кроме того, в этой версии допускается смешивать обычные   
строки и строки Юникода в выражениях, если они совместимы. Фактически   
вы можете интерпретировать тип str в версии 2.6, как тип bytes в версии 3.0,   
при выполнении операции декодирования последовательностей простых бай-  
тов в строки Юникода, при условии, что они являются допустимыми последо-  
вательностями. Ниже приводится пример сеанса работы в Python 2.6. В этой   
версии Python строки unicode отображаются в шестнадцатеричном виде, кроме   
случаев явного использования инструкции print, при этом характер отображе-  
ния не-ASCII символов может отличаться, в зависимости от используемой обо-  
лочки (большинство примеров в этом разделе выполнялось в среде IDLE):  
C:\misc> c:\python26\python  
>>> import sys  
>>> sys.version  
‘2.6 (r26:66721, Oct 2 2008, 11:35:03) [MSC v.1500 32 bit (Intel)]’  
   
>>> S = ‘A\xC4B\xE8C’ # Строка 8-битных байтов  
>>> print S # Некоторые символы не входят в набор ASCII  
AÄBèC  
   
>>> S.decode(‘latin-1’) # Декодировать байты в кодировку latin-1  
u’A\xc4B\xe8C’  
   
>>> S.decode(‘utf-8’) # Строка не в кодировке utf-8  
UnicodeDecodeError: ‘utf8’ codec can’t decode bytes in position 1-2: invalid data  
   
>>> S.decode(‘ascii’) # Имеются символы, не входящие в набор ASCII  
UnicodeDecodeError: ‘ascii’ codec can’t decode byte 0xc4 in position 1: ordinal not   
in range(128)

1012   
Глава 36. Юникод и строки байтов   
Чтобы сохранить произвольную строку с символами Юникода, требуется соз-  
дать объект типа unicode в виде литерала u’xxx’ (данная форма представления   
строковых литералов не поддерживается в Python 3.0, поскольку в этой версии   
все строки являются строками Юникода):  
>>> U = u’A\xC4B\xE8C’ # Создать строку Юникода,   
>>> U # экранированные последовательности  
u’A\xc4B\xe8C’  
>>> print U  
AÄBèC  
Создав строку Юникода, вы сможете преобразовать ее в последовательность   
простых байтов с использованием другой кодировки аналогично тому, как вы-  
полняется кодирование объектов типа str в версии 3.0:  
>>> U.encode(‘latin-1’) # Преобразовать в latin-1: 8-битные байты  
‘A\xc4B\xe8C’  
>>> U.encode(‘utf-8’) # Преобразовать в utf-8: многобайтовые символы  
‘A\xc3\x84B\xc3\xa8C’  
В Python 2.6, как и в версии 3.0, символы, не входящие в набор ASCII, могут   
быть представлены в строковых литералах в виде экранированных последова-  
тельностей. Однако экранированные последовательности вида “�...” и “�...”   
в версии 2.6 будут распознаваться только в строках типа unicode и не будут   
распознаваться в строках 8-битных символов, как и в строках типа bytes в Py-  
Py-  
thon 3.0:  
C:\misc> c:\python26\python  
>>> U = u’A\xC4B\xE8C’ # Экранированные последовательности значений байтов   
>>> U # для представления символов, не входящих   
u’A\xc4B\xe8C’ # в набор ASCII  
>>> print U  
AÄBèC  
   
>>> U = u’AÄBèC’ # Экранированные последовательности значений  
 # Юникода для представления символов,   
 # не входящих в набор ASCII  
>>> U # u’’ = 16 битов, U’’ = 32 бита  
u’A\xc4B\xe8C’  
>>> print U  
AÄBèC  
   
>>> S = ‘A\xC4B\xE8C’ # Экранированные последовательности значений   
>>> S # байтов можно использовать  
‘A\xc4B\xe8C’  
>>> print S # Но некоторые символы выводятся неправильно,   
A-BFC # если их не декодировать  
>>> print S.decode(‘latin-1’)  
AÄBèC  
   
>>> S = ‘AÄBèC’ # Экранированные последовательности значений   
>>> S # Юникода не допускаются:   
‘A\\u00C4B\\U000000E8C’ # интерпретируются буквально!  
>>> print S  
AÄBèC  
>>> len(S)  
19

Кодирование строк Юникода   
1013  
Подобно типам str и bytes в Python 3.0, типы unicode и str в 2.6 поддерживают   
практически идентичный набор операций, поэтому, если вам не требуется вы-  
полнять преобразование в другие кодировки, вы можете работать со строками   
unicode так же, как со строками str. Однако одно из важнейших отличий между   
Python 2.6 и 3.0 состоит в том, что объекты unicode и str могут смешиваться   
в выражениях, и, если строка типа str совместима с кодировкой, используе-  
мой объектом unicode, интерпретатор автоматически преобразует ее в объект   
unicode (в 3.0 не поддерживается возможность автоматического смешивания   
объектов типов str и bytes и в выражениях необходимо явно выполнять преоб-  
разования):  
>>> u’ab’ + ‘cd’ # В 2.6 допускается смешивать совместимые строки  
u’abcd’ # ‘ab’ + b’cd’ – недопустимо в 3.0  
Фактически различия в типах часто бывают достаточно тривиальными для   
программного кода в версии 2.6. Подобно обычным строкам, строки Юникода   
поддерживают операции конкатенации, индексирования, извлечения срезов,   
сопоставления с шаблонами с помощью модуля re и так далее, и они не могут   
изменяться непосредственно в памяти. Если вам когда-либо потребуется явно   
выполнять преобразования между этими двумя типами, вы можете использо-  
вать встроенные функции str и unicode:  
>>> str(u’spam’) # Преобразование строки Юникода в обычную строку  
‘spam’  
>>> unicode(‘spam’) # Преобразование обычной строки в строку Юникода  
u’spam’  
Однако такой либеральный подход к смешиванию строковых типов в выраже-  
ниях в версии 2.6 возможен, только если строки совместимы с кодировкой объ-  
ектов unicode:  
>>> S = ‘A\xC4B\xE8C’ # Смешивание несовместимых строк недопустимо  
>>> U = u’A\xC4B\xE8C’  
>>> S + U  
UnicodeDecodeError: ‘ascii’ codec can’t decode byte 0xc4 in position 1: ordinal not   
in range(128)  
   
>>> S.decode(‘latin-1’) + U # По-прежнему необходимо явно выполнять   
u’A\xc4B\xe8CA\xc4B\xe8C’ # преобразование  
   
>>> print S.decode(‘latin-1’) + U  
AÄBèCAÄBèC  
Наконец, как будет показано ниже в этой главе, функция open в версии 2.6 под-  
держивает только файлы 8-битных байтов, содержимое которых возвращается   
в виде строк str, – вам придется самим выбирать, как интерпретировать это со-  
держимое – как текст или как двоичные данные – и декодировать их по мере не-  
обходимости. Чтобы в Python 2.6 обеспечить автоматическое кодирование и де-  
Python 2.6 обеспечить автоматическое кодирование и де-  
 2.6 обеспечить автоматическое кодирование и де-  
кодирование содержимого файлов Юникода операциями записи и чтения, сле-  
дует использовать функцию codecs.open, которая описывается в руководстве по   
стандартной библиотеке Python 2.6. Эта функция предоставляет практически   
те же возможности, что и функция open в версии 3.0, и использует объекты uni-  
code для представления содержимого файлов – при чтении байтов из файла они   
декодируются в символы Юникода, а при записи строки преобразуются в по-  
следовательности байтов с учетом кодировки, указанной при открытии файла.

1014   
Глава 36. Юникод и строки байтов   
Объявление кодировки по умолчанию в файлах  
Экранированные последовательности значений Юникода отлично подходят   
для определения символов Юникода в литералах строк, но их использование   
может стать достаточно утомительным при частом употреблении для внедре-  
ния символов не-ASCII в строки. Для представления строк в исходных текстах   
сценариев Python по умолчанию использует кодировку UTF-8, однако имеется   
возможность указать любую другую кодировку, включив комментарий с на-  
званием требуемой кодировки. В сценариях для Python 2.6 и 3.0 данный ком-  
Python 2.6 и 3.0 данный ком-  
 2.6 и 3.0 данный ком-  
ментарий должен находиться в первой или во второй строке и должен иметь   
следующий вид:  
# -\*- coding: latin-1 -\*-  
Если в сценарии присутствует подобный комментарий, интерпретатор будет   
распознавать строки, представленные в указанной кодировке. Это означает,   
что вы можете редактировать файл сценария в текстовом редакторе, способ-  
ном принимать и корректно отображать национальные символы, не входящие   
в набор ASCII, а интерпретатор будет корректно декодировать их в строковые   
литералы. Например, обратите внимание, как комментарий в начале следую-  
щего файла text.py обеспечивает возможность употребления символов Latin-1   
в литералах строк:  
# -\*- coding: latin-1 -\*-  
   
# Все следующие литералы строк содержат символы latin-1.  
# Если изменить название кодировки в комментарии выше на ascii или utf-8,  
# это приведет к появлению ошибок, так как значения 0xc4 и 0xe8 в строке   
# myStr1 не являются допустимыми ни в одной из них.  
   
myStr1 = ‘aÄBèC’  
   
myStr2 = ‘AÄBèC’  
   
myStr3 = ‘A’ + chr(0xC4) + ‘B’ + chr(0xE8) + ‘C’  
   
import sys  
print(‘Default encoding:’, sys.getdefaultencoding())  
   
for aStr in myStr1, myStr2, myStr3:  
 print(‘{0}, strlen={1}, ‘.format(aStr, len(aStr)), end=’’)  
   
 bytes1 = aStr.encode() # В utf-8 по умолчанию:   
 # 2 байта на каждый символ не-ASCII  
 bytes2 = aStr.encode(‘latin-1’) # По одному байту на символ  
 #bytes3 = aStr.encode(‘ascii’) # Кодирование в ASCII приведет к ошибке:   
 # за пределами диапазона 0..127   
   
 print(‘byteslen1={0}, byteslen2={1}’.format(len(bytes1), len(bytes2)))  
Если запустить этот сценарий, он выведет следуюшее:  
C:\misc> c:\python30\python text.py  
Default encoding: utf-8  
aÄBèC, strlen=5, byteslen1=7, byteslen2=5  
AÄBèC, strlen=5, byteslen1=7, byteslen2=5  
AÄBèC, strlen=5, byteslen1=7, byteslen2=5

Использование объектов bytes в Python 3.0   
1015  
Большинство программистов вероятнее всего будут использовать стандартную   
кодировку UTF-8, поэтому я оставлю описание подробностей, касающихся   
этой и других особенностей поддержки Юникода, таких как свойства и назва-  
ния символов, за стандартным набором руководств по языку Python.  
Использование объектов bytes в Python 3.0  
В главе 7 мы изучили большое разнообразие операций, поддерживаемых ти-  
пом str в Python 3.0, – в основе своей строковый тип действует одинаково в вер-  
Python 3.0, – в основе своей строковый тип действует одинаково в вер-  
 3.0, – в основе своей строковый тип действует одинаково в вер-  
сиях 2.6 и 3.0, поэтому мы не будем вновь возвращаться к этой теме. Вместо   
этого мы поближе познакомимся с операциями, которые поддерживаются но-  
вым типом bytes Python 3.0.  
Как уже упоминалось ранее, объекты типа bytes являются последовательно-  
стями коротких целых чисел, каждое из которых имеет значение в диапазоне   
от 0 до 255, которые могут выводиться как символы ASCII. Этот тип поддер-  
ASCII. Этот тип поддер-  
. Этот тип поддер-  
живает обычные операции над последовательностями и большинство строко-  
вых методов, доступных для объектов типа str (и для объектов типа str в вер-  
сии 2.X). Однако тип bytes не поддерживает метод format и оператор % формати-  
рования, и вы не сможете смешивать и сопоставлять объекты типов bytes и str,   
не выполняя явное преобразование, – для представления тестовых данных вы   
в подавляющем большинстве случаев будете использовать объекты типа str   
и текстовые файлы, а для представления двоичных данных – объекты типа   
bytes и двоичные файлы.  
Методы  
Если вы действительно желаете увидеть, какие атрибуты имеют объекты типа   
str, которые отсутствуют в объектах типа bytes, вы всегда можете воспользо-  
ваться встроенной функцией dir. Результаты вызова этой функции сообщат   
также дополнительную информацию об операторах, поддерживаемых этими   
типами (например, методы \_\_mod\_\_ и \_\_rmod\_\_ реализуют оператор деления по   
модулю %):  
C:\misc> c:\python30\python  
   
# Атрибуты, уникальные для типа str  
   
>>> set(dir(‘abc’)) - set(dir(b’abc’))  
{‘isprintable’, ‘format’, ‘\_\_mod\_\_’, ‘encode’, ‘isidentifier’,  
‘\_formatter\_field\_name\_split’, ‘isnumeric’, ‘\_\_rmod\_\_’, ‘isdecimal’,  
‘\_formatter\_parser’, ‘maketrans’}  
   
# Атрибуты, уникальные для типа bytes  
   
>>> set(dir(b’abc’)) - set(dir(‘abc’))  
{‘decode’, ‘fromhex’}  
Как видите, типы str и bytes обладают практически идентичной функциональ-  
ностью. Атрибуты, отличающие их друг от друга, являются методами, кото-  
рые не могут применяться к объектам другого типа� например, метод decode   
преобразует последовательность простых байтов в объект типа str, а метод en-  
code преобразует строку в последовательность байтов типа bytes. Оба типа под-  
держивают множество одних и тех же методов, только методы объектов bytes

1016   
Глава 36. Юникод и строки байтов   
принимают аргументы типа bytes (напомню, что в версии 3.0 объекты строко-  
вого типа не могут смешиваться с объектами типа bytes). Кроме того, объекты   
типа bytes относятся к категории неизменяемых, как и объекты типа str в Py-  
Py-  
thon 2.6 и 3.0 (сообщения об ошибках в следующем примере были урезаны для   
краткости):  
>>> B = b’spam’ # b’...’ – литерал типа bytes  
>>> B.find(b’pa’)  
1  
   
>>> B.replace(b’pa’, b’XY’) # Методы объектов bytes принимают   
b’sXYm’ # аргументы типа bytes  
   
>>> B.split(b’pa’)  
[b’s’, b’m’]  
   
>>> B  
b’spam’  
   
>>> B[0] = ‘x’  
TypeError: ‘bytes’ object does not support item assignment  
Одно важное отличие заключается в том, что операции форматирования строк   
в версии 3.0 могут применяться только к объектам типа str и не поддержива-  
ются объектами типа bytes (подробнее о форматировании строк рассказывает-  
ся в главе 7):  
>>> b’%s’ % 99  
TypeError: unsupported operand type(s) for %: ‘bytes’ and ‘int’  
   
>>> ‘%s’ % 99  
‘99’  
   
>>> b’{0}’.format(99)  
AttributeError: ‘bytes’ object has no attribute ‘format’  
>>> ‘{0}’.format(99)  
‘99’  
Операции над последовательностями  
Кроме методов, объекты типов str и bytes в версии 3.0 поддерживают все обыч-  
ные операции над последовательностями, известные (и, возможно, полюбив-  
шиеся) вам по строкам и спискам в версии Python 2.X, – включая индексиро-  
Python 2.X, – включая индексиро-  
 2.X, – включая индексиро-  
X, – включая индексиро-  
, – включая индексиро-  
вание, извлечение срезов, конкатенацию и так далее. Обратите внимание, что   
операция индексирования в следующем примере для объекта типа bytes воз-  
вращает целое число, представляющее значение байта, – объекты bytes в дей-  
ствительности являются последовательностями 8-битных целых чисел, но для   
удобства они выводятся как строки символов ASCII, когда это возможно. Что-  
ASCII, когда это возможно. Что-  
, когда это возможно. Что-  
бы узнать, какому символу соответствует значение того или иного байта, ис-  
пользуйте встроенную функцию chr, как показано ниже:  
>>> B = b’spam’ # Последовательность коротких целых чисел  
>>> B # Выводится как последовательность символов ASCII  
b’spam’  
   
>>> B[0] # Операция индексирования возвращает целое число  
115  
>>> B[-1]

Использование объектов bytes в Python 3.0   
1017  
109  
   
>>> chr(B[0]) # Выведет символ, соответствующий целому числу  
‘s’  
>>> list(B) # Выведет целочисленные значения всех байтов   
[115, 112, 97, 109]  
   
>>> B[1:], B[:-1]  
(b’pam’, b’spa’)  
   
>>> len(B)  
4  
   
>>> B + b’lmn’  
b’spamlmn’  
>>> B \* 4  
b’spamspamspamspam’  
Другие способы создания объектов bytes  
До сих пор мы создавали объекты типа bytes с использованием синтаксиса ли-  
тералов b’...’ – однако они точно так же могут создаваться вызовом конструк-  
тора bytes с объектом типа str и названием кодировки, вызовом конструктора   
bytes с итерируемым объектом, возвращающим целые числа, или с помощью   
метода encoding объекта str с явно указанным (или подразумеваемым по умол-  
чанию) названием кодировки. Как мы уже видели, операция кодирования   
принимает объект str и возвращает последовательность байтов со значениями,   
в зависимости от указанной кодировки, – операция декодирования, напротив,   
принимает последовательность простых байтов и декодирует ее в строку – по-  
следовательность символов, возможно многобайтовых. Обе операции создают   
новые строковые объекты:  
>>> B = b’abc’  
>>> B  
b’abc’  
   
>>> B = bytes(‘abc’, ‘ascii’)  
>>> B  
b’abc’  
   
>>> ord(‘a’)  
97  
>>> B = bytes([97, 98, 99])  
>>> B  
b’abc’  
   
>>> B = ‘spam’.encode() # Или bytes()  
>>> B  
b’spam’  
>>>  
>>> S = B.decode() # Или str()  
>>> S  
‘spam’  
С более общей точки зрения, последние две операции в действительности явля-  
ются операциями преобразования между типами str и bytes – темы, которая   
была начата выше и будет продолжена в следующем разделе.

1018   
Глава 36. Юникод и строки байтов   
Смешивание строковых типов в выражениях  
Методу replace в примере, приводившемся в разделе «Методы» выше, мы пе-  
редавали два объекта типа bytes – он не принимает объекты типа str. В Py-  
Py-  
thon 2.X объекты типа str автоматически преобразуются в объекты типа uni-  
code и обратно, если это возможно (то есть, когда объекты str содержат только   
7-битные символы ASCII), однако в Python 3.0 в некоторых случаях допуска-  
ASCII), однако в Python 3.0 в некоторых случаях допуска-  
), однако в Python 3.0 в некоторых случаях допуска-  
Python 3.0 в некоторых случаях допуска-  
 3.0 в некоторых случаях допуска-  
ется использовать строковые объекты определенного типа, а все необходимые   
преобразования должны выполняться явно:  
# Функциям и методам должны передаваться объекты допустимых типов  
   
>>> B = b’spam’  
   
>>> B.replace(‘pa’, ‘XY’)  
TypeError: expected an object with the buffer interface  
   
>>> B.replace(b’pa’, b’XY’)  
b’sXYm’  
   
>>> B = B’spam’  
>>> B.replace(bytes(‘pa’), bytes(‘xy’))  
TypeError: string argument without an encoding  
   
>>> B.replace(bytes(‘pa’, ‘ascii’), bytes(‘xy’, ‘utf-8’))  
b’sxym’  
   
# При необходимости преобразования типов должны выполняться явно  
   
>>> b’ab’ + ‘cd’  
TypeError: can’t concat bytes to str  
   
>>> b’ab’.decode() + ‘cd’ # bytes в str  
‘abcd’  
   
>>> b’ab’ + ‘cd’.encode() # str в bytes  
b’abcd’  
   
>>> b’ab’ + bytes(‘cd’, ‘ascii’) # str в bytes  
b’abcd’  
Вы можете вручную создавать объекты типа bytes, представляющие упакован-  
ные двоичные данные, однако они точно так же могут создаваться автоматиче-  
ски, в процессе чтения из файлов, открытых в двоичном режиме, как будет по-  
казано далее в этой главе. Однако сначала мы должны поближе познакомиться   
с типом bytes и родственным ему изменяемым типом.  
Использование объектов bytearray в 3.0 (и 2.6)   
До сих пор мы рассматривали типы str и bytes, поскольку они соответствуют   
типам unicode и str в Python 2. Однако в Python 3.0 имеется третий строковый   
тип – bytearray, представляющий изменяемые последовательности целых чи-  
сел со значениями в диапазоне от 0 до 255. По сути это изменяемая версия типа   
bytes. Он поддерживает те же самые методы строк и операции над последова-  
тельностями, что и тип bytes, а также множество операций, изменяющих объ-  
екты в памяти, которые поддерживаются списками. Кроме того, тип bytearray

Использование объектов bytearray в 3.0 (и 2.6)   
1019  
доступен также в Python 2.6 как результат переноса из версии 3.0, но он не так   
строго разграничивает текстовые и двоичные данные, как в версии 3.0.   
Давайте совершим короткий ознакомительный тур. Объекты типа bytearray   
могут создаваться вызовом встроенной функции bytearray. В Python 2.6 для   
инициализации могут использоваться любые строки:   
# Создание в 2.6: изменяемая последовательность коротких (0..255) целых чисел  
   
>>> S = ‘spam’  
>>> C = bytearray(S) # Результат переноса из 3.0 в 2.6  
>>> C # b’..’ == ‘..’ в 2.6 (str)  
bytearray(b’spam’)  
В Python 3.0 требуется использовать строку байтов или указывать название ко-  
Python 3.0 требуется использовать строку байтов или указывать название ко-  
 3.0 требуется использовать строку байтов или указывать название ко-  
дировки, потому что в этой версии не допускается смешивать текстовые строки   
и строки байтов, даже при том, что строки байтов могут являться отражением   
строк Юникода:  
# Создание в 3.0: текст и двоичные данные не допускается смешивать  
   
>>> S = ‘spam’  
>>> C = bytearray(S)  
TypeError: string argument without an encoding  
   
>>> C = bytearray(S, ‘latin1’) # Определенный тип содержимого в 3.0  
>>> C  
bytearray(b’spam’)  
   
>>> B = b’spam’ # b’..’ != ‘..’ в 3.0 (bytes/str)  
>>> C = bytearray(B)  
>>> C  
bytearray(b’spam’)  
В результате мы получаем объекты bytearray, которые являются последова-  
тельностями коротких целых чисел, как bytes, и изменяемыми, как списки.   
В операциях присваивания по индексу требуется указывать целые числа, а не   
строки (все следующие ниже фрагменты являются продолжением данного се-  
анса в Python 3.0, если явно не оговаривается иное, – смотрите примечания   
к использованию в версии 2.6 в комментариях):  
# Изменяемый, присваиваться должны целые числа, а не строки  
   
>>> C[0]  
115  
   
>>> C[0] = ‘x’ # Эта и следующая операция работают в 2.6  
TypeError: an integer is required  
   
>>> C[0] = b’x’  
TypeError: an integer is required  
   
>>> C[0] = ord(‘x’)  
>>> C  
bytearray(b’xpam’)  
   
>>> C[1] = b’Y’[0]  
>>> C  
bytearray(b’xYam’)

1020   
Глава 36. Юникод и строки байтов   
Для обработки объектов типа bytearray допускается использовать операции,   
которые обычно применяются к строкам и спискам, поскольку они являются   
изменяемыми строками байтов. Помимо обычных методов тип bytearray реали-  
зует также методы \_\_iadd\_\_ и \_\_setitem\_\_ поддержки оператора += конкатена-  
ции в памяти и присваивания по индексу соответственно:  
# Методы, свойственные типам str и bytes, а также методы, свойственные спискам  
   
>>> set(dir(b’abc’)) - set(dir(bytearray(b’abc’)))  
{‘\_\_getnewargs\_\_’}  
   
>>> set(dir(bytearray(b’abc’))) - set(dir(b’abc’))  
{‘insert’, ‘\_\_alloc\_\_’, ‘reverse’, ‘extend’, ‘\_\_delitem\_\_’, ‘pop’, ‘\_\_setitem\_\_’,   
‘\_\_iadd\_\_’, ‘remove’, ‘append’, ‘\_\_imul\_\_’}  
Вы можете изменять объекты типа bytearray непосредственно в памяти с по-  
мощью операции присваивания по индексу, как только что было показано,   
и с помощью методов, похожих на методы списков, как показано ниже (чтобы   
в версии 2.6 изменить текст непосредственно в памяти, вам пришлось бы пре-  
образовать его в список и обратно с помощью list(str) и ‘’.join(list)):  
# Методы изменяемых объектов  
   
>>> C  
bytearray(b’xYam’)  
   
>>> C.append(b’LMN’) # В 2.6 требуется строка с длиной 1  
TypeError: an integer is required  
   
>>> C.append(ord(‘L’))  
>>> C  
bytearray(b’xYamL’)  
   
>>> C.extend(b’MNO’)  
>>> C  
bytearray(b’xYamLMNO’)  
К объектам типа bytearray могут применяться все обычные операции над по-  
следовательностями и строковые методы, как и можно было бы предполагать   
(обратите внимание, что как и в случае с объектами типа bytes, операторы и ме-  
тоды ожидают получить аргументы типа bytes, а не str):  
# Операции над последовательностями и строковые методы  
   
>>> C + b’!#’  
bytearray(b’xYamLMNO!#’)  
   
>>> C[0]  
120  
   
>>> C[1:]  
bytearray(b’YamLMNO’)  
   
>>> len(C)  
8  
   
>>> C  
bytearray(b’xYamLMNO’)  
   
>>> C.replace(‘xY’, ‘sp’) # Будет работать в 2.6

Использование текстовых и двоичных файлов   
1021  
TypeError: Type str doesn’t support the buffer API  
   
>>> C.replace(b’xY’, b’sp’)  
bytearray(b’spamLMNO’)  
   
>>> C  
bytearray(b’xYamLMNO’)  
   
>>> C \* 4  
bytearray(b’xYamLMNOxYamLMNOxYamLMNOxYamLMNO’)  
Наконец, следующие примеры демонстрируют, что объекты типов bytes и byte-  
array являются последовательностями целых чисел, а объекты типа str – по-  
следовательностями символов:  
# Двоичные и текстовые данные  
   
>>> B # В 2.6 B и S – это объекты одного типа  
b’spam’  
>>> list(B)  
[115, 112, 97, 109]  
   
>>> C  
bytearray(b’xYamLMNO’)  
>>> list(C)  
[120, 89, 97, 109, 76, 77, 78, 79]  
   
>>> S  
‘spam’  
>>> list(S)  
[‘s’, ‘p’, ‘a’, ‘m’]  
Несмотря на то, что все три строковых типа в Python 3.0 могут содержать зна-  
Python 3.0 могут содержать зна-  
 3.0 могут содержать зна-  
чения символов и поддерживают почти те же самые операции, тем не менее, вы   
всегда должны:  
 •  
Для представления текстовых данных использовать тип str.  
 •  
Для представления двоичных данных использовать тип bytes.  
 •  
Для представления двоичных данных с возможностью непосредственного   
изменения использовать тип bytearray.  
Другие похожие инструменты, такие как файлы, которые рассматриваются   
в следующем разделе, часто делают этот выбор типа объектов за вас.  
Использование текстовых и двоичных файлов  
В этом разделе мы подробно исследуем воздействие строковой модели, реали-  
зованной в Python 3.0, на основные операции над файлами, представленные   
ранее в этой книге. Как уже упоминалось выше, режим открытия файла имеет   
важное значение – он определяет типы объектов, которые будут использовать-  
ся в ваших сценариях для представления содержимого файлов. Текстовый ре-  
жим подразумевает использование объектов типа str, а двоичный режим – объ-  
ектов типа bytes:  
 •  
Содержимое файлов, открытых в текстовом  режиме, интерпретируется   
как текст, состоящий из символов Юникода. При этом используется либо   
кодировка по умолчанию в соответствии с настройками системы, либо

1022   
Глава 36. Юникод и строки байтов   
кодировка, которая была указана явно. Передавая название кодировки   
в функцию open, вы можете принудительно определить различные виды   
преобразований файлов Юникода. Кроме того, при работе с файлами, от-  
крытыми в текстовом режиме, автоматически выполняется преобразование   
символов конца строки: по умолчанию все возможные формы обозначения   
конца строки преобразуются в единственный символ ‘\n’ независимо от   
платформы, на которой выполняется сценарий. Как описывалось ранее,   
при работе с текстовыми файлами дополнительно автоматически выполня-  
ется чтение и запись маркера порядка следования байтов (Byte Order Mark,   
BOM), который сохраняется в начале файла при использовании некоторых   
схем кодирования Юникода.  
 •  
При работе с файлами, открытыми в двоичном режиме, их содержимое воз-  
вращается в виде последовательности целых чисел, представляющих зна-  
чения байтов, без предварительного кодирования или декодирования и без   
преобразования символов конца строки.  
Второй аргумент функции open определяет режим обработки файла – тексто-  
вый или двоичный, как и в Python 2.X. Добавление символа «b» в эту стро-  
ку определяет двоичный режим (например, “rb” обозначает двоичный режим   
только для чтения). По умолчанию используется режим “rt” – он обозначает то   
же самое, что и режим “r”, то есть режим чтения из текстового файла (как и в   
версии 2.X).  
Однако в Python 3.0 аргумент режима в вызове функции open также определя-  
ет тип объектов, которые будут представлять содержимое файла, независимо   
от платформы, на которой выполняется сценарий, – при работе с текстовыми   
файлами операциями чтения и записи будут возвращаться и приниматься объ-  
екты типа str, а при работе с двоичными файлами операциями чтения и записи   
будут возвращаться и приниматься объекты типа bytes (операции записи мо-  
гут также принимать объекты типа bytearray).  
Основы текстовых файлов  
Начнем с демонстрации основных операций ввода-вывода. Пока вы работаете   
с простыми текстовыми файлами (например, с файлами, содержащими текст   
ASCII) и не беспокоитесь о тонкостях кодирования строк с использованием ко-  
) и не беспокоитесь о тонкостях кодирования строк с использованием ко-  
дировок, отличных от системной кодировки по умолчанию, файлы в Python 3.0   
выглядят практически так же, как и в Python 2.X (то есть вы просто работае-  
X (то есть вы просто работае-  
 (то есть вы просто работае-  
те с обычными строками). Например, ниже выполняется запись одной строки   
текста в файл и чтение ее из файла, при этом все выглядит точно так же, как   
в версии 2.6 (обратите внимание, что в версии 3.0 слово file больше не является   
зарезервированным именем типа, поэтому мы вполне можем использовать его   
в качестве имени переменной):  
C:\misc> c:\python30\python  
   
# В простейшем случае текстовые файлы (и строки) действуют так же, как и в 2.X  
   
>>> file = open(‘temp’, ‘w’)  
>>> size = file.write(‘abc\n’) # Возвратит количество записанных байтов  
>>> file.close() # Закрыть файл, чтобы вытолкнуть выходной буфер  
   
>>> file = open(‘temp’) # Режим по умолчанию – “r” (“rt”): чтение текста  
>>> text = file.read()

Использование текстовых и двоичных файлов   
1023  
>>> text  
‘abc\n’  
>>> print(text)  
abc  
Текстовый и двоичный режимы в Python 3.0  
В Python 2.6 не делалось больших различий между текстовыми и двоичными   
файлами – в обоих случаях содержимое возвращалось и принималось в виде   
строк типа str. Единственное отличие состояло в том, что при работе с тексто-  
выми файлами на платформе �indows автоматически выполняется преобразо-  
�indows автоматически выполняется преобразо-  
 автоматически выполняется преобразо-  
вание символа \n конца строки в последовательность \r\n и обратно, тогда как   
при работе с двоичными файлами это преобразование не выполняется (в сле-  
дующем примере я объединил операции для краткости):  
C:\misc> c:\python26\python  
>>> open(‘temp’, ‘w’).write(‘abd\n’) # Запись в текстовом режиме: добавит \r  
>>> open(‘temp’, ‘r’).read() # Чтение в текстовом режиме: отбросит \r  
‘abd\n’  
>>> open(‘temp’, ‘rb’).read() # Чтение в двоичном режиме:   
‘abd\r\n’ # преобразования не выполняются  
   
>>> open(‘temp’, ‘wb’).write(‘abc\n’) # Запись в двоичном режиме  
>>> open(‘temp’, ‘r’).read() # \n не преобразуется в \r\n  
‘abc\n’  
>>> open(‘temp’, ‘rb’).read()  
‘abc\n’  
В Python 3.0 дело обстоит несколько сложнее из-за различий между типом str,   
используемым для представления текстовых данных, и типом bytes, исполь-  
зуемым для представления двоичных данных. Для демонстрации рассмотрим   
операцию записи в текстовый файл и операции чтения из этого же файла в обо-  
их режимах в Python 3.0. Обратите внимание, что операции записи мы долж-  
ны передать объект типа str, а операции чтения возвращают объект типа str   
или bytes, в зависимости от режима, в котором был открыт файл:  
C:\misc> c:\python30\python  
   
# Запись и чтение текстового файла  
   
>>> open(‘temp’, ‘w’).write(‘abc\n’) # Текстовый режим записи,  
4 # передается объект str  
   
>>> open(‘temp’, ‘r’).read() # Текстовый режим чтения,   
‘abc\n’ # возвращает объект str  
   
>>> open(‘temp’, ‘rb’).read() # Двоичный режим чтения,   
b’abc\r\n’ # возвращает объект bytes  
Обратите внимание, что на платформе �indows при записи в текстовый файл   
символ \n конца строки преобразуется в последовательность \r\n – при чтении   
в текстовом режиме последовательность \r\n преобразуется обратно в \n. Одна-  
ко в двоичном режиме такие преобразования не выполняются. То же проис-  
ходит и под управлением Python 2.6, причем именно такое поведение мы ожи-  
Python 2.6, причем именно такое поведение мы ожи-  
 2.6, причем именно такое поведение мы ожи-  
даем при работе с двоичными файлами (никаких дополнительных преобразо-  
ваний не должно выполняться). Впрочем, в Python 3.0 мы можем управлять   
этим поведением с помощью третьего аргумента функции open.

1024   
Глава 36. Юникод и строки байтов   
Теперь проделаем то же самое, но уже с двоичным файлом. На этот раз мы пере-  
даем операции записи объект типа bytes, а обратно получаем объект типа str   
или bytes, в зависимости от режима, в котором был открыт файл:   
# Запись и чтение двоичного файла  
   
>>> open(‘temp’, ‘wb’).write(b’abc\n’) # Двоичный режим записи,   
4 # передается объект bytes  
   
>>> open(‘temp’, ‘r’).read() # Текстовый режим чтения,   
‘abc\n’ # возвращается объект str  
   
>>> open(‘temp’, ‘rb’).read() # Двоичный режим чтения,   
b’abc\n’ # возвращается объект bytes  
Обратите внимание, что в двоичном режиме записи символ \n конца строки не   
преобразуется в последовательность \r\n – это именно то, что требуется при ра-  
боте с двоичными данными. Требования к типам и поведение файла остаются   
теми же самыми, даже если данные, записанные в файл, по своей природе яв-  
ляются двоичными. В следующем примере литерал “\x00” представляет двоич-  
ный байт с нулевым значением и не является печатным символом:  
# Запись и чтение двоичных данных  
   
>>> open(‘temp’, ‘wb’).write(b’a\x00c’) # Передается объект типа bytes  
3  
   
>>> open(‘temp’, ‘r’).read() # Возвращает объект типа str  
‘a\x00c’  
   
>>> open(‘temp’, ‘rb’).read() # Возвращает объект типа bytes  
b’a\x00c’  
При работе с файлами, открытыми в двоичном режиме, их содержимое воз-  
вращается в виде объекта bytes, но операции записи могут принимать объек-  
ты типа bytes или bytearray – это вполне естественно, потому что тип bytearray   
в действительности является всего лишь изменяемой версией типа bytes. Фак-  
тически большинство функций и методов в Python 3.0, которые принимают   
объекты типа bytes, также могут принимать объекты типа bytearray:  
# Также допускается передавать объекты bytearrays   
   
>>> BA = bytearray(b’\x01\x02\x03’)  
   
>>> open(‘temp’, ‘wb’).write(BA)  
3  
   
>>> open(‘temp’, ‘r’).read()  
‘\x01\x02\x03’  
   
>>> open(‘temp’, ‘rb’).read()  
b’\x01\x02\x03’  
Несоответствие типа и содержимого  
Обратите внимание, что вам не удастся избежать неприятностей в случае нару-  
шения правил использования типов str/bytes при работе с файлами. Как видно   
в следующем примере, мы получаем сообщение об ошибке (сокращено здесь)

Использование текстовых и двоичных файлов   
1025  
при попытке записать объект bytes в текстовый файл или объект str – в двоич-  
ный файл:   
# Типы не преобразуются автоматически при работе с файлами  
   
>>> open(‘temp’, ‘w’).write(‘abc\n’) # Текстовый режим создает и требует   
4 # использования объектов типа str  
>>> open(‘temp’, ‘w’).write(b’abc\n’)  
TypeError: can’t write bytes to text stream  
   
>>> open(‘temp’, ‘wb’).write(b’abc\n’) # Двоичный режим создает и требует   
4 # использования объектов типа bytes  
>>> open(‘temp’, ‘wb’).write(‘abc\n’)  
TypeError: can’t write str to binary stream  
В этом есть определенный смысл: в двоичном режиме текст не имеет смысла,   
пока не будет закодирован. Хотя нередко имеется возможность выполнить пре-  
образование между типами, закодировав объект типа str или декодировав объ-  
ект типа bytes, тем не менее, как описывалось выше в этой главе, вы будете ис-  
пользовать либо тип str, для представления текстовых данных, либо тип bytes,   
для представления двоичных данных. Так как типы str и bytes в значительной   
степени поддерживают похожие операции, выбор не будет для вас серьезной   
дилеммой в большинстве программ (обращайтесь к последнему разделу этой   
главы с описанием инструментов, предназначенных для работы со строками,   
где приводятся некоторые примеры).  
Кроме ограничений, связанных с типами, содержимое файлов также может   
иметь значение в версии 3.0. При записи в текстовый файл содержимое должно   
передаваться в виде объектов типа str, а не bytes – в Python 3.0 нет никакого   
способа, позволяющего записать двоичные данные в файл, открытый в тексто-  
вом режиме. В зависимости от правил кодирования байты со значениями вне   
диапазона набора символов по умолчанию иногда могут встраиваться в обыч-  
ные строки, и они всегда могут быть записаны в файл, открытый в двоичном   
режиме. Однако из-за того, что при работе с текстовыми файлами, открытыми   
для чтения, в версии 3.0 автоматически выполняется декодирование содержи-  
мого в соответствии с указанной кодировкой Юникода, в текстовом режиме от-  
сутствует возможность прочитать двоичные данные:  
# Двоичные данные невозможно прочитать в текстовом режиме  
   
>>> chr(0xFF) # FF – допустимый символ, FE – нет  
‘я’  
>>> chr(0xFE)  
UnicodeEncodeError: ‘charmap’ codec can’t encode character ‘\xfe’ in position 1...  
   
>>> open(‘temp’, ‘w’).write(b’\xFF\xFE\xFD’) # Двоичные данные нельзя записать   
TypeError: can’t write bytes to text stream # в текстовый файл!  
   
>>> open(‘temp’, ‘w’).write(‘\xFF\xFE\xFD’) # Можно записать, если байты   
3 # включены в состав str  
>>> open(‘temp’, ‘wb’).write(b’\xFF\xFE\xFD’) # Запись также можно выполнить   
3 # в двоичном режиме  
   
>>> open(‘temp’, ‘rb’).read() # Данные всегда можно прочитать   
b’\xff\xfe\xfd’ # в двоичном режиме

1026   
Глава 36. Юникод и строки байтов   
>>> open(‘temp’, ‘r’).read() # В текстовом режиме невозможно прочитать   
 # данные, которые не могут быть декодированы!  
UnicodeEncodeError: ‘charmap’ codec can’t encode characters in position 2-3: ...  
Последняя ошибка обусловлена тем, что все текстовые файлы в Python 3.0   
в действительности интерпретируются как текстовые файлы Юникода, как   
описывается в следующем разделе.  
Использование файлов Юникода  
До сих пор мы выполняли операции чтения и записи над простыми текстовы-  
ми и двоичными файлами, но как быть, если возникнет необходимость рабо-  
тать с файлами Юникода? Как оказывается, читать текст с символами Юни-  
кода из файлов и записывать его в файлы ничуть не сложнее – благодаря тому,   
что функция open в Python 3.0 принимает название кодировки для текстовых   
файлов, а операции записи и чтения автоматически выполняют кодирование   
и декодирование данных. Это позволяет читать текст Юникода, записанный   
с применением кодировок, отличающихся от кодировки, используемой в си-  
стеме по умолчанию, и записывать его в других кодировках.  
Чтение и запись Юникода в Python 3.0  
Фактически мы можем преобразовывать строки в различные кодировки либо   
вручную, с применением методов, либо автоматически, с применением файло-  
вых операций ввода-вывода. Для демонстрации мы будем использовать в этом   
разделе следующую строку Юникода:  
C:\misc> c:\python30\python  
>>> S = ‘A\xc4B\xe8C’ # Строка из 5 символов, включает символы,   
>>> S # не входящие в набор ASCII  
‘AÄBèC’  
>>> len(S)  
5  
Кодирование вручную  
Как мы уже знаем, у нас всегда имеется возможность закодировать строку   
в последовательность байтов в соответствии с указанной кодировкой:  
# Кодирование вручную с помощью методов  
   
>>> L = S.encode(‘latin-1’) # 5 байтов после кодирования в latin-1  
>>> L  
b’A\xc4B\xe8C’  
>>> len(L)  
5  
   
>>> U = S.encode(‘utf-8’) # 7 байтов после кодирования в utf-8  
>>> U  
b’A\xc3\x84B\xc3\xa8C’  
>>> len(U)  
7

Использование файлов Юникода   
1027  
Кодирование при записи в файл  
Теперь попробуем записать нашу строку в текстовый файл в определенной ко-  
дировке, указав ее название в вызове функции open, – мы могли бы сначала   
выполнить кодирование вручную, а затем записать результат в двоичном ре-  
жиме, однако в этом нет никакой необходимости:  
# Автоматическое кодирование при записи в файл  
   
>>> open(‘latindata’, ‘w’, encoding=’latin-1’).write(S) # Запись в latin-1  
5  
>>> open(‘utf8data’, ‘w’, encoding=’utf-8’).write(S) # Запись в utf-8  
5  
   
>>> open(‘latindata’, ‘rb’).read() # Прочитать двоичные данные  
b’A\xc4B\xe8C’  
   
>>> open(‘utf8data’, ‘rb’).read() # Содержимое файлов отличается  
b’A\xc3\x84B\xc3\xa8C’  
Декодирование при чтении из файла  
Точно так же при чтении произвольных данных Юникода мы просто передаем   
название кодировки в вызов функции open, после чего операции чтения авто-  
матически будут декодировать последовательности двоичных байтов в строки.   
Мы могли бы прочитать двоичные данные, а затем декодировать их вручную,   
хотя в этом случае могут возникнуть сложности при чтении данных блоками   
определенного размера (есть риск прочитать неполный символ), – но в этом нет   
никакой необходимости:   
# Автоматическое декодирование при чтении из файла  
   
>>> open(‘latindata’, ‘r’, encoding=’latin-1’).read() # Декодирование   
‘AÄBèC’ # выполняется при чтении  
>>> open(‘utf8data’, ‘r’, encoding=’utf-8’).read() # в соответствии   
‘AÄBèC’ # с названием кодировки  
   
>>> X = open(‘latindata’, ‘rb’).read() # Декодирование вручную:  
>>> X.decode(‘latin-1’) # не требуется  
‘AÄBèC’  
>>> X = open(‘utf8data’, ‘rb’).read()  
>>> X.decode() # UTF-8 – кодировка по умолчанию  
‘AÄBèC’  
Ошибки декодирования  
Наконец, имейте в виду, что такое поведение файлов в 3.0 накладывает ограни-  
чения на содержимое, которое можно загрузить как текст. Как уже говорилось   
в предыдущем разделе, в версии 3.0 интерпретатор должен иметь возможность   
декодировать данные из текстового файла в строку типа str в соответствии   
с кодировкой по умолчанию или указанной явно в вызове функции open. Если,   
к примеру, попытаться открыть настоящие двоичные данные в текстовом ре-  
жиме, в версии 3.0 вы, скорее всего, столкнетесь с ошибкой, даже если будете   
использовать объекты корректных типов:

1028   
Глава 36. Юникод и строки байтов   
>>> file = open(‘python.exe’, ‘r’)  
>>> text = file.read()  
UnicodeDecodeError: ‘charmap’ codec can’t decode byte 0x90 in position 2: ...  
   
>>> file = open(‘python.exe’, ‘rb’)  
>>> data = file.read()  
>>> data[:20]  
b’MZ\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00\xb8\x00\x00\x00’  
Первый из этих примеров может и не привести к появлению ошибки в Py-  
Py-  
thon 2.X (при работе с обычными файлами текст не декодируется), даже при   
том, что она должна была бы произойти: операция чтения из файла может вер-  
нуть поврежденные данные из-за автоматического преобразования символов   
конца строки, которое выполняется в текстовом режиме (в процессе чтения   
любые последовательности байтов \r\n будут преобразованы в �indows в сим-  
�indows в сим-  
 в сим-  
волы \n). Чтобы в версии 2.6 содержимое файла интерпретировалось как текст   
Юникода, вместо встроенной функции open необходимо использовать специ-  
альный инструмент, как будет показано чуть ниже. Но перед этим обратимся   
к более взрывоопасной теме....  
Обработка маркера BOM в Python 3.0  
Как описывалось выше в этой главе, некоторые кодировки подразумевают со-  
хранение специальной последовательности маркера порядка следования бай-  
тов (byte order marker, BOM) в начале файлов, определяющей прямой или об-  
byte order marker, BOM) в начале файлов, определяющей прямой или об-  
 order marker, BOM) в начале файлов, определяющей прямой или об-  
order marker, BOM) в начале файлов, определяющей прямой или об-  
 marker, BOM) в начале файлов, определяющей прямой или об-  
marker, BOM) в начале файлов, определяющей прямой или об-  
, BOM) в начале файлов, определяющей прямой или об-  
BOM) в начале файлов, определяющей прямой или об-  
) в начале файлов, определяющей прямой или об-  
ратный порядок следования байтов или объявляющей тип кодировки. В лю-  
бом случае интерпретатор пропускает этот маркер при вводе из файла и запи-  
сывает его при выводе в файл, если его наличие подразумевается используемой   
кодировкой, но иногда мы должны явно указывать название кодировки, чтобы   
явно указать порядок следования байтов.  
Например, при сохранении файла в программе Блокнот (Notepad) в системе   
�indows, можно указать тип кодировки в раскрывающемся списке – простой   
текст ASCII, UTF-8 или UTF-16 с прямым или обратным порядком следования   
байтов. Если, например, в Блокноте сохранить однострочный текстовый файл   
spam.txt, выбрав кодировку «ANSI», он будет сохранен как простой текстовый   
файл ASCII без маркера BOM. Если затем прочитать этот файл в двоичном ре-  
ASCII без маркера BOM. Если затем прочитать этот файл в двоичном ре-  
 без маркера BOM. Если затем прочитать этот файл в двоичном ре-  
BOM. Если затем прочитать этот файл в двоичном ре-  
. Если затем прочитать этот файл в двоичном ре-  
жиме из программы на языке Python, можно будет увидеть фактические дво-  
Python, можно будет увидеть фактические дво-  
, можно будет увидеть фактические дво-  
ичные данные, сохраненные в файле. При чтении этого файла в текстовом ре-  
жиме интерпретатор автоматически выполнит преобразование символов кон-  
ца строки – мы сможем декодировать его явно, как текст в кодировке UTF-8,   
потому что набор символов ASCII является подмножеством набора символов   
UTF-8 (в Python 3.0 UTF-8 является кодировкой по умолчанию):  
c:\misc> C:\Python30\python # Файл был сохранен в Блокноте  
>>> import sys  
>>> sys.getdefaultencoding()  
‘utf-8’  
>>> open(‘spam.txt’, ‘rb’).read() # Текстовый файл ASCII (UTF-8)  
b’spam\r\nSPAM\r\n’  
>>> open(‘spam.txt’, ‘r’).read() # В текстовом режиме выполняется   
‘spam\nSPAM\n’ # преобразование символов конца строки  
>>> open(‘spam.txt’, ‘r’, encoding=’utf-8’).read()  
‘spam\nSPAM\n’

Использование файлов Юникода   
1029  
Если сохранить этот файл в Блокноте, выбрав кодировку «UTF-8», в начало   
файла будет добавлена трехбайтовая последовательность маркера BOM для ко-  
BOM для ко-  
 для ко-  
дировки UTF-8, и тогда при чтении нам потребуется указать более специфиче-  
UTF-8, и тогда при чтении нам потребуется указать более специфиче-  
-8, и тогда при чтении нам потребуется указать более специфиче-  
ское название кодировки («utf-8-sig»), чтобы вынудить интерпретатор пропу-  
utf-8-sig»), чтобы вынудить интерпретатор пропу-  
-8-sig»), чтобы вынудить интерпретатор пропу-  
sig»), чтобы вынудить интерпретатор пропу-  
»), чтобы вынудить интерпретатор пропу-  
стить маркер при чтении содержимого файла:  
>>> open(‘spam.txt’, ‘rb’).read() # UTF-8 с 3-байтовым маркером BOM  
b’\xef\xbb\xbfspam\r\nSPAM\r\n’  
>>> open(‘spam.txt’, ‘r’).read()  
‘п»їspam\nSPAM\n’  
>>> open(‘spam.txt’, ‘r’, encoding=’utf-8’).read()  
‘﻿spam\nSPAM\n’  
>>> open(‘spam.txt’, ‘r’, encoding=’utf-8-sig’).read()  
‘spam\nSPAM\n’  
Если сохранить этот файл в Блокноте, выбрав кодировку «Юникод big endi-  
big endi-  
 endi-  
endi-  
an» («Unicode big endian»), данные будут записаны в формате UTF-16 с двух-  
» («Unicode big endian»), данные будут записаны в формате UTF-16 с двух-  
Unicode big endian»), данные будут записаны в формате UTF-16 с двух-  
 big endian»), данные будут записаны в формате UTF-16 с двух-  
big endian»), данные будут записаны в формате UTF-16 с двух-  
 endian»), данные будут записаны в формате UTF-16 с двух-  
endian»), данные будут записаны в формате UTF-16 с двух-  
»), данные будут записаны в формате UTF-16 с двух-  
UTF-16 с двух-  
-16 с двух-  
байтовым маркером BOM – при использовании названия кодировки «utf-16»   
в Python интерпретатор автоматически пропустит маркер BOM, потому что его   
наличие подразумевается этой кодировкой (все файлы с кодировкой UTF-16   
имеют маркер BOM). Однако, если указать кодировку «utf-16-be», интерпрета-  
BOM). Однако, если указать кодировку «utf-16-be», интерпрета-  
). Однако, если указать кодировку «utf-16-be», интерпрета-  
utf-16-be», интерпрета-  
-16-be», интерпрета-  
be», интерпрета-  
», интерпрета-  
тор будет обрабатывать файл как текст в формате UTF-16 с прямым (big-endian)   
порядком следования байтов, но не будет пропускать маркер BOM:  
>>> open(‘spam.txt’, ‘rb’).read()  
b’\xfe\xff\x00s\x00p\x00a\x00m\x00\r\x00\n\x00S\x00P\x00A\x00M\x00\r\x00\n’  
>>> open(‘spam.txt’, ‘r’).read()  
UnicodeEncodeError: ‘charmap’ codec can’t encode character ‘\xfe’ in position 1:...  
>>> open(‘spam.txt’, ‘r’, encoding=’utf-16’).read()  
‘spam\nSPAM\n’  
>>> open(‘spam.txt’, ‘r’, encoding=’utf-16-be’).read()  
‘﻿spam\nSPAM\n’  
То же верно и для операции записи. Чтобы из программы на языке Python за-  
Python за-  
 за-  
писать в начало файла Юникода маркер BOM кодировки UTF-8, мы должны   
явно указать более специфическую кодировку «utf-8-sig», потому что при ис-  
utf-8-sig», потому что при ис-  
-8-sig», потому что при ис-  
sig», потому что при ис-  
», потому что при ис-  
пользовании имени кодировки «utf-8» запись маркера BOM не выполняется:  
>>> open(‘temp.txt’, ‘w’, encoding=’utf-8’).write(‘spam\nSPAM\n’)  
10  
>>> open(‘temp.txt’, ‘rb’).read() # Нет маркера BOM  
b’spam\r\nSPAM\r\n’  
   
>>> open(‘temp.txt’, ‘w’, encoding=’utf-8-sig’).write(‘spam\nSPAM\n’)  
10  
>>> open(‘temp.txt’, ‘rb’).read() # Маркер BOM был записан  
b’\xef\xbb\xbfspam\r\nSPAM\r\n’  
   
>>> open(‘temp.txt’, ‘r’).read()  
‘п»їspam\nSPAM\n’  
>>> open(‘temp.txt’, ‘r’, encoding=’utf-8’).read() # Прочитает BOM  
‘﻿spam\nSPAM\n’  
>>> open(‘temp.txt’, ‘r’, encoding=’utf-8-sig’).read() # Пропустит BOM  
‘spam\nSPAM\n’  
Обратите внимание: несмотря на то, при использовании кодировки «utf-8» за-  
utf-8» за-  
-8» за-  
пись маркера BOM не производится, тем не менее мы сможем прочитать содер-  
BOM не производится, тем не менее мы сможем прочитать содер-  
 не производится, тем не менее мы сможем прочитать содер-

1030   
Глава 36. Юникод и строки байтов   
жимое этого файла при использовании любой из кодировок, «utf-8» и «utf-8-  
sig», – используйте последнюю при чтении файла, если вы не уверены в том,   
присутствует ли маркер BOM в файле (и не читайте этот абзац вслух при про-  
BOM в файле (и не читайте этот абзац вслух при про-  
 в файле (и не читайте этот абзац вслух при про-  
хождении досмотра в аэропорту!):  
>>> open(‘temp.txt’, ‘w’).write(‘spam\nSPAM\n’)  
10  
>>> open(‘temp.txt’, ‘rb’).read() # Данные без маркера BOM  
b’spam\r\nSPAM\r\n’  
>>> open(‘temp.txt’, ‘r’).read() # Допускается использование   
‘spam\nSPAM\n’ # любой кодировки семейства utf-8  
>>> open(‘temp.txt’, ‘r’, encoding=’utf-8’).read()  
‘spam\nSPAM\n’  
>>> open(‘temp.txt’, ‘r’, encoding=’utf-8-sig’).read()  
‘spam\nSPAM\n’  
Наконец, при использовании кодировки «utf-16» обработка маркера BOM про-  
utf-16» обработка маркера BOM про-  
-16» обработка маркера BOM про-  
BOM про-  
 про-  
изводится автоматически: при выводе данные записываются с использованием   
аппаратного порядка следования байтов, а маркер BOM всегда записывается   
в файл� при вводе данные декодируются с учетом BOM, а сам маркер BOM всег-  
BOM, а сам маркер BOM всег-  
, а сам маркер BOM всег-  
BOM всег-  
 всег-  
да пропускается. С помощью более специфичных кодировок UTF-16 можно   
явно указать иной порядок следования байтов, однако при этом вам, возмож-  
но, придется вручную записывать и пропускать маркер BOM, если он необхо-  
BOM, если он необхо-  
, если он необхо-  
дим или присутствует в файле:  
>>> sys.byteorder  
‘little’  
>>> open(‘temp.txt’, ‘w’, encoding=’utf-16’).write(‘spam\nSPAM\n’)  
10  
>>> open(‘temp.txt’, ‘rb’).read()  
b’\xff\xfes\x00p\x00a\x00m\x00\r\x00\n\x00S\x00P\x00A\x00M\x00\r\x00\n\x00’  
>>> open(‘temp.txt’, ‘r’, encoding=’utf-16’).read()  
‘spam\nSPAM\n’  
   
>>> open(‘temp.txt’, ‘w’, encoding=’utf-16-be’).write(‘﻿spam\nSPAM\n’)  
11  
>>> open(‘spam.txt’, ‘rb’).read()  
b’\xfe\xff\x00s\x00p\x00a\x00m\x00\r\x00\n\x00S\x00P\x00A\x00M\x00\r\x00\n’  
>>> open(‘temp.txt’, ‘r’, encoding=’utf-16’).read()  
‘spam\nSPAM\n’  
>>> open(‘temp.txt’, ‘r’, encoding=’utf-16-be’).read()  
‘﻿spam\nSPAM\n’  
Более специфичные кодировки UTF-16 отлично подходят для работы с файла-  
UTF-16 отлично подходят для работы с файла-  
-16 отлично подходят для работы с файла-  
ми, в которых отсутствует маркер BOM, тогда как при использовании кодиров-  
BOM, тогда как при использовании кодиров-  
, тогда как при использовании кодиров-  
ки «utf-16» требуется наличие маркера в файле, чтобы при выполнении опера-  
utf-16» требуется наличие маркера в файле, чтобы при выполнении опера-  
-16» требуется наличие маркера в файле, чтобы при выполнении опера-  
ции чтения интерпретатор мог определить порядок следования байтов:  
>>> open(‘temp.txt’, ‘w’, encoding=’utf-16-le’).write(‘SPAM’)  
4  
>>> open(‘temp.txt’, ‘rb’).read() # Маркер BOM отсутствует и не ожидается  
b’S\x00P\x00A\x00M\x00’  
>>> open(‘temp.txt’, ‘r’, encoding=’utf-16-le’).read()  
‘SPAM’  
>>> open(‘temp.txt’, ‘r’, encoding=’utf-16’).read()  
UnicodeError: UTF-16 stream does not start with BOM

Другие инструменты для работы со строками в Python 3.0   
1031  
Поэкспериментируйте с этими кодировками самостоятельно или загляните   
в руководство по стандартной библиотеке языка Python, где вы найдете больше   
информации о маркере BOM.  
Файлы Юникода в Python 2.6  
Все, что говорилось выше, относится к строковым типам и файлам в Python 3.0.   
Однако аналогичный эффект можно получить и при работе с файлами Юнико-  
да в Python 2.6, правда для этого придется использовать другие функции. Если   
в версии 2.6 вместо объектов типа str использовать объекты типа unicode и от-  
крыть файл с помощью функции codecs.open, вы получите практически тот же   
результат:  
C:\misc> c:\python26\python  
>>> S = u’A\xc4B\xe8C’  
>>> print S  
AÄBèC  
>>> len(S)  
5  
>>> S.encode(‘latin-1’)  
‘A\xc4B\xe8C’  
>>> S.encode(‘utf-8’)  
‘A\xc3\x84B\xc3\xa8C’  
   
>>> import codecs  
>>> codecs.open(‘latindata’, ‘w’, encoding=’latin-1’).write(S)  
>>> codecs.open(‘utfdata’, ‘w’, encoding=’utf-8’).write(S)  
   
>>> open(‘latindata’, ‘rb’).read()  
‘A\xc4B\xe8C’  
>>> open(‘utfdata’, ‘rb’).read()  
‘A\xc3\x84B\xc3\xa8C’  
   
>>> codecs.open(‘latindata’, ‘r’, encoding=’latin-1’).read()  
u’A\xc4B\xe8C’  
>>> codecs.open(‘utfdata’, ‘r’, encoding=’utf-8’).read()  
u’A\xc4B\xe8C’  
Другие инструменты   
для работы со строками в Python 3.0  
Некоторые другие популярные инструменты из стандартной библиотеки Py-  
Py-  
thon для работы со строками были обновлены с учетом нового деления типов   
str/bytes. Мы не будем подробно рассматривать все эти прикладные инстру-  
менты в книге, посвященной основам языка, и завершим эту главу коротким   
знакомством с четырьмя из них: с модулем re, позволяющем выполнять сопо-  
ставление по шаблону, с модулем struct, предназначенным для работы с дво-  
ичными данными, с модулем pickle преобразования объектов в последователь-  
ную форму, и с пакетом xml, предназначенным для синтаксического анализа   
разметки XML.

1032   
Глава 36. Юникод и строки байтов   
Модуль re для сопоставления с шаблонами  
Модуль re, реализующий средства сопоставления строк с шаблонами и входя-  
щий в состав стандартной библиотеки Python, обеспечивает более универсаль-  
Python, обеспечивает более универсаль-  
, обеспечивает более универсаль-  
ные способы работы со строками, чем простые строковые методы, такие как   
find, split и replace. При использовании модуля re строки, которые требуется   
отыскать или по которым требуется выполнить разбиение исходного текста,   
могут быть описаны в виде обобщенных шаблонов, а не буквального текста.   
В Python 3.0 этот модуль способен работать с любыми строковыми типами –   
str, bytes и bytearray – и в качестве результата возвращает строки того же типа,   
что и испытуемый строковый объект.  
Ниже демонстрируется, как можно использовать этот модуль в Python 3.0 для   
извлечения подстрок из текстовой строки. В пределах строки шаблона кон-  
струкция (.\*) означает ноль или более (\*) любых символов (.), которые должны   
сохраняться в виде подстроки совпадения (()). Части исходной строки, совпав-  
шие с частями шаблона в круглых скобках, можно получить после успешного   
поиска с помощью метода group или groups:  
C:\misc> c:\python30\python  
>>> import re  
>>> S = ‘Bugger all down here on earth!’ # Строка текста  
>>> B = b’Bugger all down here on earth!’ # То, что обычно получается в   
 # результате чтения из файла  
>>> re.match(‘(.\*) down (.\*) on (.\*)’, S).groups() # Выборка совпадений с   
 # шаблоном  
(‘Bugger all’, ‘here’, ‘earth!’) # Совпавшие подстроки  
   
>>> re.match(b’(.\*) down (.\*) on (.\*)’, B).groups() # Подстроки типа bytes   
(b’Bugger all’, b’here’, b’earth!’)  
В Python 2.6 эти операции возвращают похожие результаты, только для пред-  
Python 2.6 эти операции возвращают похожие результаты, только для пред-  
 2.6 эти операции возвращают похожие результаты, только для пред-  
ставления текста с символами, выходящими за пределы диапазона ASCII, ис-  
ASCII, ис-  
, ис-  
пользуется строковый тип unicode, а тип str может применяться для работы   
с 8-битными текстовыми и двоичными данными:  
C:\misc> c:\python26\python  
>>> import re  
>>> S = ‘Bugger all down here on earth!’ # Простой текст и двоичные данные  
>>> U = u’Bugger all down here on earth!’ # Текст Юникода  
   
>>> re.match(‘(.\*) down (.\*) on (.\*)’, S).groups()  
(‘Bugger all’, ‘here’, ‘earth!’)  
   
>>> re.match(‘(.\*) down (.\*) on (.\*)’, U).groups()  
(u’Bugger all’, u’here’, u’earth!’)  
Поскольку объекты типов bytes и str поддерживают практически одни и те же   
операции, различия между ними становятся практически незаметными. Но,   
обратите внимание, что как и в случае других функций, в Python 3.0 вы не   
сможете смешивать объекты типов str и bytes в вызовах функций модуля re   
(впрочем, если вы не планируете производить поиск по шаблону в двоичных   
данных, то вам не о чем беспокоиться):  
C:\misc> c:\python30\python  
>>> import re  
>>> S = ‘Bugger all down here on earth!’

Другие инструменты для работы со строками в Python 3.0   
1033  
>>> B = b’Bugger all down here on earth!’  
   
>>> re.match(‘(.\*) down (.\*) on (.\*)’, B).groups()  
TypeError: can’t use a string pattern on a bytes-like object  
   
>>> re.match(b’(.\*) down (.\*) on (.\*)’, S).groups()  
TypeError: can’t use a bytes pattern on a string-like object  
   
>>> re.match(b’(.\*) down (.\*) on (.\*)’, bytearray(B)).groups()  
(bytearray(b’Bugger all’), bytearray(b’here’), bytearray(b’earth!’))  
   
>>> re.match(‘(.\*) down (.\*) on (.\*)’, bytearray(B)).groups()  
TypeError: can’t use a string pattern on a bytes-like object  
Модуль struct для работы с двоичными данными  
Модуль struct в языке Python используется для создания и извлечения упа-  
Python используется для создания и извлечения упа-  
 используется для создания и извлечения упа-  
кованных двоичных данных из строк. В версии 3.0 он действует точно так же,   
как и в версии 2.X, с той лишь разницей, что упакованные двоичные данные   
могут быть представлены исключительно объектами типа bytes и bytearray (что   
имеет определенный смысл, особенно если учесть, что эти типы предназначе-  
ны для работы с двоичными данными, а не с произвольным текстом).  
Ниже приводится пример, в котором в строку упаковываются три объекта,   
в соответствии с двоичной спецификацией (создаются чеырехбайтовое целое,   
четырехбайтовая строка и двухбайтовое целое):  
C:\misc> c:\python30\python  
>>> from struct import pack  
>>> pack(‘>i4sh’, 7, ‘spam’, 8) # Тип bytes в 3.0 (строка 8-битных данных)  
b’\x00\x00\x00\x07spam\x00\x08’  
   
C:\misc> c:\python26\python  
>>> from struct import pack  
>>> pack(‘>i4sh’, 7, ‘spam’, 8) # Тип str в 2.6 (строка 8-битных данных)  
‘\x00\x00\x00\x07spam\x00\x08’  
Однако поскольку тип bytes имеет интерфейс, практически идентичный ин-  
терфейсу типа str в обеих версиях Python, 3.0 и 2.6, большинству программи-  
Python, 3.0 и 2.6, большинству программи-  
, 3.0 и 2.6, большинству программи-  
стов, вероятно, не придется беспокоиться – различия между ними не скажутся   
на работоспособности большинства существующих программ, особенно если   
учесть, что при чтении из двоичных файлов объекты типа bytes создаются ав-  
томатически. Даже при том, что последняя операция в примере ниже терпит   
неудачу из-за несоответствия типов, тем не менее, в большинстве программ   
двоичные данные обычно читаются из файла, а не создаются в виде строк:  
C:\misc> c:\python30\python  
>>> import struct  
>>> B = struct.pack(‘>i4sh’, 7, ‘spam’, 8)  
>>> B  
b’\x00\x00\x00\x07spam\x00\x08’  
   
>>> vals = struct.unpack(‘>i4sh’, B)  
>>> vals  
(7, b’spam’, 8)  
   
>>> vals = struct.unpack(‘>i4sh’, B.decode())  
TypeError: ‘str’ does not have the buffer interface

1034   
Глава 36. Юникод и строки байтов   
За исключением нового синтаксиса определения объектов типа bytes, создание   
и чтение двоичных данных из файлов в версии 3.0 выполняется точно так же,   
как и в версии 2.X. Ниже приводится программный код, в котором тип bytes   
объектов становится наиболее заметным:  
C:\misc> c:\python30\python  
   
# Запись упакованных двоичных данных в двоичный файл  
   
>>> F = open(‘data.bin’, ‘wb’) # Открыть файл в двоичном режиме для записи  
>>> import struct  
>>> data = struct.pack(‘>i4sh’, 7, ‘spam’, 8) # Создать упаков. двоич. данные  
>>> data # В 3.0 – тип bytes, не str  
b’\x00\x00\x00\x07spam\x00\x08’  
>>> F.write(data) # Записать в файл  
10  
>>> F.close()  
   
# Чтение упакованных двоичных данных из двоичного файла  
>>> F = open(‘data.bin’, ‘rb’) # Открыть файл в двоичном режиме для чтения  
>>> data = F.read() # Прочитать байты  
>>> data  
b’\x00\x00\x00\x07spam\x00\x08’  
>>> values = struct.unpack(‘>i4sh’, data) # Извлечь упаков. двоичные данные  
>>> values # обратно в объекты языка Python  
(7, b’spam’, 8)  
После извлечения упакованных двоичных данных в объекты языка Python,   
как в этом примере, вы можете попробовать еще глубже погрузиться в двоич-  
ный мир – из строк можно извлекать значения отдельных байтов и получать   
срезы� из целых чисел с помощью битовых операций можно извлекать отдель-  
ные биты и так далее (операции, которые демонстрируются ниже, подробно   
были описаны ранее в этой книге):  
>>> values # Результат вызова метода struct.unpack  
(7, b’spam’, 8)  
   
# Доступ к отдельным битам в целых числах  
   
>>> bin(values[0]) # Можно получить целое число в двоичном виде  
‘0b111’  
>>> values[0] & 0x01 # Проверит первый (младший) бит в целом числе  
1  
>>> values[0] | 0b1010 # Битовое ИЛИ: установит указанные биты  
15  
>>> bin(values[0] | 0b1010) # 15 – десятичное, 1111 – двоичное  
‘0b1111’  
>>> bin(values[0] ^ 0b1010) # Битовое ИСКЛЮЧАЮЩЕЕ ИЛИ: сбросит биты   
‘0b1101’ # с одинаковыми значениями  
>>> bool(values[0] & 0b100) # Проверит, установлен ли бит 3  
True  
>>> bool(values[0] & 0b1000) # Проверит, установлен ли бит 4  
False  
Так как строки bytes, получающиеся в результате извлечения двоичных дан-  
ных, являются последовательностями коротких целых чисел, мы можем при-  
менить те же самые операции к отдельным байтам строки:

Другие инструменты для работы со строками в Python 3.0   
1035  
# Доступ к байтам в получившихся строках и к битам внутри них  
   
>>> values[1]  
b’spam’  
>>> values[1][0] # Строка байтов: последовательность целых чисел  
115  
>>> values[1][1:] # Выведет в виде символов ASCII  
b’pam’  
>>> bin(values[1][0]) # Байты в строках можно получить в двоичном виде  
‘0b1110011’  
>>> bin(values[1][0] | 0b1100) # Установит указанные биты  
‘0b1111111’  
>>> values[1][0] | 0b1100  
127  
Конечно, большинству программистов на языке Python не придется иметь дело   
с отдельными битами – в языке Python имеются более высокоуровневые типы   
объектов, такие как списки и словари, которые обычно лучше подходят для   
представления информации в программах на языке Python. Однако если вам   
придется анализировать или воспроизводить низкоуровневые данные, исполь-  
зуемые в программах на языке C, сетевыми и другими библиотеками, то вы   
будете знать, что в языке Python имеются инструменты, которые помогут вам   
в этом.  
Модуль pickle для сериализации объектов  
Мы уже встречались с модулем pickle в главах 9 и 30. В главе 27 мы также ис-  
пользовали модуль shelve, который в свою очередь использует модуль pickle.   
Имейте в виду, что в Python 3.0 модуль pickle всегда создает объекты типа bytes   
независимо от используемой версии «протокола» (формата данных). Убедиться   
в этом можно, воспользовавшись функцией dumps из модуля, которая возвра-  
щает строку с объектом в последовательной форме:  
C:\misc> C:\Python30\python  
>>> import pickle # dumps() возвращает строку с сериализованным объектом  
>>> pickle.dumps([1, 2, 3]) # В Python 3.0 протокол по умолчанию=3=двоичный  
b’\x80\x03]q\x00(K\x01K\x02K\x03e.’  
   
>>> pickle.dumps([1, 2, 3], protocol=0) # Протокол ASCII 0, но все равно   
b’(lp0\nL1L\naL2L\naL3L\na.’ # возвращает объект bytes!  
Вследствие этого подразумевается, что файлы, в которых сохраняются сериа-  
лизованные объекты, в Python 3.0 всегда должны открываться в двоичном ре-  
жиме, потому что для представления данных в текстовых файлах используют-  
ся объекты типа str, а не bytes – функция dump просто пытается записать строку   
с объектом в файл, открытый для записи:  
>>> pickle.dump([1, 2, 3], open(‘temp’, ‘w’)) # В текстовый файл нельзя   
 # записать объект bytes!  
TypeError: can’t write bytes to text stream # Независимо от протокола  
   
>>> pickle.dump([1, 2, 3], open(‘temp’, ‘w’), protocol=0)  
TypeError: can’t write bytes to text stream  
   
>>> pickle.dump([1, 2, 3], open(‘temp’, ‘wb’)) # Всегда используйте двоичный   
>>> open(‘temp’, ‘r’).read() # режим в 3.0  
UnicodeEncodeError: ‘charmap’ codec can’t encode character ‘€’ in ...

1036   
Глава 36. Юникод и строки байтов   
Поскольку сериализованные данные не декодируются в текст Юникода, то же   
самое относится к операциям чтения из файла – в версии 3.0 запись и чтение   
сериализованных данных всегда должны выполняться в двоичном режиме:  
>>> pickle.dump([1, 2, 3], open(‘temp’, ‘wb’))  
>>> pickle.load(open(‘temp’, ‘rb’))  
[1, 2, 3]  
>>> open(‘temp’, ‘rb’).read()  
b’\x80\x03]q\x00(K\x01K\x02K\x03e.’  
В Python 2.6 (и в более ранних версиях) мы можем использовать текстовый ре-  
Python 2.6 (и в более ранних версиях) мы можем использовать текстовый ре-  
 2.6 (и в более ранних версиях) мы можем использовать текстовый ре-  
жим работы с файлами для сериализованных данных при условии использо-  
вания протокола 0 (используется по умолчанию в 2.6) и непротиворечивого ис-  
пользования текстового режима для преобразования символов конца строки:  
C:\misc> c:\python26\python  
>>> import pickle  
>>> pickle.dumps([1, 2, 3]) # В Python 2.6 протокол по умолчанию=0=ASCII  
‘(lp0\nI1\naI2\naI3\na.’  
   
>>> pickle.dumps([1, 2, 3], protocol=1)  
‘]q\x00(K\x01K\x02K\x03e.’  
   
>>> pickle.dump([1, 2, 3], open(‘temp’, ‘w’)) # Текстовый режим   
>>> pickle.load(open(‘temp’)) # допускается в 2.6  
[1, 2, 3]  
>>> open(‘temp’).read()  
‘(lp0\nI1\naI2\naI3\na.’  
Однако, если вы заинтересованы в переносимости программного кода между   
версиями или не хотите беспокоиться о версии протокола или его значениях   
по умолчанию в разных версиях, всегда используйте для хранения сериализо-  
ванных данных файлы, открытые в двоичном режиме, – следующий пример   
одинаково действует в Python 3.0 и 2.6:  
>>> import pickle  
>>> pickle.dump([1, 2, 3], open(‘temp’, ‘wb’)) # Не зависит от версии Python  
>>> pickle.load(open(‘temp’, ‘rb’)) # И требуется в 3.0  
[1, 2, 3]  
Поскольку в большинстве программ сохранение и извлечение сериализован-  
ных объектов производится интерпретатором автоматически и программному   
коду не приходится иметь дело непосредственно с сериализованными данны-  
ми, требование всегда использовать двоичный режим при работе с файлами   
является единственным существенным отличием новой модели сохранения   
объектов в Python 3.0. Более подробные сведения о механизмах сохранения   
объектов ищите в справочной литературе или в руководствах по языку Python.  
Инструменты синтаксического анализа разметки XML  
XML – это язык разметки, основанный на тегах, используемый для пред-  
 – это язык разметки, основанный на тегах, используемый для пред-  
ставления информации в структурированном виде. Часто применяется для   
оформления документов и данных, доставляемых через Веб. Некоторую долю   
информации можно извлечь из текста XML с помощью простых строковых ме-  
XML с помощью простых строковых ме-  
 с помощью простых строковых ме-  
тодов или модуля re, однако для извлечения информации из многоуровневых   
конструкций и из атрибутов тегов требуется выполнять более точный и более   
полный синтаксический анализ разметки.

Другие инструменты для работы со строками в Python 3.0   
1037  
Вследствие того, что формат XML получил чрезвычайно широкое распростра-  
XML получил чрезвычайно широкое распростра-  
 получил чрезвычайно широкое распростра-  
нение, в состав Python был включен целый пакет инструментов для синтак-  
Python был включен целый пакет инструментов для синтак-  
 был включен целый пакет инструментов для синтак-  
сического анализа разметки XML, поддерживающих модели парсинга SAX   
и DOM, а также пакет, известный под названием ElementTree  – интерфейс   
на языке Python, позволяющий анализировать и конструировать документы   
XML. Помимо простого синтаксического анализа среди свободного программ-  
. Помимо простого синтаксического анализа среди свободного программ-  
ного обеспечения можно найти поддержку дополнительных инструментов   
XML, таких как XPath, Xquery, XSLT, и многих других.  
Формат XML по умолчанию представляет текст в виде Юникода с целью обеспе-  
XML по умолчанию представляет текст в виде Юникода с целью обеспе-  
 по умолчанию представляет текст в виде Юникода с целью обеспе-  
чить поддержку интернационализации. Большинство инструментов синтак-  
сического анализа XML в языке Python всегда возвращали строки Юникода,   
однако тип результатов изменился с unicode в Python 2.X на более универсаль-  
Python 2.X на более универсаль-  
 2.X на более универсаль-  
X на более универсаль-  
 на более универсаль-  
ный str в Python 3.0. Это вполне объяснимо, если вспомнить, что в Python 3.0   
строки типа str являются строками Юникода независимо от того, использует-  
ся кодировка ASCII или какая-то другая.  
Мы не будем углубляться в детали, а просто рассмотрим некоторые примеры,   
чтобы получить общее представление. Предположим, что у нас имеется про-  
стой файл XML mybooks.xml:  
<books>  
 <date>2009</date>  
 <title>Learning Python</title>  
 <title>Programming Python</title>  
 <title>Python Pocket Reference</title>  
 <publisher>O’Reilly Media</publisher>  
</books>  
и нам требуется извлечь и отобразить содержимое всех вложенных тегов title,   
как показано ниже:  
Learning Python  
Programming Python  
Python Pocket Reference  
Существует по меньшей мере четыре основных способа решить поставленную   
задачу (не учитывая дополнительных инструментов, таких как XPath). Во-  
первых, мы могли бы выполнить поиск по шаблону, однако такой способ не   
дает желаемой точности, когда содержимое документа XML невозможно пред-  
сказать. Там, где это применимо, с подобной работой прекрасно справляется   
модуль re, с которым мы познакомились выше. Его метод match отыскивает со-  
впадения с начала строки, метод search заглядывает вперед в поисках совпаде-  
ний, а метод findall, используемый здесь, отыскивает в строке все совпадения   
с шаблоном (и возвращает результат в виде списка совпавших подстрок, соот-  
ветствующих группам в круглых скобках, или кортежей для множественных   
групп):  
# Файл patternparse.py  
   
import re  
text = open(‘mybooks.xml’).read()  
found = re.findall(‘<title>(.\*)</title>’, text)  
for title in found: print(title)  
Во-вторых, для большей надежности мы могли бы выполнить полный син-  
таксический анализ разметки XML с помощью парсера  DOM, имеющегося

1038   
Глава 36. Юникод и строки байтов   
в стандартной библиотеке. Парсер DOM преобразует документ XML в дерево   
объектов и предоставляет интерфейс для навигации по дереву, извлечения   
атрибутов и значений тегов – интерфейс имеет формальную спецификацию, не   
зависящую от языка Python:  
# Файл domparse.py  
   
from xml.dom.minidom import parse, Node  
xmltree = parse(‘mybooks.xml’)  
for node1 in xmltree.getElementsByTagName(‘title’):  
 for node2 in node1.childNodes:  
 if node2.nodeType == Node.TEXT\_NODE:  
 print(node2.data)  
В качестве третьего варианта можно было бы использовать парсер SAX, также   
входящий в стандартную библиотеку. В модели SAX в процессе анализа произ-  
SAX в процессе анализа произ-  
 в процессе анализа произ-  
водятся вызовы методов класса, которым передается дополнительная инфор-  
мация, позволяющая определить, в каком месте документа находится парсер,   
и выбрать необходимые данные:  
# Файл saxparse.py  
   
import xml.sax.handler  
class BookHandler(xml.sax.handler.ContentHandler):  
 def \_\_init\_\_(self):  
 self.inTitle = False  
 def startElement(self, name, attributes):  
 if name == ‘title’:  
 self.inTitle = True  
 def characters(self, data):  
 if self.inTitle:  
 print(data)  
 def endElement(self, name):  
 if name == ‘title’:  
 self.inTitle = False  
   
import xml.sax  
parser = xml.sax.make\_parser()  
handler = BookHandler()  
parser.setContentHandler(handler)  
parser.parse(‘mybooks.xml’)  
Наконец, система ElementTree, доступная в виде пакета etree в стандартной   
библиотеке, часто позволяет добиться того же эффекта, что и парсеры XML   
DOM, но за счет меньшего объема программного кода. Этот характерный для   
Python способ позволяет анализировать и конструировать документы XML.   
После анализа документа система ElementTree обеспечивает доступ к компо-  
нентам документа:  
# Файл etreeparse.py  
   
from xml.etree.ElementTree import parse  
tree = parse(‘mybooks.xml’)  
for E in tree.findall(‘title’):  
 print(E.text)  
Все четыре сценария отображают одни и те же результаты в обеих версиях Py-  
Py-  
thon, 2.6 и 3.0:

В заключение   
1039  
C:\misc> c:\python26\python domparse.py  
Learning Python  
Programming Python  
Python Pocket Reference  
   
C:\misc> c:\python30\python domparse.py  
Learning Python  
Programming Python  
Python Pocket Reference  
Однако с технической точки зрения, некоторые из этих сценариев при работе   
под управлением Python 2.6 будут создавать строковые объекты типа unicode,   
тогда как под управлением Python 3.0 все они будут создавать строки типа str,   
потому что именно этот тип данных охватывает Юникод (будь то набор симво-  
лов ASCII или какой-то другой):  
C:\misc> c:\python30\python  
>>> from xml.dom.minidom import parse, Node  
>>> xmltree = parse(‘mybooks.xml’)  
>>> for node in xmltree.getElementsByTagName(‘title’):  
... for node2 in node.childNodes:  
... if node2.nodeType == Node.TEXT\_NODE:  
... node2.data  
...  
‘Learning Python’  
‘Programming Python’  
‘Python Pocket Reference’  
   
C:\misc> c:\python26\python  
>>> ...тот же программный код...  
...  
u’Learning Python’  
u’Programming Python’  
u’Python Pocket Reference’  
Программы, выполняющие обработку результатов анализа документов XML   
нетривиальными способами, должны учитывать различия в типах объектов,   
имеющиеся между версиями Python 2.6 и 3.0. Однако поскольку в версиях 2.6   
и 3.0 все строки имеют практически идентичные интерфейсы, различия меж-  
ду типами объектов не будут иметь большого значения для большинства про-  
грамм – методы, доступные в объектах типа unicode в версии 2.6, точно так же   
доступны в в объектах типа str в версии 3.0.  
К сожалению, дальнейшее обсуждение особенностей синтаксического анализа   
документов XML выходит далеко за рамки этой книги. Если вас заинтересо-  
XML выходит далеко за рамки этой книги. Если вас заинтересо-  
 выходит далеко за рамки этой книги. Если вас заинтересо-  
вала тема анализа текстовых файлов или документов XML, подробное ее осве-  
XML, подробное ее осве-  
, подробное ее осве-  
щение вы найдете в следующей книге, «Программирование на Python», посвя-  
щенной прикладным аспектам программирования на языке Python. Дополни-  
тельные подробности относительно модулей re, struct, pickle и инструментов   
для работы с разметкой XML вы найдете в Сети, в вышеупомянутой и в других   
книгах, а также в руководстве по стандартной библиотеке Python.  
В заключение  
В этой главе мы исследовали дополнительные строковые типы, доступные   
в Python 3.0 и 2.6, позволяющие обрабатывать текст Юникода и двоичные дан-  
Python 3.0 и 2.6, позволяющие обрабатывать текст Юникода и двоичные дан-  
 3.0 и 2.6, позволяющие обрабатывать текст Юникода и двоичные дан-

1040   
Глава 36. Юникод и строки байтов   
ные. Как мы узнали, многие программисты используют текст ASCII и вполне   
могут обойтись базовыми строковыми типами и операциями. Для более слож-  
ных случаев Python предоставляет строковую модель, которая обеспечивает   
полную поддержку текста Юникода, состоящего из многобайтовых симво-  
лов (в виде обычного строкового типа в версии 3.0 и специального типа в вер-  
сии 2.6), и двоичных данных (которые могут быть представлены с помощью   
объектов типа bytes в Python 3.0 и с помощью обычных строк – в Python 2.6).  
Кроме того, мы узнали, как изменилась реализация объектов файлов в Py-  
Py-  
thon 3.0, которые теперь автоматически выполняют кодирование и декодиро-  
 3.0, которые теперь автоматически выполняют кодирование и декодиро-  
вание текста Юникода и обрабатывают строки байтов при работе с двоичными   
файлами. Наконец, мы коротко познакомились с некоторыми дополнительны-  
ми инструментами обработки текста и двоичных данных, входящими в состав   
стандартной библиотеки Python, и рассмотрели примеры их использования   
в версии 3.0.  
В следующей главе мы перейдем к обсуждению темы, связанной с построением   
инструментов программирования и рассмотрим способы управления доступом   
к атрибутам объектов за счет добавления программного кода, который вызы-  
вается автоматически. Однако прежде чем двинуться дальше, ответьте на кон-  
трольные вопросы к этой главе, чтобы закрепить знания, полученные здесь.  
Закрепление пройденного  
Контрольные вопросы  
1. Какие строковые типы в Python 3.0 вы знаете и каковы их роли?  
2. Какие строковые типы в Python 2.6 вы знаете и каковы их роли?  
3. Насколько совпадают строковые типы в Python 2.6 и 3.0?  
4. Какие различия в поддерживаемых операциях имеются между строковы-  
ми типами в Python 3.0?  
5. Как можно представить символы Юникода, не входящие в набор ASCII,   
в строках в Python 3.0?  
6. Назовите основные отличия между текстовыми и двоичными файлами   
в Python 3.0.  
7. Как бы вы прочитали содержимое текстового файла, текст в котором пред-  
ставлен в кодировке, отличной от кодировки, используемой по умолчанию   
в вашей системе?  
8. Как создать текстовый файл в определенной кодировке?  
9. Почему символы ASCII рассматриваются как разновидность символов   
Юникода?  
10. Насколько существенное влияние окажут изменения строковых типов   
в Python 3.0 на ваш программный код?  
Ответы  
1. В Python 3.0 имеется три строковых типа: str (для представления текста   
Юникода, включая ASCII), bytes (для представления двоичных данных)   
и bytearray (изменяемая разновидность типа bytes). Тип str обычно исполь-

Закрепление пройденного   
1041  
зуется для представления содержимого текстовых файлов, а два других –   
для представления содержимого двоичных файлов.  
2. В Python 2.6 имеется два основных строковых типа: str (для представления   
8-битных символов и двоичных данных) и unicode (для представления строк   
многобайтовых символов). Тип str используется для представления содер-  
жимого текстовых и двоичных файлов, а тип unicode – для представления   
содержимого текстовых файлов, которые могут включать многобайтовые   
символы. В версии Python 2.6 (но не ниже) также имеется тип bytearray, но   
реализация поддержки этого типа данных является результатом переноса   
из версии 3.0, и он не делает таких строгих различий между текстовыми   
и двоичными данными, как в версии 3.0.  
3. Строковые типы в версиях 2.6 и 3.0 неточно соответствуют друг другу, по-  
тому что тип str в версии 2.6 до определенной степени эквивалентен типам   
str и bytes в версии 3.0, а тип str в версии 3.0 до определенной степени эк-  
вивалентен типам str и unicode в версии 2.6. Кроме того, тип bytearray в вер-  
сии 3.0 является уникальным.  
4. Строковые типы в Python 3.0 имеют определенный общий набор операций:   
методы, операции над последовательностями и даже такие обширные ин-  
струменты, как средства сопоставления с шаблонами, действуют одинако-  
во. С другой стороны, операции форматирования строк поддерживает толь-  
ко тип str, а тип bytearray поддерживает множество дополнительных опе-  
раций, выполняющих изменения непосредственно в самом объекте. Кроме   
того, типы str и bytes обладают методами кодирования и декодирования   
текста соответственно.  
5. Символы Юникода, не входящие в набор ASCII, можно включать в строки   
в виде экранированных шестнадцатеричных значений байтов (\xNN) и экра-  
нированных последовательностей значений Юникода (�NNNN, �NNNNNNNN).   
На некоторых клавиатурах предусмотрена возможность непосредственного   
ввода символов, не входящих в набор ASCII, таких как Latin-1.  
6. В версии 3.0 предполагается, что текстовые файлы содержат символы Юни-  
кода (даже если это символы ASCII) и при работе с ними поддерживаются   
автоматическое декодирование данных при чтении и кодирование при за-  
писи. При работе с двоичными файлами данные передаются в файл и об-  
ратно без какой-либо промежуточной обработки. Для представления содер-  
жимого текстовых файлов обычно используются объекты типа str, а для   
представления содержимого двоичных файлов – объекты типа bytes (или   
bytearray). Кроме того, при работе с текстовыми файлами для некоторых ко-  
дировок поддерживается обработка маркера BOM и автоматически выпол-  
няется преобразование символов конца строки в символ \n и обратно при   
чтении и записи, если это преобразование явно не запрещено� при работе   
с двоичными файлами ни одно из этих преобразований не выполняется.  
7. Чтобы прочитать содержимое текстового файла, текст в котором представ-  
лен в кодировке, отличной от кодировки, используемой по умолчанию в ва-  
шей системе, в Python 3.0 достаточно просто передать название кодировки   
в вызов встроенной функции open (codecs.open() – в версии 2.6). В результа-  
те данные будут декодироваться с учетом указанной кодировки в процессе   
чтения. Кроме того, данные можно прочитать в двоичном режиме и затем   
вручную декодировать их в строку, указав название требуемой кодировки,   
но такой способ является более трудоемким и при его использовании мо-

1042   
Глава 36. Юникод и строки байтов   
гут возникать ошибки, связанные с многобайтовыми символами (есть риск   
прочитать по неосторожности неполную последовательность байтов, пред-  
ставляющую один символ).  
8. Чтобы создать текстовый файл в определенной кодировке, в Python 3.0   
нужно передать название требуемой кодировки в вызов встроенной функ-  
ции open (codecs.open() – в версии 2.6). В результате в процессе записи стро-  
ки будут кодироваться в соответствии с указанной кодировкой. Кроме того,   
можно вручную закодировать строку в последовательность байтов и запи-  
сать ее в двоичном режиме, но такой способ обычно более трудоемкий.  
9. Символы ASCII рассматриваются как разновидность символов Юникода,   
потому что диапазон 7-битных значений является подмножеством боль-  
шинства кодировок Юникода. Например, допустимые символы ASCII одно-  
ASCII одно-  
 одно-  
временно являются допустимыми символами Latin-1 (в кодировке Latin-1   
все остальные возможные 8-битные значения присвоены дополнительным   
символам) и допустимыми символами UTF-8 (в кодировке UTF-8 приме-  
UTF-8 (в кодировке UTF-8 приме-  
-8 (в кодировке UTF-8 приме-  
UTF-8 приме-  
-8 приме-  
няется схема представления символов переменным числом байтов, но при   
этом символы ASCII в ней представлены теми же значениями, по одному   
байту на символ).  
10. Степень влияния изменений, появившихся в строковых типах Python 3.0,   
зависит от того, какие типы строк используются в программном коде. На   
сценарии, в которых используется только простой текст ASCII, эти изме-  
ASCII, эти изме-  
, эти изме-  
нения, скорее всего, вообще не окажут никакого влияния: в данном слу-  
чае строковый тип str действует одинаково в версиях 2.6 и 3.0. Кроме того,   
несмотря на то, что инструменты для работы со строками, имеющиеся   
в стандартной библиотеке, такие как модули re, struct, pickle и пакет xml,   
технически могут использовать типы, отличающиеся в версиях 3.0 и 2.6,   
влияние на большинство программ будет весьма незначительным, потому   
что типы str и bytes в версии 3.0 и тип str в версии 2.6 поддерживают прак-  
тически идентичные наборы операций. При работе с Юникодом вам просто   
нужно будет перейти от использования unicode и codecs.open() в версии 2.6   
к использованию str и open в версии 3.0. При работе с двоичными данными   
вам придется иметь дело с объектами типа bytes. Однако поскольку объек-  
ты этого типа имеют интерфейс, похожий на интерфейс объектов типа str   
в 2.6, воздействие описываемых изменений будет минимальным.

Глава 37.  
   
Управляемые атрибуты  
В этой главе подробно рассматривается представленный ранее прием перехва-  
та обращений к атрибутам, вводится другой прием и приводятся примеры   
их использования. Как и в других главах этой части книги, в данной главе   
рассматриваются дополнительные, расширенные темы, потому что далеко не   
всем программистам придется вникать в приемы, описываемые здесь, – они   
могут получать и изменять значения атрибутов объектов, не задаваясь вопро-  
сами реализации этих атрибутов. Однако возможность управления доступом   
к атрибутом может оказаться весьма важной особенностью, обеспечивающей   
значительную долю гибкости, особенно для разработчиков инструментальных   
средств.  
Зачем нужно управлять атрибутами?  
Атрибуты объектов занимают центральное положение в большинстве про-  
грамм на языке Python – они хранят информацию об объектах или о процес-  
Python – они хранят информацию об объектах или о процес-  
 – они хранят информацию об объектах или о процес-  
сах, протекающих в сценарии. Обычно атрибуты являются простыми имена-  
ми объектов – атрибут name, например, может быть простой строкой, получить   
и изменить значение которой можно с применением обычного синтаксиса об-  
ращения к атрибутам:  
person.name # Возвращает значение атрибута  
person.name = value # Изменяет значение атрибута  
В большинстве случаев атрибуты присоединяются непосредственно к объ-  
ектам или наследуются от родительского класса. Такой простейшей модели   
вполне достаточно для большинства программ, которые вам придется писать   
на протяжении своей карьеры программиста на языке Python.  
Однако иногда требуется обеспечить больше гибкости. Предположим, что вы   
написали программу, которая обращается к атрибуту name непосредственно,   
но затем вам потребовалось изменить ее, например добавить логику проверки   
имен при присваивании атрибуту или обеспечить получение видоизмененного   
имени при обращении к нему. Такого рода доступ к значению атрибута легко   
реализовать с помощью методов (проверка и преобразование в примере ниже   
выполняются абстрактно):

1044   
Глава 37. Управляемые атрибуты   
class Person:  
 def getName(self):  
 if not valid():  
 raise TypeError(‘cannot fetch name’)  
 else:  
 return self.name.transform()  
 def setName(self, value):  
 if not valid(value):  
 raise TypeError(‘cannot change name’)  
 else:  
 self.name = transform(value)  
   
person = Person()  
person.getName()  
person.setName(‘value’)  
Однако в этом случае придется изменить программный код везде, где произ-  
водится обращение к атрибуту name, что может оказаться не самой простой за-  
дачей. Кроме того, при таком подходе требуется помнить, как экспортируются   
значения: как простые значения или как вызовы методов. Если вы помните об   
этом и используете интерфейс доступа к данным на основе методов, клиенты   
будут защищены от изменений, – в противном случае изменения могут стать   
источником проблем.  
Данная проблема может возникать гораздо чаще, чем можно было бы ожидать.   
Значение ячейки в программе электронной таблицы, например, может начи-  
нать свое существование как простое значение, но позднее превратиться в поле,   
значение которого вычисляется по некоторой формуле. Поскольку интерфейсы   
объектов должны быть достаточно гибкими, чтобы поддерживать подобные из-  
менения в будущем, не влияя на работоспособность существующего программ-  
ного кода, переход к использованию методов выглядит не самым идеальным.  
Добавление программного кода,   
вызываемого при обращении к атрибуту  
Более удачное решение заключается в том, чтобы в случае необходимости обе-  
спечить автоматический вызов программного кода при обращениях к атрибу-  
ту. Выше в этой книге мы уже встречались с некоторыми инструментами, ко-  
торые позволяют динамически вычислять значения атрибутов при обращении   
к ним и проверять или изменять значения атрибутов при присваивании. В этой   
главе мы подробнее остановимся на инструментах, которые уже были пред-  
ставлены ранее, исследуем другие инструменты и изучим несколько больших   
примеров их использования. В частности, в этой главе будут представлены:  
 •  
Методы \_\_getattr\_\_ и \_\_setattr\_\_, которые вызываются при обращении к не-  
существующим атрибутам и при присваивании значений любым атрибу-  
там.  
 •  
Метод \_\_getattribute\_\_, который вызывается при обращении к любым атри-  
бутам в классах нового стиля в Python 2.6 и во всех классах в Python 3.0.  
 •  
Встроенная функция property, которая позволяет определить для отдель-  
ных атрибутов методы чтения и записи, – такие атрибуты часто называют   
свойствами.

Свойства   
1045  
 •  
Протокол дескрипторов, который позволяет организовать доступ к отдель-  
ным атрибутам с помощью экземпляров классов с произвольными метода-  
ми чтения и записи.  
Первый и третий инструменты уже были коротко представлены в шестой ча-  
сти� другие являются новыми темами, которые будут рассматриваться в этой   
главе.  
Как мы увидим далее, все четыре инструмента служат практически одной   
и той же цели, поэтому обычно возможно решить проблему управления досту-  
пом к атрибутам с помощью любого из них. Однако на практике они имеют не-  
сколько важных отличий друг от друга. Например, последние два инструмента   
в списке применяются к отдельным атрибутам, тогда как первые два настоль-  
ко универсальны, что могут использоваться в классах, опирающихся на прием   
делегирования, которые должны передавать произвольные атрибуты оберну-  
тым объектам. Мы также увидим, что все четыре инструмента отличаются как   
по сложности, так и по эстетике использования, о чем вы сможете судить сами   
после того, как увидите их в действии.  
Помимо изучения особенностей, составляющих основу всех четырех инстру-  
ментов управления доступом к атрибутам, в этой главе также будет предостав-  
лена возможность исследовать более крупные программы, чем те, что до этого   
рассматривались в этой книге. Так, пример CardHolder в конце главы может   
служить наглядной демонстрацией использования крупных классов. Кроме   
того, некоторые из приемов, представленных здесь, мы будем использовать   
в следующей главе при создании декораторов, поэтому вам необходимо полу-  
чить хотя бы общее представление об обсуждаемых здесь темах, прежде чем   
переходить к следующей главе.  
Свойства  
Протокол свойств позволяет направлять операции чтения и записи для отдель-  
ных атрибутов нашим функциям и методам, что позволяет нам добавлять про-  
граммный код, который будет вызываться автоматически при попытках обра-  
щения к атрибуту, перехватывать операции удаления атрибутов и возвращать   
описание атрибутов, в случае необходимости.  
Свойства создаются с помощью встроенной функции property и присваивают-  
ся атрибутам классов, точно так же, как выполняется присваивание функций   
методам. Кроме того, свойства наследуются подклассами и экземплярами, как   
любые другие атрибуты класса. Функциям, реализующим доступ к атрибу-  
там, передается сам экземпляр в виде аргумента self, что обеспечивает доступ   
к информации о состоянии объекта и к атрибутам класса, доступным объекту   
экземпляра.  
Каждое свойство управляет доступом к единственному атрибуту – свойства   
не могут перехватывать обращения ко всем атрибутам, однако они позволяют   
нам управлять операциями чтения и записи и дают возможность превратить   
атрибут из простого хранилища данных в значение, вычисляемое по произ-  
вольной формуле, не оказывая влияния на работоспособность существующего   
программного кода. Как мы увидим далее, свойства тесно связаны с дескрип-  
торами – фактически свойства являются ограниченной разновидностью де-  
скрипторов.

1046   
Глава 37. Управляемые атрибуты   
Основы  
Свойство создается операцией присваивания атрибуту класса результата, воз-  
вращаемого встроенной функцией:  
attribute = property(fget, fset, fdel, doc)  
Ни один из аргументов этой функции не является обязательным, и все они   
получают значение None по умолчанию. Если какой-то аргумент опущен, это   
означает, что соответствующая ему операция не поддерживается, а попытка   
выполнить ее приводит к исключению. При вызове функции в аргументе fget   
передается функция, которая будет вызываться при попытке прочитать зна-  
чение атрибута, в аргументе fset – функция, которая будет вызываться при   
попытке выполнить операцию присваивания, и в аргументе fdel – функция,   
которая будет вызываться при попытке удалить атрибут. В аргументе doc пере-  
дается строка документирования с описанием атрибута, если это необходимо   
(в противном случае будет скопирована строка документирования из функции   
fget, если имеется, которая по умолчанию получает значение None). Функция   
fget должна возвращать вычисленное значение атрибута, а функции fset и fdel   
ничего не должны возвращать (в действительности, они возвращают значение   
None).  
Данная встроенная функция возвращает объект свойства, присваиваемый   
имени атрибута, который будет находиться в области видимости класса и на-  
следоваться всеми его экземплярами.  
Первый пример  
Следующий класс демонстрирует, как это выглядит в действующем программ-  
 класс демонстрирует, как это выглядит в действующем программ-  
класс демонстрирует, как это выглядит в действующем программ-  
 демонстрирует, как это выглядит в действующем программ-  
демонстрирует, как это выглядит в действующем программ-  
, как это выглядит в действующем программ-  
как это выглядит в действующем программ-  
 это выглядит в действующем программ-  
это выглядит в действующем программ-  
 выглядит в действующем программ-  
выглядит в действующем программ-  
 в действующем программ-  
в действующем программ-  
 действующем программ-  
действующем программ-  
 программ-  
программ-  
ном коде. Он использует свойство для управления доступом к атрибуту с име-  
 коде. Он использует свойство для управления доступом к атрибуту с име-  
коде. Он использует свойство для управления доступом к атрибуту с име-  
. Он использует свойство для управления доступом к атрибуту с име-  
Он использует свойство для управления доступом к атрибуту с име-  
нем name – фактические данные хранятся в атрибуте с именем \_name, благодаря   
чему исключается возможность конфликта с именем свойства:  
class Person: # Используйте (object) в Python 2.6  
 def \_\_init\_\_(self, name):  
 self.\_name = name  
 def getName(self):  
 print(‘fetch...’)  
 return self.\_name  
 def setName(self, value):  
 print(‘change...’)  
 self.\_name = value  
 def delName(self):  
 print(‘remove...’)  
 del self.\_name  
 name = property(getName, setName, delName, “name property docs”)  
   
bob = Person(‘Bob Smith’) # Объект bob имеет управляемый атрибут  
print(bob.name) # Вызовет getName  
bob.name = ‘Robert Smith’ # Вызовет setName  
print(bob.name)  
del bob.name # Вызовет delName  
print(‘-’\*20)  
sue = Person(‘Sue Jones’) # Объект sue также наследует свойство  
print(sue.name)  
print(Person.name.\_\_doc\_\_) # Или help(Person.name)

Свойства   
1047  
Свойства доступны в обеих версиях Python, 2.6 и 3.0, но в версии 2.6 необходи-  
Python, 2.6 и 3.0, но в версии 2.6 необходи-  
, 2.6 и 3.0, но в версии 2.6 необходи-  
мо, чтобы класс наследовал класс object для корректной работы операций при-  
сваивания, – добавьте object как суперкласс в строку заголовка этого класса,   
прежде чем запускать пример в версии 2.6 (вы также можете добавить супер-  
класс в Python 3.0, но это это необязательно, так как суперкласс object под-  
разумевается по умолчанию).  
Данное конкретное свойство не делает ничего особенного – оно просто перехва-  
тывает обращения к атрибуту, но оно прекрасно демонстрирует действие прото-  
кола. Если запустить этот пример, он создаст два экземпляра, которые унасле-  
дуют свойство, как любой другой атрибут, присоединенный к классу. Однако   
доступ к их атрибутам находится под полным нашим контролем:  
fetch...  
Bob Smith  
change...  
fetch...  
Robert Smith  
remove...  
--------------------  
fetch...  
Sue Jones  
name property docs  
Как и все остальные атрибуты класса, свойства наследуются обоими экзем-  
плярами, а также подклассами, находящимися ниже в иерархии наследова-  
ния. Если изменить пример, как показано ниже:  
class Super:  
 ...оригинальная реализация класса Person...  
 name = property(getName, setName, delName, ‘name property docs’)  
   
class Person(Super):  
 pass # Унаследует свойства  
   
bob = Person(‘Bob Smith’)  
...остальной программный код остался без изменений...  
он выведет те же самые результаты – подкласс Person унаследовал свойство name   
от класса Super, а экземпляр bob получил его от класса Person. С точки зрения   
наследования, свойства действуют точно так же, как обычные методы, – они   
имеют доступ к экземпляру через аргумент self, поэтому они могут обращаться   
к информации о состоянии экземпляра подобно методам, как демонстрируется   
в следующем разделе.  
Вычисляемые атрибуты  
Пример в предыдущем разделе просто выводит сообщения при обращении   
к атрибуту. Однако обычно свойства реализуют более полезные действия, на-  
пример динамически вычисляют значение атрибута при обращении к нему.   
Эту возможность иллюстрирует следующий пример:  
class PropSquare:  
 def \_\_init\_\_(self, start):  
 self.value = start  
 def getX(self): # Операция получения атрибута  
 return self.value \*\* 2

1048   
Глава 37. Управляемые атрибуты   
 def setX(self, value): # Операция присваивания значения атрибуту  
 self.value = value  
 X = property(getX, setX) # Операция удаления не поддерживается,  
 # описание отсутствует  
   
P = PropSquare(3) # 2 экземпляра класса со свойством  
Q = PropSquare(32) # Каждый хранит собственное значение  
   
print(P.X) # 3 \*\* 2  
P.X = 4  
print(P.X) # 4 \*\* 2  
print(Q.X) # 32 \*\* 2  
Этот класс определяет атрибут X, к которому можно обращаться, как если бы   
это был атрибут со статическими данными, но в действительности вычисля-  
ет значение этого атрибута в момент обращения к нему. Этот эффект очень на-  
 эффект очень на-  
эффект очень на-  
 очень на-  
очень на-  
 на-  
на-  
поминает неявный вызов метода. В процессе работы этот фрагмент сохранит   
в экземпляре начальное значение, а потом, при каждой попытке обратиться   
к управляемому атрибуту, его значения автоматически возводятся в квадрат:  
9  
16  
1024  
Обратите внимание, что мы создали два различных экземпляра – благодаря   
тому, что методы свойства автоматически получают аргумент self, они полу-  
чают доступ к информации, хранящейся в экземплярах. В нашем случае это   
означает, что вычисляется квадрат числа, хранящегося в объекте экземпляра.  
Определение свойств с помощью декораторов  
Мы познакомились с основами декораторов в главе 31, однако оставили изуче-  
ние дополнительных подробностей до следующей главы. Вспомните синтаксис   
декораторов функций:  
@decorator  
def func(args): ...  
Он автоматически преобразуется интерпретатором в следующую конструк-  
цию, которая повторно присваивает имени функции результат вызова декора-  
тора:  
def func(args): ...  
func = decorator(func)  
Благодаря этой особенности встроенная функция property может играть роль   
декоратора, позволяющего определить функцию, которая автоматически бу-  
дет вызываться при попытке получить значение атрибута:   
class Person:  
 @property  
 def name(self): ... # Повторное присваивание: name = property(name)  
В процессе выполнения этого определения декорируемый метод автоматиче-  
ски передается встроенной функции property в первом аргументе. Фактически   
это всего лишь альтернативный синтаксис конструкции, которая создает свой-  
ство и повторно выполняет присваивание имени атрибута вручную:

Свойства   
1049  
class Person:  
 def name(self): ...  
 name = property(name)  
Начиная с версии Python 2.6, объекты свойств также обладают методами get-  
ter, setter и deleter, которые присваивают соответствующие методы доступа   
к свойству и возвращают копию самого свойства. Мы можем использовать эти   
методы, чтобы определить компоненты свойств, декорируя обычные методы,   
однако компонент getter обычно устанавливается автоматически, в процессе   
создания самого свойства:  
class Person:  
 def \_\_init\_\_(self, name):  
 self.\_name = name  
 @property  
 def name(self): # name = property(name)  
 “name property docs”  
 print(‘fetch...’)  
 return self.\_name  
   
 @name.setter  
 def name(self, value): # name = name.setter(name)  
 print(‘change...’)  
 self.\_name = value  
   
 @name.deleter  
 def name(self): # name = name.deleter(name)  
 print(‘remove...’)  
 del self.\_name  
   
bob = Person(‘Bob Smith’) # Объект bob имеет управляемый атрибут  
print(bob.name) # Вызовет метод getter свойства name (name 1)  
bob.name = ‘Robert Smith’ # Вызовет метод setter свойства name (name 2)  
print(bob.name)  
del bob.name # Вызовет метод deleter свойства name (name 3)  
   
print(‘-’\*20)  
sue = Person(‘Sue Jones’) # Объект sue также наследует свойство  
print(sue.name)  
print(Person.name.\_\_doc\_\_) # Или: help(Person.name)  
Фактически этот программный код эквивалентен первому примеру в этом   
разделе. В данном случае декорирование – это просто альтернативный способ   
определения свойств. Если запустить этот пример, он выведет те же результа-  
ты:  
fetch...  
Bob Smith  
change...  
fetch...  
Robert Smith  
remove...  
--------------------  
fetch...  
Sue Jones  
name property docs

1050   
Глава 37. Управляемые атрибуты   
По сравнению с присваиванием результата вызова функции property вручную,   
в данном случае применение декораторов требует добавления всего трех стро-  
чек программного кода (небольшая разница). Как это часто бывает при нали-  
чии альтернативных вариантов, выбор между ними в значительной степени   
зависит от личных предпочтений.  
Дескрипторы  
Дескрипторы обеспечивают альтернативный способ управления доступом   
к атрибутам. Они тесно связаны со свойствами, обсуждавшимися в предыду-  
щем разделе. Фактически свойства являются разновидностью дескрипторов.   
С технической точки зрения, встроенная функция property лишь упрощает   
способ создания дескриптора определенного типа, который вызывает функ-  
ции, управляющие доступом к атрибутам.  
С функциональной точки зрения, протокол дескрипторов позволяет переда-  
вать выполнение операций чтения и записи для определенного атрибута мето-  
дам отдельного объекта класса, что дает возможность определять программ-  
ный код, который будет вызываться автоматически при попытках обращения   
к атрибуту, а также при выполнении операции удаления атрибута и получения   
его описания, если это необходимо.  
Дескрипторы создаются как независимые классы и присваиваются атрибутам   
класса точно так же, как функции методов. Подобно любым другим атрибу-  
там классов они наследуются подклассами и экземплярами. Методы дескрип-  
торов, управляющие доступом, получают аргумент self со ссылкой на сам де-  
скриптор и экземпляр клиентского класса. Благодаря этому они могут сохра-  
нять и использовать собственные данные, а также данные объекта экземпляра.   
Например, дескриптор может вызывать не только методы клиентского класса,   
но и собственные методы, определенные в классе дескриптора.  
Подобно свойству дескриптор управляет доступом к единственному атрибуту –   
дескрипторы не могут использоваться для организации управления доступом   
сразу ко всем атрибутам, однако они позволяют управлять операциями чте-  
ния и записи и дают возможность превратить атрибут из простого хранилища   
данных в значение, вычисляемое по некоторой формуле, не оказывая влияния   
на работоспособность существующего программного кода. В действительности   
свойства – это упрощенный способ создания дескрипторов определенного типа,   
и, как мы увидим ниже, они могут быть определены непосредственно в виде   
дескрипторов.  
Возможности свойств достаточно ограничены, тогда как дескрипторы позволя-  
ют получить более общее решение. Например, благодаря тому, что дескрипто-  
ры определяются как обычные классы, они имеют собственные данные, могут   
занимать место в иерархиях наследования дескрипторов, могут использовать   
прием композиции для агрегирования других объектов и предоставляют   
естественный способ задания внутренних методов и строк документирования   
атрибутов.  
Основы  
Как уже упоминалось выше, дескрипторы определяются в виде отдельных   
классов и предоставляют методы доступа со специальными именами, реали-  
зующие операции доступа к атрибутам, – методы чтения, записи и удаления,

Дескрипторы   
1051  
определенные в классе дескриптора, будут вызываться автоматически при вы-  
полнении соответствующих операций над атрибутом, после того как ему будет   
присвоен экземпляр класса дескриптора:  
class Descriptor:  
 “docstring goes here”  
 def \_\_get\_\_(self, instance, owner): ... # Возвращает значение атрибута  
 def \_\_set\_\_(self, instance, value): ... # Ничего не возвращает (None)  
 def \_\_delete\_\_(self, instance): ... # Ничего не возвращает (None)  
Классы с любыми из этих методов считаются дескрипторами, а их методы ста-  
новятся специальными, когда экземпляры этих классов присваиваются атри-  
бутам других классов, – они будут вызываться автоматически при обращении   
к таким атрибутам. Если реализация какого-либо из методов отсутствует,   
в общем случае это означает, что соответствующая ему операция не поддержи-  
вается. Однако в отличие от свойств, отсутствие метода \_\_set\_\_ позволяет пере-  
определить атрибут в экземпляре и тем самым отключить дескриптор – чтобы   
сделать атрибут доступным только для чтения, необходимо определить метод   
\_\_set\_\_, который будет перехватывать операции присваивания и возбуждать   
исключение.  
Аргументы методов дескриптора  
Прежде чем перейти к изучению действующих примеров, нам необходимо ко-  
ротко познакомиться с некоторыми основами. Все три метода дескрипторов,   
представленные в предыдущем разделе, получают экземпляр класса дескрип-  
тора (self) и экземпляр клиентского класса, к которому присоединен экзем-  
пляр дескриптора (instance).  
Метод \_\_get\_\_ дополнительно принимает аргумент owner, определяющий класс,   
к которому присоединен экземпляр дескриптора. В аргументе instance ему   
передается экземпляр, к атрибуту которого выполняется обращение (instance.  
attr), или None, если обращение к атрибуту выполняется непосредственно через   
имя клиентского класса (class.attr). В первом случае метод обычно возвращает   
вычисленное значение атрибута экземпляра, а во втором – значение self, если   
поддерживается доступ к объекту дескриптора.  
Например, в следующем фрагменте, когда производится попытка получить   
значение атрибута X.attr, интерпретатор автоматически вызывает метод \_\_  
get\_\_ класса Descriptor, который присвоен атрибуту Subject.attr класса (как   
и в случае со свойствами, в Python 2.6 класс дескриптора должен наследовать   
суперкласс object – в Python 3.0 это наследование подразумевается по умолча-  
нию, хотя его указание и не повредит):  
>>> class Descriptor(object):  
... def \_\_get\_\_(self, instance, owner):  
... print(self, instance, owner, sep=’\n’)  
...  
>>> class Subject:  
... attr = Descriptor() # Атрибут класса – экземпляр класса Descriptor  
...  
>>> X = Subject()  
   
>>> X.attr  
<\_\_main\_\_.Descriptor object at 0x0281E690>  
<\_\_main\_\_.Subject object at 0x028289B0>

1052   
Глава 37. Управляемые атрибуты   
<class ‘\_\_main\_\_.Subject’>  
   
>>> Subject.attr  
<\_\_main\_\_.Descriptor object at 0x0281E690>  
None  
<class ‘\_\_main\_\_.Subject’>  
Обратите внимание на аргументы, которые автоматически передаются методу   
\_\_get\_\_ в первой попытке получить значение атрибута, – выражение X.attr как   
бы преобразуется в следующую конструкцию (хотя здесь обращение к Subject.  
attr не приводит к повторному вызову метода \_\_get\_\_):  
X.attr -> Descriptor.\_\_get\_\_(Subject.attr, X, Subject)  
Дескриптор понимает, что была выполнена попытка обращения к самому де-  
скриптору, когда получает значение None в аргументе instance.  
Дескрипторы атрибутов, доступных только для чтения  
Как уже упоминалось выше, в отличие от свойств, если в дескрипторе просто   
опустить реализацию метода \_\_set\_\_, этого будет недостаточно для создания   
атрибута, доступного только для чтения, потому что имени атрибута экземпля-  
ра, которому присвоен дескриптор, может быть присвоено любое другое значе-  
ние. Ниже демонстрируется, что операция присваивания атрибуту X.a сохра-  
няет присваиваемое значение в атрибуте экземпляра X и тем самым отключает   
дескриптор, хранящийся в классе C:  
>>> class D:  
... def \_\_get\_\_(\*args): print(‘get’)  
...  
>>> class C:  
... a = D()  
...  
>>> X = C()  
>>> X.a # Вызовет метод \_\_get\_\_ унаследованного дескриптора  
get  
>>> C.a  
get  
>>> X.a = 99 # Сохранит значение в X, отключит C.a  
>>> X.a  
99  
>>> list(X.\_\_dict\_\_.keys())  
[‘a’]  
>>> Y = C()  
>>> Y.a # Y также наследует дескриптор  
get  
>>> C.a  
get  
Именно так реализовано присваивание значений атрибутам экземпляров   
в языке Python, и это позволяет экземплярам переопределять значения по   
умолчанию в атрибутах классов. Чтобы с помощью дескриптора сделать атри-  
бут доступным только для чтения, необходимо перехватить операцию присва-  
ивания в классе дескриптора и возбудить исключение, чтобы предотвратить   
возможность присваивания. Когда выполняется присваивание атрибуту, ко-  
торый является дескриптором, интерпретатор фактически обходит обычную

Дескрипторы   
1053  
процедуру присваивания атрибутам экземпляров и обращается к объекту де-  
скриптора:  
>>> class D:  
... def \_\_get\_\_(\*args): print(‘get’)  
... def \_\_set\_\_(\*args): raise AttributeError(‘cannot set’)  
...  
>>> class C:  
... a = D()  
...  
>>> X = C()  
>>> X.a # Вызовет C.a.\_\_get\_\_  
get  
>>> X.a = 99 # Вызовет C.a.\_\_set\_\_  
AttributeError: cannot set  
Кроме того, не путайте метод дескрипторов \_\_delete\_\_ с обыч-  
ным методом \_\_del\_\_. Первый вызывается при попытке удалить   
управляемый атрибут экземпляра клиентского класса, тогда   
как второй является обычным методом деструктора экземпля-  
ра, который вызывается, когда экземпляр любого класса утили-  
зируется сборщиком мусора. Метод \_\_delete\_\_ более похож на   
метод \_\_delattr\_\_ удаления атрибутов, с которым мы познако-  
мимся ниже в этой главе. Подробнее о методах перегрузки опе-  
раторов рассказывается в главе 29.  
Первый пример  
Чтобы рассмотреть, как все это действует в комплексе, на более реалистичном   
примере, начнем с того же примера, с которого мы начинали, когда знакомились   
со свойствами. Ниже приводится определение дескриптора, который управляет   
доступом к атрибуту name в клиентском классе. Для доступа к данным в объ-  
екте экземпляра, где фактически сохраняется строка с именем, его методы ис-  
пользуют аргумент instance. Подобно свойствам дескрипторы корректно рабо-  
тают, только когда они определяются как классы нового стиля и подключаются   
к классам нового стиля, поэтому, если вы пользуетесь Python 2.6, не забудьте   
добавить суперкласс object в определения обоих классов:  
class Name: # Используйте (object) в 2.6  
 “name descriptor docs”  
 def \_\_get\_\_(self, instance, owner):  
 print(‘fetch...’)  
 return instance.\_name  
 def \_\_set\_\_(self, instance, value):  
 print(‘change...’)  
 instance.\_name = value  
 def \_\_delete\_\_(self, instance):  
 print(‘remove...’)  
 del instance.\_name  
   
class Person: # Use (object) in 2.6  
 def \_\_init\_\_(self, name):  
 self.\_name = name  
 name = Name() # Присвоить атрибуту дескриптор

1054   
Глава 37. Управляемые атрибуты   
bob = Person(‘Bob Smith’) # Объект bob имеет управляемый атрибут  
print(bob.name) # Вызовет Name.\_\_get\_\_  
bob.name = ‘Robert Smith’ # Вызовет Name.\_\_set\_\_  
print(bob.name)  
del bob.name # Вызовет Name.\_\_delete\_\_  
   
print(‘-’\*20)  
sue = Person(‘Sue Jones’) # Объект sue также наследует дескриптор  
print(sue.name)  
print(Name.\_\_doc\_\_) # Или: help(Name)  
Обратите внимание, как в этом примере выполняется присваивание экземпля-  
ра класса дескриптора атрибуту класса клиентского класса – благодаря это-  
му он будет унаследован всеми экземплярами класса, точно так же, как и ме-  
тоды класса. В действительности мы должны присвоить дескриптор атрибуту   
класса, как в данном случае, – если присвоить его атрибуту экземпляра self,   
дескриптор не будет работать. Когда вызывается метод \_\_get\_\_ дескриптора,   
ему передается три объекта, определяющие контекст вызова:  
 •  
self – экземпляр класса Name.  
 •  
instance – экземпляр класса Person.  
 •  
owner – класс Person.  
В процессе работы этого примера методы дескриптора будут перехватывать по-  
пытки обращения к атрибуту практически так же, как в версии примера со   
свойствами. Фактически этот пример выведет те же результаты:  
fetch...  
Bob Smith  
change...  
fetch...  
Robert Smith  
remove...  
--------------------  
fetch...  
Sue Jones  
name descriptor docs  
Кроме того, как и в примере со свойствами, экземпляр класса дескриптора   
является значением атрибута класса и потому будет унаследован всеми экзем-  
плярами клиентского класса и любыми подклассами. Если теперь изменить   
класс Person, как показано ниже, вывод сценария не изменится:  
...  
class Super:  
 def \_\_init\_\_(self, name):  
 self.\_name = name  
 name = Name()  
   
class Person(Super): # Унаследует дескриптор  
 pass  
...  
Обратите также внимание, что если класс дескриптора не имеет практическо-  
го применения за пределами клиентского класса, вполне разумно включить   
определение дескриптора внутрь клиентского класса. Ниже показано, как вы-  
глядит наш пример с использованием вложенного класса:

Дескрипторы   
1055  
class Person:  
 def \_\_init\_\_(self, name):  
 self.\_name = name  
   
 class Name: # Вложенный класс  
 “name descriptor docs”  
 def \_\_get\_\_(self, instance, owner):  
 print(‘fetch...’)  
 return instance.\_name  
 def \_\_set\_\_(self, instance, value):  
 print(‘change...’)  
 instance.\_name = value  
 def \_\_delete\_\_(self, instance):  
 print(‘remove...’)  
 del instance.\_name  
 name = Name()  
В этом случае имя Name становится локальной переменной в области видимости   
инструкции определения класса Person, благодаря этому оно не будет конфлик-  
товать с именами за пределами класса. Данная версия действует точно так же,   
как и оригинал. Мы просто переместили определение класса дескриптора в об-  
ласть видимости клиентского класса – однако последнюю строку в программ-  
ном коде, выполняющем тестирование, необходимо изменить, чтобы он извле-  
кал строку документирования из нового местоположения:  
...  
print(Person.Name.\_\_doc\_\_) # Изменено: за пределами класса имя Name.\_\_doc\_\_   
 # больше недоступно  
Вычисляемые атрибуты  
Как и в случае со свойствами, наш первый пример дескриптора из предыду-  
щего раздела не делает ничего особенного – он просто выводит сообщение при   
попытке обратиться к атрибуту. На практике дескрипторы часто используются   
для вычисления значений атрибутов при каждой попытке обращения к ним.   
Ниже приводится переработанная версия того же самого примера со свойства-  
ми, в котором с помощью дескриптора реализовано автоматическое возведение   
значения атрибута в квадрат при попытке получить его:  
class DescSquare:  
 def \_\_init\_\_(self, start): # Каждый дескриптор имеет свои данные  
 self.value = start  
 def \_\_get\_\_(self, instance, owner): # Операция получения значения  
 return self.value \*\* 2  
 def \_\_set\_\_(self, instance, value): # Операция присваивания  
 self.value = value # Операции удаления и получения   
 # описания не поддерживаются  
class Client1:  
 X = DescSquare(3) # Присвоить экземпляр дескриптора атрибуту класса  
   
class Client2:  
 X = DescSquare(32) # Другой экземпляр в другом клиентском классе  
 # Также можно было бы создать 2 экземпляра   
 # в одном классе  
c1 = Client1()  
c2 = Client2()  
print(c1.X) # 3 \*\* 2

1056   
Глава 37. Управляемые атрибуты   
c1.X = 4  
print(c1.X) # 4 \*\* 2  
print(c2.X) # 32 \*\* 2  
Если запустить этот пример, он выведет те же результаты, что и оригинальная   
версия, использующая свойства, только в данном случае доступом к атрибуту   
управляет объект класса дескриптора:  
9  
16  
1024  
Использование данных в дескрипторах  
Если внимательно изучить два примера дескрипторов, которые мы уже напи-  
сали, можно заметить, что они получают информацию из разных мест: в пер-  
вом примере они используют данные, хранящиеся в экземпляре клиентского   
класса (атрибут name), а во втором – данные, присоединенные непосредствен-  
но к объекту дескриптора (атрибут, значение которого возводится в квадрат).   
Фактически дескрипторы могут одновременно использовать данные экземпля-  
ра и дескриптора или любые их комбинации:  
 •  
Данные дескриптора используются для управления внутренней работой де-  
скриптора.  
 •  
Данные экземпляра хранят информацию, связанную с клиентским клас-  
сом и, возможно, созданную этим классом.  
Методы дескриптора могут использовать любые данные, но когда в дескрипто-  
рах определяются собственные атрибуты, часто бывает необязательным следо-  
вать специальным соглашениям об именовании, чтобы избежать конфликтов   
с именами атрибутов экземпляров. Например, следующий дескриптор опреде-  
ляет атрибуты собственных экземпляров, имена которых никак не конфликту-  
ют с именами атрибутов клиентского класса:  
class DescState: # Дескриптор использует собственный атрибут  
 def \_\_init\_\_(self, value):  
 self.value = value  
 def \_\_get\_\_(self, instance, owner): # Операция получения значения   
 print(‘DescState get’)  
 return self.value \* 10  
 def \_\_set\_\_(self, instance, value): # Операция присваивания  
 print(‘DescState set’)  
 self.value = value  
   
# Клиентский класс  
   
class CalcAttrs:  
 X = DescState(2) # Дескриптор атрибута класса  
 Y = 3 # Атрибут класса  
 def \_\_init\_\_(self):  
 self.Z = 4 # Атрибут экземпляра  
   
obj = CalcAttrs()  
print(obj.X, obj.Y, obj.Z) # X - вычисляется, другие - нет  
obj.X = 5 # Операция присваивания X перехватывается  
obj.Y = 6

Дескрипторы   
1057  
obj.Z = 7  
print(obj.X, obj.Y, obj.Z)  
В этом примере атрибут value находится в области видимости дескриптора,   
поэтому никаких конфликтов не возникнет, если в экземплярах клиентского   
класса будет присутствовать атрибут с таким же именем. Обратите внимание,   
что здесь только один атрибут управляется дескриптором – перехватываются   
только операции чтения и записи над атрибутом X, доступ к атрибутам Y и Z ни-  
как не регулируется (атрибут Y присоединен к клиентскому классу, а атрибут   
Z – к экземпляру). Если запустить этот пример, при обращении к атрибуту X его   
значение будет вычисляться динамически:  
DescState get  
20 3 4  
DescState set  
DescState get  
50 6 7  
Дескриптор точно так же может сохранять и использовать информацию в атри-  
бутах экземпляра клиентского класса. В следующем примере дескриптор пред-  
полагает, что экземпляр имеет атрибут \_Y, присоединенный к клиентскому   
классу, и использует его для вычисления значения атрибута, который он пред-  
ставляет:  
class InstState: # Дескриптор использует атрибут экземпляра  
 def \_\_get\_\_(self, instance, owner):  
 print(‘InstState get’) # Предполагает наличие атрибута   
 return instance.\_Y \* 100 # в клиентском классе  
 def \_\_set\_\_(self, instance, value):  
 print(‘InstState set’)  
 instance.\_Y = value  
   
# Клиентский класс  
   
class CalcAttrs:  
 X = DescState(2) # Дескриптор атрибута класса  
 Y = InstState() # Дескриптор атрибута класса  
 def \_\_init\_\_(self):  
 self.\_Y = 3 # Атрибут экземпляра  
 self.Z = 4 # Атрибут экземпляра  
   
obj = CalcAttrs()  
print(obj.X, obj.Y, obj.Z) # X и Y - вычисляемые, Z - нет  
obj.X = 5 # Присваивания атрибутам X и Y   
 # перехватываются  
obj.Y = 6  
obj.Z = 7  
print(obj.X, obj.Y, obj.Z)  
На этот раз дескрипторы присваиваются двум атрибутам, X и Y, и при попытке   
обратиться к ним их значения вычисляются динамически (атрибуту X присво-  
ен экземпляр класса дескриптора из предыдущего примера). В этом примере   
определен новый дескриптор, который не имеет собственных данных, но он   
использует атрибут, который, как предполагается, существует в экземпляре�   
этот атрибут был назван \_Y, чтобы избежать конфликта с именем самого атрибу-  
та дескриптора. Если запустить этот пример, он выведет похожие результаты,

1058   
Глава 37. Управляемые атрибуты   
но на этот раз у нас имеется второй управляемый атрибут, значение которого   
вычисляется с учетом значения атрибута экземпляра, а не самого дескриптора:  
DescState get  
InstState get  
20 300 4  
DescState set  
InstState set  
DescState get  
InstState get  
50 600 7  
Данные дескриптора и экземпляра выступают каждый в своей роли. Факти-  
чески именно в этом заключается основное преимущество дескрипторов перед   
свойствами – благодаря тому, что дескрипторы могут иметь собственные дан-  
ные, они легко могут сохранять данные внутри себя, не засоряя пространство   
имен клиентского объекта экземпляра.  
Взаимосвязь между свойствами и дескрипторами  
Как уже упоминалось выше, свойства и дескрипторы тесно связаны между со-  
бой – встроенная функция property является просто удобным способом созда-  
ния дескриптора. Теперь, когда вы знаете, как действуют свойства и дескрип-  
торы, вы сможете также понять, как можно имитировать встроенную функ-  
цию property с помощью класса дескриптора:  
class Property:  
 def \_\_init\_\_(self, fget=None, fset=None, fdel=None, doc=None):  
 self.fget = fget  
 self.fset = fset  
 self.fdel = fdel # Сохранить несвязанные методы  
 self.\_\_doc\_\_ = doc # или другие вызываемые объекты  
   
 def \_\_get\_\_(self, instance, instancetype=None):  
 if instance is None:  
 return self  
 if self.fget is None:  
 raise AttributeError(“can’t get attribute”)  
 return self.fget(instance) # Передать методу доступа экземпляр  
 # в аргументе self  
 def \_\_set\_\_(self, instance, value):  
 if self.fset is None:  
 raise AttributeError(“can’t set attribute”)  
 self.fset(instance, value)  
   
 def \_\_delete\_\_(self, instance):  
 if self.fdel is None:  
 raise AttributeError(“can’t delete attribute”)  
 self.fdel(instance)  
   
class Person:  
 def getName(self): ...  
 def setName(self, value): ...  
 name = Property(getName, setName) # Используется подобно property()  
Данный класс Property перехватывает операции доступа к атрибутам с помо-  
щью протокола дескрипторов и передает их выполнение функциям или мето-

\_\_getattr\_\_ и \_\_getattribute\_\_   
1059  
дам, сохраненным в самом дескрипторе в процессе создания класса. Операция   
получения значения атрибута, например, передается из класса Person методу   
\_\_get\_\_ класса Property и затем – методу getName класса Person. Благодаря де-  
 де-  
де-  
скрипторам этот прием «просто работает».  
Обратите внимание, что этот класс дескриптора имитирует только простейший   
случай использования функции property. Чтобы можно было использовать син-  
таксис декораторов @ и определять операции присваивания и удаления, в наш   
класс Property следовало бы добавить методы setter и deleter, которые должны   
сохранять декорированную функцию доступа и возвращать объект свойства   
(self – вполне достаточно). Поскольку все это уже реализовано во встроенной   
функции property, мы не будем продолжать доработку нашего класса.  
Обратите внимание, что дескрипторы также используются в языке Python для   
реализации атрибута \_\_slots\_\_ – при обращении к именам в слотах словари   
атрибутов экземпляров не просматриваются интерпретатором, потому что эти   
операции перехватываются с помощью дескрипторов, хранящихся на уровне   
класса. Подробнее о слотах рассказывается в главе 31.  
В главе 38 дескрипторы будут использоваться для реализации   
декораторов функций, которые могут применяться как к функ-  
циям, так и к методам. Как вы узнаете в этой главе, благодаря   
тому, что дескрипторы получают ссылку на сам дескриптор и на   
объект экземпляра класса, они прекрасно справляются с этой   
ролью, хотя вложенные функции позволяют получить более   
простое решение.  
\_\_getattr\_\_ и \_\_getattribute\_\_  
До сих пор мы изучали свойства и дескрипторы – инструменты управления от-  
дельными атрибутами. Методы \_\_getattr\_\_ и \_\_getattribute\_\_ перегрузки опе-  
раторов предоставляют иной способ управления доступом к атрибутам клас-  
сов. Подобно свойствам и дескрипторам они позволяют добавлять программ-  
ный код, который будет вызываться автоматически при попытках обращения   
к атрибутам – как вы увидите далее, эти два метода обеспечивают более обоб-  
щенные способы управления.  
Операция чтения значения атрибута может быть перехвачена с помощью двух   
разных методов:  
 •  
\_\_getattr\_\_ вызывается при обращении к неопределенным атрибутам – то   
есть к атрибутам, которые отсутствуют в экземпляре или в наследуемых им   
классах.  
 •  
\_\_getattribute\_\_ вызывается при обращении к любому атрибуту, поэтому   
при его использовании следует проявлять особую осторожность, чтобы не   
попасть в бесконечный цикл рекурсивных вызовов этого метода, и переа-  
дресовать операцию чтения суперклассу.  
С первым методом мы уже встречались в главе 29 – он доступен во всех версиях   
Python. Второй доступен в Python 2.6 только в классах нового стиля и во всех   
классах в Python 3.0 (которые уже являются классами нового стиля). Эти два   
метода входят в состав множества методов управления доступом к атрибутам,

1060   
Глава 37. Управляемые атрибуты   
в число которых также входят методы \_\_setattr\_\_ и \_\_delattr\_\_. Поскольку эти   
методы играют схожие роли, мы объединим их в рамках одной темы.  
В отличие от свойств и дескрипторов, эти методы являются частью протокола   
перегрузки операторов в языке Python – комплекса методов со специальны-  
Python – комплекса методов со специальны-  
 – комплекса методов со специальны-  
ми именами, которые наследуются подклассами и автоматически вызываются   
при использовании экземпляров в соответствующей операции. Подобно лю-  
бым методам класса все они принимают ссылку на экземпляр в первом аргу-  
менте self, благодаря которой получают доступ к любым данным экземпляра   
и к другим методам класса.  
Кроме того, \_\_getattr\_\_ и \_\_getattribute\_\_ позволяют реализовать более обоб-  
щенный способ управления атрибутами, чем свойства и дескрипторы, – они   
могут использоваться для управления чтением любых (или даже всех) атри-  
бутов экземпляра. Благодаря такой особенности эти два метода отлично под-  
ходят для реализации шаблона делегирования  – они могут использоваться   
для реализации объектов-оберток, управляющих доступом ко всем атрибутам   
встроенного объекта. В противоположность этому при использовании свойств   
и дескрипторов мы можем управлять доступом только для каждого атрибута   
в отдельности.  
Наконец, эти два метода являются более узкоспециализированными по срав-  
нению с альтернативными решениями, с которыми мы познакомились выше:   
они перехватывают только операцию чтения значения атрибута, но не опера-  
цию присваивания. Чтобы перехватить операцию присваивания, нам также   
потребуется реализовать метод \_\_setattr\_\_ – метод перегрузки операторов,   
который вызывается при попытке присвоить значение любому атрибуту и тре-  
бующий особой осторожности, чтобы не попасть в бесконечный цикл рекур-  
сивных вызовов, выполняя присваивание посредством словаря пространства   
имен экземпляра.  
Чтобы перехватывать операцию удаления атрибутов, мы можем также реали-  
зовать метод \_\_delattr\_\_ (точно так же проявляя осторожность, чтобы избежать   
зацикливания), хотя на практике он используется гораздо реже. В противопо-  
ложность этим методам, свойства и дескрипторы перехватывают операции чте-  
ния, присваивания и удаления по умолчанию.  
Большинство из этих методов перегрузки операторов были представлены ранее   
в книге – здесь мы подробнее познакомимся с особенностями их использова-  
ния и изучим их роли в более широком контексте.  
Основы  
Методы \_\_getattr\_\_ и \_\_setattr\_\_ были представлены в главах 29 и 31, а ме-  
тод \_\_getattribute\_\_ коротко упоминался в главе 31. В двух словах, если класс   
определяет или наследует следующие методы, они будут вызываться автома-  
тически в случаях, которые описываются в комментариях справа:  
def \_\_getattr\_\_(self, name): # Обращение к неопределенному атрибуту   
 # [obj.name]  
def \_\_getattribute\_\_(self, name): # Обращение к любому атрибуту [obj.name]  
def \_\_setattr\_\_(self, name, value): # Присваивание любому атрибуту   
 # [obj.name=value]  
def \_\_delattr\_\_(self, name): # Удаление любого атрибута [del obj.name]

\_\_getattr\_\_ и \_\_getattribute\_\_   
1061  
Во всех этих случаях в аргументе self, как обычно, передается подразумевае-  
мый объект экземпляра, в аргументе name – строка с именем атрибута, к которо-  
му выполняется доступ, а в аргументе value – объект, присваиваемый атрибуту.   
Два метода get обычно возвращают значение атрибута, а два других – значение   
None. Например, чтобы перехватить попытки обращения к любым атрибутам,   
можно использовать любой из двух первых методов, а чтобы перехватить по-  
пытки присваивания любым атрибутам, можно использовать третий метод:  
class Catcher:  
 def \_\_getattr\_\_(self, name):  
 print(‘Get:’, name)  
 def \_\_setattr\_\_(self, name, value):  
 print(‘Set:’, name, value)  
   
X = Catcher()  
X.job # Выведет “Get: job”  
X.pay # Выведет “Get: pay”  
X.pay = 99 # Выведет “Set: pay 99”  
Такой способ может использоваться для реализации шаблона делегирования,   
с которым мы познакомились в главе 30. Поскольку все операции обращения   
к атрибутам переадресуются нашим методам, мы можем проверять их допу-  
стимость и делегировать их выполнение встроенным объектам. Следующий   
класс (заимствованный из главы 30), например, сообщает обо всех попытках   
чтения атрибутов другого объекта, которые производятся относительно экзем-  
пляра класса обертки:  
class Wrapper:  
 def \_\_init\_\_(self, object):  
 self.wrapped = object # Сохранить объект  
 def \_\_getattr\_\_(self, attrname):  
 print(‘Trace:’, attrname) # Сообщить о попытке чтения  
 return getattr(self.wrapped, attrname) # Делегировать операцию чтения  
На основе свойств и дескрипторов невозможно реализовать аналогичное реше-  
ние, разве только написать методы доступа для всех возможных атрибутов во   
всех возможных встраиваемых объектах.  
Предотвращение зацикливаний   
в методах управления доступом к атрибутам  
Эти методы достаточно просты в использовании – единственная сложность   
заключается в возможности попасть в бесконечный цикл рекурсивных вызо-  
вов. Метод \_\_getattr\_\_ вызывается только при попытке обратиться к неопреде-  
ленному атрибуту, поэтому он может без всякой опаски обращаться к другим   
атрибутам. Однако методы \_\_getattribute\_\_ и \_\_setattr\_\_ вызываются при об-  
ращении к любым атрибутам, поэтому внутри них следует проявлять осторож-  
ность, когда возникает необходимость обратиться к другим атрибутам, чтобы   
избежать повторного вызова этого же метода и попадания в бесконечный цикл   
рекурсивных вызовов.  
Например, если внутри метода \_\_getattribute\_\_ обратиться к другому атрибуту,   
эта операция произведет вызов метода \_\_getattribute\_\_ и программный код нач-  
нет выполнять бесконечный цикл, пока не будет исчерпана доступная память:

1062   
Глава 37. Управляемые атрибуты   
def \_\_getattribute\_\_(self, name):  
 x = self.other # ЦИКЛ!  
Чтобы решить эту проблему, следует делегировать выполнение этой операции   
вышестоящему суперклассу – для исключения возможности вызова метода на   
данном уровне. Класс object является суперклассом для любого класса и пре-  
красно подходит на эту роль:  
def \_\_getattribute\_\_(self, name):  
 x = object.\_\_getattribute\_\_(self, ‘other’) # Принудительный вызов метода   
 # суперкласса  
В случае с методом \_\_setattr\_\_ складывается похожая ситуация – операция   
присваивания любому атрибуту внутри этого метода снова приводит к вызову   
метода \_\_setattr\_\_ и к попаданию в бесконечный цикл:  
def \_\_setattr\_\_(self, name, value):  
 self.other = value # ЦИКЛ!  
Чтобы решить эту проблему, присваивание должно выполняться как запись   
значения в словарь \_\_dict\_\_ пространства имен по ключу с именем атрибута.   
Тем самым предотвращается выполнение прямой операции присваивания   
атрибуту:  
def \_\_setattr\_\_(self, name, value):  
 self.\_\_dict\_\_[‘other’] = value # С использованием словаря атрибутов  
Кроме того, чтобы избежать зацикливания, метод \_\_setattr\_\_ может делеги-  
ровать выполнение операции присваивания вышестоящему суперклассу, как   
в случае с методом \_\_getattribute\_\_, однако такой подход редко используется   
на практике:  
def \_\_setattr\_\_(self, name, value):  
 object.\_\_setattr\_\_(self, ‘other’, value) # Принудительный вызов метода  
 # суперкласса  
При этом в методе \_\_getattribute\_\_ мы не можем использовать трюк со словарем   
\_\_dict\_\_, чтобы избежать зацикливания:  
def \_\_getattribute\_\_(self, name):  
 x = self.\_\_dict\_\_[‘other’] # ЦИКЛ!  
Попытка обратиться к атрибуту \_\_dict\_\_ также приводит к вызову метода \_\_ge-  
ge-  
tattribute\_\_, что опять вызывает попадание в бесконечный цикл. Странно, но   
верно!  
Метод \_\_delattr\_\_ редко используется на практике, но если используется, то   
будет вызываться при удалении любого атрибута (так же, как метод \_\_setattr\_\_   
вызывается при попытке присвоить значение любому атрибуту). Поэтому вы   
также должны принять все меры, чтобы предотвратить попадание в бесконеч-  
ный цикл при удалении атрибутов, используя тот же самый прием: выполнить   
операцию над словарем пространства имен или вызвать метод суперкласса.  
Первый пример  
На самом деле все оказывается не так сложно, как могло вам показаться при   
чтении предыдущего раздела. Чтобы увидеть, как применить на практике все   
предложенные идеи, ниже приводится тот же самый первый пример, который

\_\_getattr\_\_ и \_\_getattribute\_\_   
1063  
мы использовали для демонстрации свойств и дескрипторов, но на этот раз   
он реализован с применением методов перегрузки операторов. Поскольку эти   
методы вызываются при операциях с любыми атрибутами, мы добавили про-  
верку имен атрибутов, чтобы отличать управляемые атрибуты – все остальные   
атрибуты обрабатываются обычным способом:  
class Person:  
 def \_\_init\_\_(self, name): # Вызывается при обращении Person()  
 self.\_name = name # Вызовет \_\_setattr\_\_!  
   
 def \_\_getattr\_\_(self, attr): # Вызывается операцией obj.undefined  
 if attr == ‘name’: # Для отсутствующего атрибута с именем name  
 print(‘fetch...’)  
 return self.\_name # Не вызывает зацикливание:   
 # существующий атрибут  
 else: # Обращение к другим несуществующим   
 raise AttributeError(attr) # атрибутам вызывает ошибку  
   
 def \_\_setattr\_\_(self, attr, value): # Вызывается операцией obj.any = value  
 if attr == ‘name’:  
 print(‘change...’)  
 attr = ‘\_name’ # Внутреннее имя атрибута  
 self.\_\_dict\_\_[attr] = value # Предотвратить зацикливание  
   
 def \_\_delattr\_\_(self, attr): # Вызывается операцией del obj.any  
 if attr == ‘name’:  
 print(‘remove...’)  
 attr = ‘\_name’ # Предотвратить зацикливание,  
 del self.\_\_dict\_\_[attr] # но менее обычным способом  
   
bob = Person(‘Bob Smith’) # Объект bob обладает управляемым атрибутом  
print(bob.name) # Вызовет \_\_getattr\_\_  
bob.name = ‘Robert Smith’ # Вызовет \_\_setattr\_\_  
print(bob.name)  
del bob.name # Вызовет \_\_delattr\_\_  
   
print(‘-’\*20)  
sue = Person(‘Sue Jones’) # Объект sue также наследует свойство  
print(sue.name)  
#print(Person.name.\_\_doc\_\_) # Не имеет эквивалента в данном случае  
Обратите внимание, что операция присваивания значения атрибуту в кон-  
структоре \_\_init\_\_ приводит к вызову метода \_\_setattr\_\_ – этот метод перехва-  
тывает операции присваивания любым атрибутам, даже когда они выполня-  
ются внутри класса. Если запустить этот пример, он выведет те же результаты,   
но на этот раз будет действовать обычный механизм перегрузки операторов   
в языке Python и наши методы обработки операций над атрибутами:  
fetch...  
Bob Smith  
change...  
fetch...  
Robert Smith  
remove...  
--------------------  
fetch...  
Sue Jones

1064   
Глава 37. Управляемые атрибуты   
Обратите также внимание, что в отличие от свойств и дескрипторов, здесь от-  
сутствует возможность определения документации с описанием атрибутов –   
управляемые атрибуты существуют не в виде отдельных объектов, а внутри   
программного кода, обрабатывающего операции доступа к атрибутам.  
Чтобы добиться тех же результатов с помощью метода \_\_getattribute\_\_, доста-  
точно заменить метод \_\_getattr\_\_ в примере следующим фрагментом. Так как   
этот метод перехватывает обращения к любым атрибутам, в этой версии пред-  
приняты меры, предотвращающие зацикливание, за счет делегирования вы-  
полнения операции чтения суперклассу, а кроме того, отпала необходимость   
явно возбуждать исключение при попытке обращения к неизвестному имени:  
# Замените метод \_\_getattr\_\_ следующим фрагментом  
   
def \_\_getattribute\_\_(self, attr): # Вызывается операцией [obj.any]  
 if attr == ‘name’: # Перехватывает обращения к любым именам  
 print(‘fetch...’)  
 attr = ‘\_name’ # Отображает на внутреннее имя  
 return object.\_\_getattribute\_\_(self, attr) # Предотвратить зацикливание  
Этот пример эквивалентен примерам, реализованным на основе свойств и де-  
скрипторов, но он немного надуманный и не подчеркивает в достаточной мере   
практическую ценность этих инструментов. Вследствие своей обобщенности   
методы \_\_getattr\_\_ и \_\_getattribute\_\_ наиболее часто используются для реа-  
лизации шаблона делегирования (как уже отмечалось выше), где выполнение   
операции доступа к атрибутам делегируется вложенному объекту. Везде, где   
необходимо реализовать управление единственным атрибутом, свойства и де-  
скрипторы могут предложить решение не только не хуже, а даже лучше.  
Вычисляемые атрибуты  
Как и прежде, предыдущий пример не делает ничего особенного – он просто   
выводит сообщение при попытке обратиться к атрибуту. Однако ничего не сто-  
ит реализовать динамическое вычисление значения атрибута при попытке по-  
лучить его. Как и в случае со свойствами и дескрипторами, в следующем при-  
мере создается виртуальный атрибут X, при попытке обращения к которому   
производится вычисление его значения:  
class AttrSquare:  
 def \_\_init\_\_(self, start):  
 self.value = start # Вызовет \_\_setattr\_\_!  
   
 def \_\_getattr\_\_(self, attr): # Вызывается при обращении   
 if attr == ‘X’: # к отсутствующему атрибуту  
 return self.value \*\* 2 # value – существующий атрибут  
 else:  
 raise AttributeError(attr)  
   
 def \_\_setattr\_\_(self, attr, value): # Вызывается всеми операциями   
 if attr == ‘X’: # присваивания  
 attr = ‘value’  
 self.\_\_dict\_\_[attr] = value  
   
A = AttrSquare(3) # 2 экземпляра класса с методами перегрузки операторов  
B = AttrSquare(32) # Каждый экземпляр хранит свои данные

\_\_getattr\_\_ и \_\_getattribute\_\_   
1065  
print(A.X) # 3 \*\* 2  
A.X = 4  
print(A.X) # 4 \*\* 2  
print(B.X) # 32 \*\* 2  
Если запустить этот пример, он выведет те же результаты, которые мы полу-  
чили ранее, в примерах на основе свойств и дескрипторов, но на этот раз будут   
действовать обобщенные методы обработки операций над атрибутами:  
9  
16  
1024  
Как и прежде, того же эффекта можно добиться с использованием метода \_\_ge-  
tattribute\_\_ вместо \_\_getattr\_\_. В следующем примере мы заменили метод чте-  
В следующем примере мы заменили метод чте-  
ния атрибутов на метод \_\_getattribute\_\_ и изменили метод \_\_setattr\_\_, реализу-  
ющий операцию присваивания, воспользовавшись способом предотвращения   
зацикливания, основанным на прямом вызове метода суперкласса:  
class AttrSquare:  
 def \_\_init\_\_(self, start):  
 self.value = start # Вызовет \_\_setattr\_\_!  
   
 def \_\_getattribute\_\_(self, attr): # Вызывается при обращении ко всем атр.  
 if attr == ‘X’:  
 return self.value \*\* 2 # Снова вызовет \_\_getattribute\_\_!  
 else:  
 return object.\_\_getattribute\_\_(self, attr)  
   
def \_\_setattr\_\_(self, attr, value): # Вызывается всеми операциями присваив.  
 if attr == ‘X’:  
 attr = ‘value’  
 object.\_\_setattr\_\_(self, attr, value)  
Если запустить эту версию примера, она выведет те же самые результаты. Об-  
ратите внимание, как внутри класса происходит неявный вызов методов:  
 •  
Инструкция self.value=start внутри конструктора вызовет метод \_\_setattr\_\_  
 •  
Выражение self.value внутри метода \_\_getattribute\_\_ повторно вызовет ме-  
тод \_\_getattribute\_\_  
Фактически при каждой попытке получить значение атрибута X, метод \_\_ge-  
ge-  
tattribute\_\_ вызывается дважды. Этого не происходит в версии, использующей   
метод \_\_getattr\_\_, потому что атрибут value не является неопределенным. Если   
вас волнует потеря производительности и хотелось бы избежать ее, измените   
реализацию метода \_\_getattribute\_\_ так, чтобы он обращался к суперклассу   
для получения значения атрибута value:  
def \_\_getattribute\_\_(self, attr):  
 if attr == ‘X’:  
 return object.\_\_getattribute\_\_(self, ‘value’) \*\* 2  
Конечно, эта версия все равно вызывает метод суперкласса, но здесь не проис-  
ходит рекурсивный вызов, необходимый, чтобы добраться до требуемого атри-  
бута. Добавьте вызов функции print, чтобы наглядно увидеть, как и когда вы-  
зывается этот метод.

1066   
Глава 37. Управляемые атрибуты   
Сравнение методов \_\_getattr\_\_ и \_\_getattribute\_\_  
Чтобы обобщить различия между методами \_\_getattr\_\_ и \_\_getattribute\_\_,   
в следующем примере используются оба метода для реализации доступа   
к трем атрибутам, в числе которых: attr1 – атрибут класса, attr2 – атрибут эк-  
земпляра и attr3 – виртуальный атрибут, значение которого вычисляется при   
обращении к нему:  
class GetAttr:  
 attr1 = 1  
 def \_\_init\_\_(self):  
 self.attr2 = 2  
 def \_\_getattr\_\_(self, attr): # Только для неопределенных атрибутов  
 print(‘get: ‘ + attr) # Не attr1: наследуется от класса  
 return 3 # Не attr2: хранится в экземпляре  
   
X = GetAttr()  
print(X.attr1)  
print(X.attr2)  
print(X.attr3)  
   
print(‘-’\*40)  
   
class GetAttribute(object): # (object) – необходимо только в 2.6  
 attr1 = 1  
 def \_\_init\_\_(self):  
 self.attr2 = 2  
 def \_\_getattribute\_\_(self, attr): # Вызывается всеми операциями присваивания  
 print(‘get: ‘ + attr) # Для предотвращения зацикливания   
 if attr == ‘attr3’: # используется суперкласс  
 return 3  
 else:  
 return object.\_\_getattribute\_\_(self, attr)  
   
X = GetAttribute()  
print(X.attr1)  
print(X.attr2)  
print(X.attr3)  
Если запустить этот пример, можно будет увидеть, что версия на основе метода   
\_\_getattr\_\_ перехватывает только попытки обращения к атрибуту attr3, пото-  
му что это единственный неопределенный аргумент. С другой стороны, версия   
на основе метода \_\_getattribute\_\_ перехватывает все попытки чтения значений   
атрибутов и для получения значений неуправляемых атрибутов должна вызы-  
вать метод суперкласса, чтобы избежать зацикливания:  
1  
2  
get: attr3  
3  
----------------------------------------  
get: attr1  
1  
get: attr2  
2  
get: attr3  
3

\_\_getattr\_\_ и \_\_getattribute\_\_   
1067  
Несмотря на то, что метод \_\_getattribute\_\_ перехватывает больше операций   
обращения к атрибутам, чем метод \_\_getattr\_\_, тем не менее, на практике они   
оказываются лишь вариациями на одну тему – если атрибуты физически не   
сохраняются в памяти, эти два метода дают один и тот же эффект.  
Сравнение приемов управления  
Чтобы обобщить различия между всеми четырьмя приемами управления   
атрибутами, которые мы рассмотрели в этой главе, пройдемся по более полно-  
му примеру реализации вычисляемых атрибутов, в котором используются все   
описанные приемы. В следующей версии свойства используются для управ-  
ления доступом к вычисляемым атрибутам с именами square и cube. Обратите   
внимание, что их базовые значения сохраняются в атрибутах, имена которых   
начинаются с символа подчеркивания, – благодаря этому исключается кон-  
фликты между их именами и именами свойств:  
# 2 динамически вычисляемых атрибута, реализованные с помощью свойств  
   
class Powers:  
 def \_\_init\_\_(self, square, cube):  
 self.\_square = square # \_square – базовое значение  
 self.\_cube = cube # square – имя свойства  
   
 def getSquare(self):  
 return self.\_square \*\* 2  
 def setSquare(self, value):  
 self.\_square = value  
 square = property(getSquare, setSquare)  
   
 def getCube(self):  
 return self.\_cube \*\* 3  
 cube = property(getCube)  
   
X = Powers(3, 4)  
print(X.square) # 3 \*\* 2 = 9  
print(X.cube) # 4 \*\* 3 = 64  
X.square = 5  
print(X.square) # 5 \*\* 2 = 25  
Чтобы реализовать то же самое с помощью дескрипторов, мы определили   
полноценные классы, экземпляры которых будут представлять вычисляемые   
атрибуты. Обратите внимание, что эти дескрипторы сохраняют базовые значе-  
ния в атрибутах экземпляра, поэтому, чтобы избежать конфликтов с именами   
дескрипторов, мы снова использовали имена, начинающиеся с символа под-  
черкивания (как будет показано в заключительном примере в конце этой гла-  
вы, мы могли бы избежать необходимости соблюдать это требование, сохранив   
базовые значения в самих дескрипторах):  
# То же самое, но на основе дескрипторов  
   
class DescSquare:  
 def \_\_get\_\_(self, instance, owner):  
 return instance.\_square \*\* 2  
 def \_\_set\_\_(self, instance, value):  
 instance.\_square = value

1068   
Глава 37. Управляемые атрибуты   
class DescCube:  
 def \_\_get\_\_(self, instance, owner):  
 return instance.\_cube \*\* 3  
   
class Powers: # Используйте (object) в 2.6  
 square = DescSquare()  
 cube = DescCube()  
 def \_\_init\_\_(self, square, cube):  
 self.\_square = square # “self.square = square” также будет работать,  
 self.\_cube = cube # потому что вызовет метод \_\_set\_\_ дескриптора!  
   
X = Powers(3, 4)  
print(X.square) # 3 \*\* 2 = 9  
print(X.cube) # 4 \*\* 3 = 64  
X.square = 5  
print(X.square) # 5 \*\* 2 = 25  
Для достижения тех же результатов с помощью метода \_\_getattr\_\_ нам снова   
придется сохранять базовые значения в атрибутах с именами, начинающими-  
ся с символа подчеркивания, чтобы имена управляемых атрибутов оказались   
неопределенными и при обращении к ним вызывался бы наш метод. Нам так-  
же потребуется реализовать метод \_\_setattrr\_\_, чтобы перехватывать операции   
присваивания и принять меры к предотвращению зацикливания:  
# То же самое, но на основе обобщенного метода \_\_getattr\_\_ управления доступом   
# к неопределенным атрибутам  
   
class Powers:  
 def \_\_init\_\_(self, square, cube):  
 self.\_square = square  
 self.\_cube = cube  
   
 def \_\_getattr\_\_(self, name):  
 if name == ‘square’:  
 return self.\_square \*\* 2  
 elif name == ‘cube’:  
 return self.\_cube \*\* 3  
 else:  
 raise TypeError(‘unknown attr:’ + name)  
   
 def \_\_setattr\_\_(self, name, value):  
 if name == ‘square’:  
 self.\_\_dict\_\_[‘\_square’] = value  
 else:  
 self.\_\_dict\_\_[name] = value  
   
X = Powers(3, 4)  
print(X.square) # 3 \*\* 2 = 9  
print(X.cube) # 4 \*\* 3 = 64  
X.square = 5  
print(X.square) # 5 \*\* 2 = 25  
Последний вариант, основанный на использовании метода \_\_getattribute\_\_, по-  
хож на предыдущую версию. Поскольку теперь мы перехватываем все опера-  
ции чтения значений атрибутов, мы должны переадресовать попытки получе-  
ния базовых значений суперклассу, чтобы избежать зацикливания:

\_\_getattr\_\_ и \_\_getattribute\_\_   
1069  
# То же самое, но на основе обобщенного метода \_\_getattribute\_\_ управления   
# доступом ко всем атрибутам  
   
class Powers:  
 def \_\_init\_\_(self, square, cube):  
 self.\_square = square  
 self.\_cube = cube  
   
 def \_\_getattribute\_\_(self, name):  
 if name == ‘square’:  
 return object.\_\_getattribute\_\_(self, ‘\_square’) \*\* 2  
 elif name == ‘cube’:  
 return object.\_\_getattribute\_\_(self, ‘\_cube’) \*\* 3  
 else:  
 return object.\_\_getattribute\_\_(self, name)  
   
 def \_\_setattr\_\_(self, name, value):  
 if name == ‘square’:  
 self.\_\_dict\_\_[‘\_square’] = value  
 else:  
 self.\_\_dict\_\_[name] = value  
   
X = Powers(3, 4)  
print(X.square) # 3 \*\* 2 = 9  
print(X.cube) # 4 \*\* 3 = 64  
X.square = 5  
print(X.square) # 5 \*\* 2 = 25  
Как видите, реализации всех приемов выглядят по-разному, но все они воспро-  
изводят одни и те же результаты:  
9  
64  
25  
Сравнить эти альтернативы еще раз и увидеть другие способы реализации   
вы сможете в примере более реалистичного приложения, в разделе «Пример:   
проверка атрибутов», ниже. Но сначала нам необходимо познакомиться с ло-  
вушками, которые поджидают программистов, использующих два последних   
приема.  
Управление атрибутами встроенных операций  
Когда я представлял методы \_\_getattr\_\_ и \_\_getattribute\_\_, я отмечал, что они   
перехватывают операции обращения к неопределенным и ко всем атрибутам   
соответственно, что делает их идеальными средствами реализации шаблона де-  
легирования. Все вышесказанное справедливо для обычных атрибутов, однако   
поведение этих методов требует дополнительных пояснений: при обращении   
к атрибутам-методам, которые неявно вызываются встроенными операциями,   
эти два метода могут вообще не вызываться. То есть вызов метода перегрузки   
оператора невозможно делегировать обернутому объекту, если класс-обертка   
сам переопределяет их.  
Например, операции обращения к атрибутам методов \_\_str\_\_, \_\_add\_\_ и \_\_get-  
get-  
item\_\_, которые неявно выполняются операциями вывода, оператором + и опе-  
рацией индексирования соответственно, в Python 3.0 не переадресуются ме-

1070   
Глава 37. Управляемые атрибуты   
тодам \_\_getattr\_\_ и \_\_getattribute\_\_ управления атрибутами. Если говорить   
точнее:  
 •  
Для таких атрибутов в Python 3.0 не вызывается ни один из методов \_\_ge-  
ge-  
tattr\_\_ и \_\_getattribute\_\_.  
 •  
В Python 2.6 метод \_\_getattr\_\_ вызывается для таких атрибутов, если они не   
определены в классе.  
 •  
В Python 2.6 метод \_\_getattribute\_\_ доступен только в классах нового стиля   
и действует точно так же, как в Python 3.0.  
Другими словами, классы в Python 3.0 (и классы нового стиля в Python 2.6) не   
имеют прямой возможности перехватывать обращения к реализациям встро-  
енных операций, таких как вывод или сложение. В Python 2.X поиск методов   
таких операций начинается в экземплярах, как и всех остальных атрибутов,   
а в Python 3.0 поиск этих методов начинается в классах, минуя экземпляры.  
Это изменение усложняет реализацию шаблона делегирования в Python 3.0,   
поскольку в этой версии нет универсального способа перехватить обращения   
к методам перегрузки операторов и переадресовать их встроенному объекту.   
Но это не является непреодолимым препятствием – класс-обертка может обой-  
ти это ограничение, определив все необходимые методы перегрузки операторов   
и реализовав в них делегирование вызовов вложенному объекту. Эти допол-  
нительные методы могут быть добавлены вручную, с помощью инструментов,   
или за счет определения их в общих суперклассах и наследования их оттуда.   
Однако это делает реализацию обертки сложнее, чем раньше, когда методы   
перегрузки операторов являлись частью интерфейса обернутого объекта.  
Имейте в виду, что эта проблема касается только методов \_\_getattr\_\_ и \_\_getat-  
getat-  
tribute\_\_. Свойства и дескрипторы могут определяться только для конкретных   
атрибутов, поэтому в действительности они вообще не могут использоваться   
для реализации шаблона делегирования – единственное свойство или дескрип-  
тор не может использоваться для управления доступом к произвольным атри-  
бутам. Кроме того, класс, в котором одновременно определяются и методы пере-  
грузки операторов, и методы управления доступом к атрибутам, будет работать   
корректно независимо от типа управляемого атрибута. Здесь нас интересуют   
лишь классы, которые не переопределяют методы перегрузки операторов, но   
в которых требуется перехватить обращения к ним обобщенным способом.  
Рассмотрим следующий пример, находящийся в файле getattr.py, который   
проверяет типы различных атрибутов и встроенных операций в экземплярах   
классов, содержащих методы \_\_getattr\_\_ и \_\_getattribute\_\_:  
class GetAttr:  
 eggs = 88 # eggs – атрибут класса, spam – атрибут экземпляра  
 def \_\_init\_\_(self):  
 self.spam = 77  
 def \_\_len\_\_(self): # Реализация операции len, иначе \_\_getattr\_\_   
 print(‘\_\_len\_\_: 42’) # будет вызываться с именем \_\_len\_\_  
 return 42  
 def \_\_getattr\_\_(self, attr): # Возвращает реализацию \_\_str\_\_ по запросу,   
 print(‘getattr: ‘ + attr) # иначе – функцию-заглушку  
 if attr == ‘\_\_str\_\_’:  
 return lambda \*args: ‘[Getattr str]’  
 else:  
 return lambda \*args: None

\_\_getattr\_\_ и \_\_getattribute\_\_   
1071  
class GetAttribute(object): # object – требуется в 2.6, подразумевается в 3.0  
 eggs = 88 # В 2.6 для всех классов автоматически   
 def \_\_init\_\_(self): # выполняется условие isinstance(object)  
 self.spam = 77 # Но мы вынуждены наследовать object, чтобы  
 def \_\_len\_\_(self): # обрести инструменты, присущие классам нового  
 print(‘\_\_len\_\_: 42’) # стиля, включая \_\_getattribute\_\_, и некоторые   
 return 42 # атрибуты по умолчанию, с именами вида \_\_X\_\_  
 def \_\_getattribute\_\_(self, attr):  
 print(‘getattribute: ‘ + attr)  
 if attr == ‘\_\_str\_\_’:  
 return lambda \*args: ‘[GetAttribute str]’  
 else:  
 return lambda \*args: None  
   
for Class in GetAttr, GetAttribute:  
 print(‘\n’ + Class.\_\_name\_\_.ljust(50, ‘=’))  
   
 X = Class()  
 X.eggs # Атрибут класса  
 X.spam # Атрибут экземпляра  
 X.other # Отсутствующий атрибут  
 len(X) # Метод \_\_len\_\_ определен явно  
   
 try: # Классы нового стиля должны поддерживать [], +, вызов:   
 X[0] # \_\_getitem\_\_?  
 except:  
 print(‘fail []’)  
   
 try:  
 X + 99 # \_\_add\_\_?  
 except:  
 print(‘fail +’)  
   
 try:  
 X() # \_\_call\_\_? (неявный вызов, встроенная операция)  
 except:  
 print(‘fail ()’)  
 X.\_\_call\_\_() # \_\_call\_\_? (явный вызов, нет унаследованного метода)  
   
 print(X.\_\_str\_\_()) # \_\_str\_\_? (явный вызов, унаследован от type)  
 print(X) # \_\_str\_\_? (неявный вызов, встроенная операция)  
Если запустить этот пример под управлением Python 2.6, метод \_\_getattr\_\_ бу-  
дет вызван множество раз различными неявными попытками получения зна-  
чения атрибутов встроенными операциями, потому что интерпретатор будет   
пытаться отыскать эти атрибуты, начиная поиск с экземпляра. Метод \_\_getat-  
getat-  
tribute\_\_, напротив, не будет вызван ни разу для получения значений имен,   
соответствующих методам перегрузки операторов, потому что поиск таких   
имен производится, начиная с класса:  
C:\misc> c:\python26\python getattr.py  
   
GetAttr===========================================  
getattr: other  
\_\_len\_\_: 42  
getattr: \_\_getitem\_\_  
getattr: \_\_coerce\_\_  
getattr: \_\_add\_\_

1072   
Глава 37. Управляемые атрибуты   
getattr: \_\_call\_\_  
getattr: \_\_call\_\_  
getattr: \_\_str\_\_  
[Getattr str]  
getattr: \_\_str\_\_  
[Getattr str]  
   
GetAttribute======================================  
getattribute: eggs  
getattribute: spam  
getattribute: other  
\_\_len\_\_: 42  
fail []  
fail +  
fail ()  
getattribute: \_\_call\_\_  
getattribute: \_\_str\_\_  
[GetAttribute str]  
<\_\_main\_\_.GetAttribute object at 0x025EA1D0>  
Обратите внимание, что в версии 2.6 метод \_\_getattr\_\_ перехватывает явные   
и неявные попытки обращения к атрибутам \_\_call\_\_ и \_\_str\_\_. Метод \_\_getat-  
getat-  
tribute\_\_, напротив, не вызывается неявными попытками обращения к любым   
атрибутам, которые выполняются встроенными операцияим.  
Ситуация с методом \_\_getattribute\_\_ совершенно одинаковая в версиях 2.6   
и 3.0, потому что в версии 2.6, чтобы классы могли использовать этот метод,   
они должны унаследовать его от класса object, то есть быть классами нового   
стиля. Явное наследование класса object в этом примере необязательно в Py-  
Py-  
thon 3.0, так как в этой версии все классы являются классами нового стиля.  
Если запустить этот пример под управлением Python 3.0, результаты исполь-  
Python 3.0, результаты исполь-  
 3.0, результаты исполь-  
зования метода \_\_getattr\_\_ получатся иными – ни один из неявных вызов мето-  
дов перегрузки операторов, производимых встроенными операциями, не будет   
перехвачен ни одним из методов управления доступом к атрибутам. При раз-  
решении имен специальных методов Python 3.0 пропускает этап поиска в эк-  
Python 3.0 пропускает этап поиска в эк-  
 3.0 пропускает этап поиска в эк-  
земплярах:  
C:\misc> c:\python30\python getattr.py  
   
GetAttr===========================================  
getattr: other  
\_\_len\_\_: 42  
fail []  
fail +  
fail ()  
getattr: \_\_call\_\_  
<\_\_main\_\_.GetAttr object at 0x025D17F0>  
<\_\_main\_\_.GetAttr object at 0x025D17F0>  
   
GetAttribute======================================  
getattribute: eggs  
getattribute: spam  
getattribute: other  
\_\_len\_\_: 42  
fail []  
fail +

\_\_getattr\_\_ и \_\_getattribute\_\_   
1073  
fail ()  
getattribute: \_\_call\_\_  
getattribute: \_\_str\_\_  
[GetAttribute str]  
<\_\_main\_\_.GetAttribute object at 0x025D1870>  
По полученным результатам мы можем проследить, как работает этот пример:  
 •  
В версии 3.0 обе попытки перехватить обращение к методу \_\_str\_\_ с помо-  
щью метода \_\_getattr\_\_ потерпели неудачу� одно из обращений выполняет-  
ся встроенной функцией print, а второе – явным вызовом метода, потому   
что по умолчанию он наследуется из класса (в действительности, из встро-  
енного класса object, который является суперклассом для любых классов).  
 •  
Метод \_\_getattribute\_\_ сумел перехватить только одну попытку обратиться   
к методу \_\_str\_\_ – явный вызов, который выполняется в обход унаследо-  
ванной версии. Перехватить неявную попытку, выполняемую встроенной   
функцией print, не удалось.  
 •  
В версии 3.0 ни один из методов не смог перехватить обращение к методу \_\_  
call\_\_, которое выполняется встроенной операцией вызова, но оба перехва-  
тили явный вызов – в отличие от метода \_\_str\_\_, данные объекты не имеют   
унаследованного метода \_\_call\_\_, что могло бы повлечь неудачу с методом   
\_\_getattr\_\_.  
 •  
Вызов метода \_\_len\_\_ был перехвачен обоими классами просто потому, что   
он явно определен в классах – если удалить метод \_\_len\_\_ из классов, при   
обращении к нему в версии 3.0 не будет вызываться ни метод \_\_getattr\_\_, ни   
метод \_\_getattribute\_\_.  
 •  
Попытки перехватить все остальные встроенные операции в версии 3.0 не   
увенчались успехом.  
Результаты показывают, что в Python 3.0 при неявном вызове методов пере-  
Python 3.0 при неявном вызове методов пере-  
 3.0 при неявном вызове методов пере-  
грузки операторов встроенными операциями ни один из методов управления   
доступом к атрибутам не участвует в этом процессе: поиск таких атрибутов ин-  
терпретатор выполняет в классах, полностью пропуская этап поиска в экзем-  
плярах.  
Это обстоятельство усложняет реализацию классов-оберток в шаблоне деле-  
гирования в версии 3.0 – если обертываемый класс может содержать методы   
перегрузки операторов, эти методы должны быть переопределены в классе-  
обертке и должны делегировать выполнение операций методам обернутого   
объекта. Вообще, при таком подходе может потребоваться создать множество   
дополнительных методов, чтобы реализовать шаблон делегирования.  
Конечно, процедура добавления таких методов может быть частично автома-  
тизирована с помощью инструментов, позволяющих добавлять новые методы   
в классы (здесь могут пригодиться декораторы классов и метаклассы, которые   
рассматриваются в двух следующих главах). Кроме того, все эти дополнитель-  
ные методы могут быть определены в суперклассе и наследоваться классами,   
опирающимися на прием делегирования. И все равно, реализация шаблона де-  
легирования в версии 3.0 требует от нас дополнительной работы.  
Более реалистичную демонстрацию этой проблемы, а также способы ее реше-  
ния вы найдете в примере реализации декоратора Private, в следующей главе.   
Там мы увидим, что имеется возможность добавить в клиентский класс метод

1074   
Глава 37. Управляемые атрибуты   
\_\_getattribute\_\_, чтобы сохранить его оригинальный тип, хотя этот метод по-  
прежнему не будет вызываться при обращении к методам перегрузки операто-  
ров – операция вывода, например, напрямую обращается к методу \_\_str\_\_ в та-  
ких классах, вместо того чтобы направить запрос через метод \_\_getattribute\_\_.  
В качестве еще одного примера в следующем разделе повторно рассматривает-  
ся учебный пример из главы 27. Теперь, когда вы понимаете, как выполняется   
управление атрибутами, я смогу объяснить одну из малопонятных его особен-  
ностей.  
Пример того, как это изменение в версии 3.0 отразилось на рабо-  
те самого интерпретатора, вы найдете в обсуждении реализации   
объекта os.popen в версии 3.0 в главе 14. Поскольку в Python 3.0   
он реализован с применением класса-обертки, в котором для де-  
легирования обращений к атрибутам вложенного объекта ис-  
пользуется метод \_\_getattr\_\_, он не может перехватывать вызо-  
вы встроенной функции next(X), которая вызывает метод \_\_  
next\_\_. Однако он перехватывает и делегирует явные вызовы   
метода X.\_\_next\_\_(), потому что они не наследуются от супер-  
класса, как метод \_\_str\_\_, и такие вызовы выполняются не   
встроенными операциями.  
Такое поведение похоже на то, как обрабатываются обращения   
к методу \_\_call\_\_ в нашем примере, – неявные обращения, вы-  
полняемые встроенными функциями, не приводят к вызову ме-  
тода \_\_getattr\_\_, тогда как явные вызовы методов, которые не   
были унаследованы от типа класса, перехватываются. Другими   
словами, это изменение повлияло не только на наши классы-  
обертки, но даже на классы в стандартной библиотеке Python!   
Учитывая размах влияния этого изменения, есть вероятность,   
что в будущем оно будет пересмотрено, поэтому не забывайте   
проверять наличие этой проблемы в следующих версиях.  
Повторный обзор примера   
реализации шаблона делегирования  
В учебнике по объектно-ориентированному программированию в главе 27 был   
представлен класс Manager, использующий прием встраивания объектов и де-  
легирования вызовов методов, адаптирующих поведение суперкласса, вместо   
приема наследования. Ниже для справки приводится реализация этого приме-  
ра, из которого была удалена часть программного кода самопроверки:  
class Person:  
 def \_\_init\_\_(self, name, job=None, pay=0):  
 self.name = name  
 self.job = job  
 self.pay = pay  
 def lastName(self):  
 return self.name.split()[-1]  
 def giveRaise(self, percent):  
 self.pay = int(self.pay \* (1 + percent))  
 def \_\_str\_\_(self):  
 return ‘[Person: %s, %s]’ % (self.name, self.pay)

\_\_getattr\_\_ и \_\_getattribute\_\_   
1075  
class Manager:  
 def \_\_init\_\_(self, name, pay):  
 self.person = Person(name, ‘mgr’, pay) # Встроенный объект Person  
 def giveRaise(self, percent, bonus=.10):  
 self.person.giveRaise(percent + bonus) # Перехватывает и делегирует  
 def \_\_getattr\_\_(self, attr):  
 return getattr(self.person, attr) # Делегирует остальные атрибуты  
 def \_\_str\_\_(self):  
 return str(self.person) # Необходимо переопределить (в 3.0)  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 sue = Person(‘Sue Jones’, job=’dev’, pay=100000)  
 print(sue.lastName())  
 sue.giveRaise(.10)  
 print(sue)  
 tom = Manager(‘Tom Jones’, 50000) # Manager.\_\_init\_\_  
 print(tom.lastName()) # Manager.\_\_getattr\_\_ -> Person.lastName  
 tom.giveRaise(.10) # Manager.giveRaise -> Person.giveRaise  
 print(tom) # Manager.\_\_str\_\_ -> Person.\_\_str\_\_  
Комментарии в конце этого фрагмента поясняют, какие методы вызываются   
данными операциями. В частности, обратите внимание, что метод lastName не   
определен в классе Manager, и поэтому при обращении к нему вызывается обоб-  
щенный метод \_\_getattr\_\_, который в свою очередь вызывает метод встроенно-  
го объекта класса Person. Ниже приводятся результаты, полученные в процессе   
выполнения этого сценария – Сью (Sue) получила 10% надбавку, как объект   
класса Person, а Том (Tom) получил 20% надбавку, потому что метод giveRaise   
был адаптирован в классе Manager:  
C:\misc> c:\python30\python person.py  
Jones  
[Person: Sue Jones, 110000]  
Jones  
[Person: Tom Jones, 60000]  
Но, обратите внимание, что происходит, когда мы выводим объект класса Ma-  
nager в конце сценария: функция print вызывает метод \_\_str\_\_ класса-обертки,   
который делегирует выполнение операции методу \_\_str\_\_ встроенного объекта   
класса Person. Помня об этом, посмотрите, что получится, если мы удалим ме-  
тод Manager.\_\_str\_\_:  
# Удален метод Manager.\_\_str\_\_  
   
class Manager:  
 def \_\_init\_\_(self, name, pay):  
 self.person = Person(name, ‘mgr’, pay) # Встроенный объект Person  
 def giveRaise(self, percent, bonus=.10):  
 self.person.giveRaise(percent + bonus) # Перехватывает и делегирует  
 def \_\_getattr\_\_(self, attr):  
 return getattr(self.person, attr) # Делегирует остальные атрибуты  
В Python 3.0 при выполнении операции вывода объектов класса Manager не про-  
исходит обращения к обобщенному методу \_\_getattr\_\_ управления доступом   
к атрибутам. Вместо этого вызывается метод \_\_str\_\_ по умолчанию, унасле-  
дованный от неявного суперкласса object (объект sue по-прежнему выводится   
корректно, потому что класс Person явно определяет метод \_\_str\_\_):

1076   
Глава 37. Управляемые атрибуты   
C:\misc> c:\python30\python person.py  
Jones  
[Person: Sue Jones, 110000]  
Jones  
<\_\_main\_\_.Manager object at 0x02A5AE30>  
Интересно, что если запустить версию примера без метода \_\_str\_\_ под управ-  
лением Python 2.6, попытка вывода объекта tom приведет к вызову метода \_\_  
getattr\_\_, потому что классические классы не имеют унаследованного метода   
\_\_str\_\_, а поиск неопределенных атрибутов, соответствующих методам пере-  
грузки операторов, производится через обращение к методу \_\_getattr\_\_:  
C:\misc> c:\python26\python person.py  
Jones  
[Person: Sue Jones, 110000]  
Jones  
[Person: Tom Jones, 60000]  
Переход на использование метода \_\_getattribute\_\_ не исправит ситуацию – как   
и метод \_\_getattr\_\_, он не вызывается, когда встроенная операция пытается   
обратиться к соответствующему ей методу перегрузки операторов ни в Py-  
Py-  
thon 2.6, ни в Python 3.0:  
# Замена \_\_getattr\_ на \_\_getattribute\_\_  
   
class Manager: # Используйте (object) в 2.6  
 def \_\_init\_\_(self, name, pay):  
 self.person = Person(name, ‘mgr’, pay) # Встроенный объект Person  
 def giveRaise(self, percent, bonus=.10):  
 self.person.giveRaise(percent + bonus) # Перехватывает и делегирует  
 def \_\_getattribute\_\_(self, attr):  
 print(‘\*\*’, attr)  
 if attr in [‘person’, ‘giveRaise’]:  
 return object.\_\_getattribute\_\_(self, attr) # Возвращает свой атр.  
 else:  
 return getattr(self.person, attr) # Делегирует остальные атрибуты  
Независимо от того, какой метод управления доступом к атрибутам будет ис-  
пользоваться в версии 3.0, мы все равно должны включить в класс Manager пе-  
реопределенную версию метода \_\_str\_\_ (как показано выше), чтобы перехва-  
тывать операции вывода и делегировать их выполнение встроенному объекту   
класса Person:  
C:\misc> c:\python30\python person.py  
Jones  
[Person: Sue Jones, 110000]  
\*\* lastName  
\*\* person  
Jones  
\*\* giveRaise  
\*\* person  
<\_\_main\_\_.Manager object at 0x028E0590>  
Обратите внимание, что метод \_\_getattribute\_\_ вызывается дважды, когда про-  
изводится попытка получить имя метода: первый раз – с именем метода и вто-  
рой раз – чтобы получить атрибут встроенного объекта self.person. Мы могли

\_\_getattr\_\_ и \_\_getattribute\_\_   
1077  
бы избавиться от такого повторения, прибегнув к различным ухищрениям, но   
мы по-прежнему должны переопределить метод \_\_str\_\_, чтобы перехватывать   
операцию вывода, хотя в следующем фрагменте она реализована иначе (ис-  
пользование self.person привело бы к неудаче в этой реализации метода \_\_ge-  
ge-  
tattribute\_\_):  
# Иная реализация \_\_getattribute\_\_ с целью избавиться от лишних вызовов  
   
class Manager:  
 def \_\_init\_\_(self, name, pay):  
 self.person = Person(name, ‘mgr’, pay)  
 def \_\_getattribute\_\_(self, attr):  
 print(‘\*\*’, attr)  
 person = object.\_\_getattribute\_\_(self, ‘person’)  
 if attr == ‘giveRaise’:  
 return lambda percent: person.giveRaise(percent+.10)  
 else:  
 return getattr(person, attr)  
 def \_\_str\_\_(self):  
 person = object.\_\_getattribute\_\_(self, ‘person’)  
 return str(person)  
Если запустить эту альтернативу, наш объект будет выводиться корректно, но   
только потому, что мы явно добавили метод \_\_str\_\_ в класс-обертку – при об-  
ращении к этому атрибуту из встроенных операций, обобщенный метод управ-  
ления доступом к атрибутам по-прежнему не вызывается:  
Jones  
[Person: Sue Jones, 110000]  
\*\* lastName  
Jones  
\*\* giveRaise  
[Person: Tom Jones, 60000]  
Из всего вышеизложенного следует вывод, что при реализации шаблона де-  
легирования в Python 3.0 классы, такие как Manager, должны переопределять   
некоторые методы перегрузки операторов (например, \_\_str\_\_), чтобы деле-  
гировать выполнение соответствующих им операций встроенным объектам.   
В Python 2.6 этого не требуется, если не используются классы нового стиля.   
Похоже, что нашими единственными очевидными вариантами являются ис-  
пользование метода \_\_getattr\_\_ и Python 2.6 или избыточное переопределение   
методов перегрузки операторов в классах-обертках в Python 3.0.  
Замечу еще раз, что все вышеизложенное не является неразрешимой пробле-  
мой – при создании большинства классов-оберток можно заранее определить,   
какие методы перегрузки операторов потребуются, а применение дополнитель-  
ных инструментов и создание суперклассов помогут частично автоматизиро-  
вать эту задачу. Кроме того, не во всех классах используются методы перегруз-  
ки операторов (в действительности, в большинстве прикладных классов этого   
не требуется). Однако об этих особенностях реализации шаблона делегирова-  
ния в Python 3.0 следует помнить – когда методы перегрузки операторов яв-  
Python 3.0 следует помнить – когда методы перегрузки операторов яв-  
 3.0 следует помнить – когда методы перегрузки операторов яв-  
ляются частью интерфейса объектов, обертки должны учитывать это обстоя-  
тельство и обеспечивать переносимую реализацию, переопределяя эти методы   
локально.

1078   
Глава 37. Управляемые атрибуты   
Пример: проверка атрибутов  
В заключение этой главы обратимся к более реалистичному примеру, в кото-  
ром реализованы все четыре схемы управления атрибутами. В этом примере   
определяется класс CardHolder с четырьмя атрибутами, три из которых явля-  
ются управляемыми. Для управляемых атрибутов реализована проверка или   
преобразование значений при присваивании или при чтении. Во всех четырех   
версиях присутствует один и тот же программный код самопроверки, который   
воспроизводит одинаковые результаты, но само управление атрибутами реа-  
лизовано разными способами. Примеры предназначены в основном для само-  
стоятельного изучения – я не буду углубляться в подробности при их рассмо-  
трении, но отмечу, что во всех примерах используются понятия, которые мы   
уже исследовали в этой главе.  
Использование свойств для проверки  
Наш первый пример реализует управление атрибутами с помощью свойств.   
Как обычно, вместо управляемых атрибутов мы могли бы использовать обыч-  
ные методы, но свойства помогут нам, если атрибуты уже используются где-то   
в существующем программном коде. Свойства позволяют автоматически вы-  
зывать программный код при обращении к ним, но они могут применяться   
только к отдельным атрибутам – свойства не позволяют перехватывать обра-  
щения к атрибутам обобщенным способом.  
Для понимания этого примера крайне важно отметить, что операции присваи-  
вания значений атрибутам в конструкторе \_\_init\_\_ также вызывают методы   
записи свойств. Например, когда в конструкторе выполняется присваивание   
значения атрибуту self.name, эта операция вызывает метод setName, который   
преобразует значение и сохраняет его в атрибуте экземпляра с именем \_\_name,   
благодаря чему исключается конфликт с именем свойства.  
Такое переименование (иногда называется искажением имен) необходимо по-  
тому, что свойства не имеют своих данных и используют данные экземпляра.   
Фактические данные всегда хранятся в атрибуте с именем \_\_name, а атрибут   
с именем name – это свойство, а не данные.  
Наконец, этот класс реализует управление атрибутами с именами name, age   
и acct, обеспечивает прямой доступ к атрибуту addr и реализует атрибут remain,   
доступный только для чтения, который фактически является виртуальным   
атрибутом, значение которого вычисляется по требованию. Для сравнения, эта   
версия, основанная на применении свойств, содержит 39 строк программного   
кода:  
class CardHolder:  
 acctlen = 8 # Данные класса  
 retireage = 59.5  
   
 def \_\_init\_\_(self, acct, name, age, addr):  
 self.acct = acct # Данные экземпляра  
 self.name = name # Эти операции вызывают методы записи свойств  
 self.age = age # К именам вида \_\_X добавляется имя класса  
 self.addr = addr # addr – неуправляемый атрибут  
 # remain – не имеет фактических данных  
 def getName(self):  
 return self.\_\_name

Пример: проверка атрибутов   
1079  
 def setName(self, value):  
 value = value.lower().replace(‘ ‘, ‘\_’)  
 self.\_\_name = value  
 name = property(getName, setName)  
   
 def getAge(self):  
 return self.\_\_age  
 def setAge(self, value):  
 if value < 0 or value > 150:  
 raise ValueError(‘invalid age’)  
 else:  
 self.\_\_age = value  
 age = property(getAge, setAge)  
   
 def getAcct(self):  
 return self.\_\_acct[:-3] + ‘\*\*\*’  
 def setAcct(self, value):  
 value = value.replace(‘-’, ‘’)  
 if len(value) != self.acctlen:  
 raise TypeError(‘invald acct number’)  
 else:  
 self.\_\_acct = value  
 acct = property(getAcct, setAcct)  
   
 def remainGet(self): # Можно было бы реализовать как   
 return self.retireage - self.age # метод, если нигде не используется   
 remain = property(remainGet) # как атрибут  
Программный код самопроверки  
Ниже приводится фрагмент программного кода, тестирующего наш класс, –   
добавьте его в конец файла или поместите определение класса в модуль и им-  
портируйте его. Во всех четырех версиях этого примера мы будем использовать   
один и тот же программный код самопроверки. Он создает два экземпляра клас-  
са с управляемыми атрибутами и извлекает и изменяет значения его атрибутов.   
Операции, которые могут вызвать исключение, обернуты в инструкции try:  
bob = CardHolder(‘1234-5678’, ‘Bob Smith’, 40, ‘123 main st’)  
print(bob.acct, bob.name, bob.age, bob.remain, bob.addr, sep=’ / ‘)  
bob.name = ‘Bob Q. Smith’  
bob.age = 50  
bob.acct = ‘23-45-67-89’  
print(bob.acct, bob.name, bob.age, bob.remain, bob.addr, sep=’ / ‘)  
   
sue = CardHolder(‘5678-12-34’, ‘Sue Jones’, 35, ‘124 main st’)  
print(sue.acct, sue.name, sue.age, sue.remain, sue.addr, sep=’ / ‘)  
try:  
 sue.age = 200  
except:  
 print(‘Bad age for Sue’)  
   
try:  
 sue.remain = 5  
except:  
 print(“Can’t set sue.remain”)  
   
try:  
 sue.acct = ‘1234567’

1080   
Глава 37. Управляемые атрибуты   
except:  
 print(‘Bad acct for Sue’)  
Ниже следуют результаты, которые воспроизводятся программным кодом са-  
мопроверки. Напомню, что все четыре версии этого примера дают одни и те же   
результаты. Внимательно изучите их, чтобы понять, как вызываются методы   
класса – в учетных номерах вместо некоторых цифр отображаются звездочки,   
имена преобразуются в стандартный формат, а время, оставшееся до выхода на   
пенсию, вычисляется с использованием атрибута класса:   
12345\*\*\* / bob\_smith / 40 / 19.5 / 123 main st  
23456\*\*\* / bob\_q.\_smith / 50 / 9.5 / 123 main st  
56781\*\*\* / sue\_jones / 35 / 24.5 / 124 main st  
Bad age for Sue  
Can’t set sue.remain  
Bad acct for Sue  
Использование дескрипторов для проверки  
Теперь реализуем пример с применением дескрипторов вместо свойств. Как мы   
уже знаем, дескрипторы очень похожи на свойства в смысле функционально-  
сти и области применения – фактически свойства по сути являются ограни-  
ченной формой дескрипторов. Подобно свойствам дескрипторы предназначены   
для обработки отдельных атрибутов – они не позволяют реализовать обобщен-  
ное управление сразу всеми атрибутами. В отличие от свойств, дескрипторы   
могут хранить собственные данные и обеспечивают более универсальное ре-  
шение.  
Для понимания этого примера снова важно отметить, что операции присваи-  
вания значений атрибутам в конструкторе \_\_init\_\_ также вызывают методы \_\_  
set\_\_ дескрипторов. Например, когда в конструкторе выполняется присваива-  
ние значения атрибуту self.name, эта операция автоматически вызывает метод   
Name.\_\_set\_\_(), который преобразует значение и сохраняет его в атрибуте name   
дескриптора.  
Однако в отличие от предыдущей версии, в данном случае атрибут name с фак-  
тическим значением присоединяется к объекту дескриптора, а не к экземпляру   
клиентского класса. Конечно, мы могли бы реализовать хранение фактических   
значений и в экземпляре, однако сохранение данных в дескрипторе позволяет   
нам избежать необходимости искажать имена, добавлением символов подчер-  
кивания во избежание конфликтов. В клиентском классе CardHolder атрибут   
с именем name всегда представляет дескриптор, а не фактические данные.  
Наконец, этот класс реализует управление теми же атрибутами, что и класс   
в предыдущей версии примера: он управляет атрибутами с именами name, age   
и acct, обеспечивает прямой доступ к атрибуту addr и реализует атрибут remain,   
доступный только для чтения, который фактически является виртуальным   
атрибутом, значение которого вычисляется по требованию. Обратите внима-  
ние, что попытки выполнить присваивание атрибуту remain должны перехва-  
тываться его дескриптором и возбуждать исключение – как мы узнали выше,   
если этого не сделать, операция присваивания создаст обычный атрибут экзем-  
пляра, который отменит действие дескриптора атрибута класса. Для сравне-  
ния, эта версия, основанная на применении дескрипторов, содержит 45 строк   
программного кода:

Пример: проверка атрибутов   
1081  
class CardHolder:  
 acctlen = 8 # Данные класса  
 retireage = 59.5  
   
 def \_\_init\_\_(self, acct, name, age, addr):  
 self.acct = acct # Данные экземпляра  
 self.name = name # Эти операции вызывают методы \_\_set\_\_  
 self.age = age # Имена вида \_\_X не требуются в дескрипторах  
 self.addr = addr # addr – неуправляемый атрибут  
 # remain – не имеет фактических данных  
 class Name:  
 def \_\_get\_\_(self, instance, owner): # Класс имен: локальный для   
 return self.name # CardHolder  
 def \_\_set\_\_(self, instance, value):  
 value = value.lower().replace(‘ ‘, ‘\_’)  
 self.name = value  
 name = Name()  
   
 class Age:  
 def \_\_get\_\_(self, instance, owner):  
 return self.age # Данные дескриптора  
 def \_\_set\_\_(self, instance, value):  
 if value < 0 or value > 150:  
 raise ValueError(‘invalid age’)  
 else:  
 self.age = value  
 age = Age()  
   
 class Acct:  
 def \_\_get\_\_(self, instance, owner):  
 return self.acct[:-3] + ‘\*\*\*’  
 def \_\_set\_\_(self, instance, value):  
 value = value.replace(‘-’, ‘’)  
 if len(value) != instance.acctlen: # Данные экземпляра  
 raise TypeError(‘invald acct number’)  
 else:  
 self.acct = value  
 acct = Acct()  
   
 class Remain:  
 def \_\_get\_\_(self, instance, owner):  
 return instance.retireage - instance.age # Вызовет Age.\_\_get\_\_  
 def \_\_set\_\_(self, instance, value):  
 raise TypeError(‘cannot set remain’) # При необходимости здесь   
 remain = Remain() # можно реализовать операцию   
 # присваивания  
Использование метода \_\_getattr\_\_ для проверки  
Как мы уже знаем, метод \_\_getattr\_\_ перехватывает попытки обратиться к лю-  
бым неопределенным атрибутам, поэтому на его основе можно реализовать бо-  
лее обобщенный способ управления атрибутами, чем с применением свойств   
или дескрипторов. В нашем примере мы просто будем проверять имена атрибу-  
тов, чтобы отличать управляемые атрибуты от всех остальных, – другие атри-  
буты будут физически храниться в экземпляре и потому при попытках полу-

1082   
Глава 37. Управляемые атрибуты   
чить их значения метод \_\_getattr\_\_ вызываться не будет. Хотя такой подход яв-  
ляется более универсальным, чем с использованием свойств или дескрипторов,   
тем не менее, он может потребовать приложить больше усилий, чтобы имити-  
ровать действие других, более специализированных инструментов. Во время   
выполнения нам необходимо будет проверять имена атрибутов, и мы должны   
определить метод \_\_setattr\_\_, чтобы реализовать проверку допустимости опе-  
раций присваивания значений атрибутам.  
Как и в версиях этого примера, основанных на применении свойств и дескрип-  
торов, важно отметить, что операции присваивания значений атрибутам в кон-  
структоре \_\_init\_\_ вызывают метод \_\_setattr\_\_ класса. Например, когда в кон-  
структоре выполняется присваивание значения атрибуту self.name, эта опера-  
ция автоматически вызывает метод \_\_setattr\_\_, который преобразует значение   
и сохраняет его в атрибуте name. Сохранение имени в атрибуте name экземпляра   
гарантирует, что последующие обращения к нему не будут приводить к вызову   
метода \_\_getattr\_\_. Значение атрибута acct, напротив, сохраняется в фактиче-  
ском атрибуте \_acct, поэтому все последующие попытки обращения к нему бу-  
дут вызывать метод \_\_getattr\_\_.  
Наконец, этот класс, как и два предыдущих, реализует управление атрибу-  
тами с именами name, age и acct, обеспечивает прямой доступ к атрибуту addr   
и реализует атрибут remain, доступный только для чтения, который фактиче-  
ски является виртуальным атрибутом, значение которого вычисляется по тре-  
бованию.   
Для сравнения, эта версия содержит 32 строки программного кода – на 7 строк   
меньше, чем версия на основе свойств, и на 13 строк меньше, чем версия на   
основе дескрипторов. Безусловно, очевидность программного кода важнее его   
размера, но иногда дополнительный программный код подразумевает допол-  
нительные затраты на его разработку и сопровождение. Вероятно, здесь более   
важна роль программного кода: обобщенные инструменты, такие как метод \_\_  
getattr\_\_, лучше подходят для реализации шаблона делегирования, тогда как   
свойства и дескрипторы более предназначены для управления конкретными   
атрибутами.  
Обратите также внимание, что при присваивании значений неуправляемым   
атрибутам (таким как addr) в этой реализации выполняются дополнительные   
вызовы, но при обращении к неуправляемым атрибутам таких дополнитель-  
ных вызовов не производится, поскольку они физически существуют в экзем-  
пляре. В большинстве случаев применение свойств и дескрипторов, вероятно,   
приведет к некоторому снижению производительности, тем не менее, при их   
использовании дополнительные вызовы производятся только при попытках   
обращения к управляемым атрибутам.  
Ниже приводится версия примера, основанного на использовании метода \_\_ge-  
ge-  
tattr\_\_:  
class CardHolder:  
 acctlen = 8 # Данные класса  
 retireage = 59.5  
   
 def \_\_init\_\_(self, acct, name, age, addr):  
 self.acct = acct # Данные экземпляра  
 self.name = name # Эти операции вызывают метод \_\_setattr\_\_  
 self.age = age # \_acct не искажается: проверяемое имя

Пример: проверка атрибутов   
1083  
 self.addr = addr # addr – неуправляемый атрибут  
 # remain – не имеет фактических данных  
 def \_\_getattr\_\_(self, name):  
 if name == ‘acct’: # Вызывается для неопределенных атрибутов  
 return self.\_acct[:-3] + ‘\*\*\*’ # name, age, addr - определены  
 elif name == ‘remain’:  
 return self.retireage - self.age # Не вызывает \_\_getattr\_\_  
 else:  
 raise AttributeError(name)  
   
 def \_\_setattr\_\_(self, name, value):  
 if name == ‘name’: # Вызывается всеми операциями присваивания  
 value = value.lower().replace(‘ ‘, ‘\_’) # addr сохраняется непоср.  
 elif name == ‘age’: # acct изменено на \_acct  
 if value < 0 or value > 150:  
 raise ValueError(‘invalid age’)  
 elif name == ‘acct’:  
 name = ‘\_acct’  
 value = value.replace(‘-’, ‘’)  
 if len(value) != self.acctlen:  
 raise TypeError(‘invald acct number’)  
 elif name == ‘remain’:  
 raise TypeError(‘cannot set remain’)  
 self.\_\_dict\_\_[name] = value # Предотвращение зацикливания  
Использование метода \_\_getattribute\_\_ для проверки  
Наша заключительная версия основана на использовании метода \_\_getattrib-  
getattrib-  
ute\_\_, который перехватывает все попытки обращения к атрибутам и реализу-  
ет управление ими в случае необходимости. Этот метод перехватывает обраще-  
ния к любым атрибутам, поэтому мы будем проверять имена атрибутов, что-  
бы отличать управляемые атрибуты от всех остальных и делегировать чтение   
остальных атрибутов суперклассу. В этой версии используется точно такой же   
метод \_\_setattr\_\_, как и в предыдущей версии, который перехватывает опера-  
ции присваивания значений.  
Эта версия действует практически так же, как и предыдущая, на основе мето-  
да \_\_getattr\_\_, поэтому я не буду повторять полное ее описание. Тем не менее,   
обратите внимание, что в этой версии метод \_\_getattribute\_\_ перехватывает об-  
ращения ко всем атрибутам, поэтому здесь нам не требуется искажать имена,   
чтобы перехватить обращения к ним (атрибут acct хранится под своим именем   
acct). С другой стороны, в этой версии необходимо переадресовать операцию   
чтения значений неуправляемых атрибутов суперклассу, чтобы избежать за-  
цикливания.  
Обратите также внимание, что в этой версии выполняются дополнительные   
вызовы методов как при чтении, так и при присваивании значений неуправля-  
емым атрибутам (таким, как addr), – с точки зрения скорости выполнения, эта   
версия может оказаться самой медленной. Для сравнения, эта версия содержит   
32 строки программного кода, как и предыдущая:  
class CardHolder:  
 acctlen = 8 # Данные класса  
 retireage = 59.5  
   
 def \_\_init\_\_(self, acct, name, age, addr):

1084   
Глава 37. Управляемые атрибуты   
 self.acct = acct # Данные экземпляра  
 self.name = name # Эти операции вызывают метод \_\_setattr\_\_  
 self.age = age # acct не искажается: проверяемое имя  
 self.addr = addr # addr – неуправляемый атрибут  
 # remain – не имеет фактических данных  
 def \_\_getattribute\_\_(self, name):  
 superget = object.\_\_getattribute\_\_ # Не зацикливается: на уровень выше  
 if name == ‘acct’: # Вызывается для всех атрибутов  
 return superget(self, ‘acct’)[:-3] + ‘\*\*\*’  
 elif name == ‘remain’:  
 return superget(self, ‘retireage’) - superget(self, ‘age’)  
 else:  
 return superget(self, name) # name, age, addr: сохраняются  
   
 def \_\_setattr\_\_(self, name, value):  
 if name == ‘name’: # Вызывается всеми операциями присваивания  
 value = value.lower().replace(‘ ‘, ‘\_’) # addr сохраняется непоср.  
 elif name == ‘age’:  
 if value < 0 or value > 150:  
 raise ValueError(‘invalid age’)  
 elif name == ‘acct’:  
 value = value.replace(‘-’, ‘’)  
 if len(value) != self.acctlen:  
 raise TypeError(‘invald acct number’)  
 elif name == ‘remain’:  
 raise TypeError(‘cannot set remain’)  
 self.\_\_dict\_\_[name] = value # Предотвращение зацикливания, исх. имена  
Обязательно изучите и опробуйте примеры в этом разделе, чтобы полнее по-  
нять, как реализуется управление атрибутами.  
В заключение  
В этой главе мы рассмотрели различные способы управления доступом к атри-  
бутам в языке Python, включая методы перегрузки операторов \_\_getattr\_\_ и \_\_  
getattribute\_\_, свойства классов и дескрипторы атрибутов. Попутно мы срав-  
нили эти инструменты и рассмотрели несколько случаев, демонстрирующих   
особенности их поведения.   
В главе 38 мы продолжим обзор средств создания инструментов программи-  
ста и поближе познакомимся с декораторами, позволяющими добавлять про-  
граммный код, который выполняется автоматически на этапе создания функ-  
ций и классов, а не в момент, когда производится обращение к атрибутам. Од-  
нако прежде чем двинуться дальше, ответьте на контрольные вопросы к этой   
главе, чтобы закрепить знания полученные здесь.  
Закрепление пройденного  
Контрольные вопросы  
1. Чем отличаются методы \_\_getattr\_\_ и \_\_getattribute\_\_?  
2. Чем отличаются свойства и дескрипторы?  
3. Как связаны между собой свойства и декораторы?

Закрепление пройденного   
1085  
4. В чем заключается основное функциональное отличие методов \_\_getattr\_\_   
и \_\_getattribute\_\_ от свойств и дескрипторов?  
5. Можно ли все эти сравнения особенностей назвать своего рода спором?  
Ответы  
1. Метод \_\_getattr\_\_ вызывается только при обращении к неопределенным   
атрибутам – то есть к тем из них, которые отсутствуют в экземпляре и не   
наследуются от классов. Метод \_\_getattribute\_\_, напротив, вызывается при   
обращении к любым атрибутам независимо от того, определен атрибут или   
нет. Благодаря этому программный код внутри метода \_\_getattr\_\_ может   
безопасно обращаться к другим атрибутам, если они определены, тогда как   
в методе \_\_getattribute\_\_ необходимо предпринять специальные меры, что-  
бы избежать зацикливания (обращения к атрибутам должны переадресо-  
вываться суперклассу, чтобы избежать рекурсивного вызова самого себя).  
2. Свойства служат узкоспециализированной цели, тогда как дескрипторы   
обеспечивают более универсальное решение. Свойства позволяют опреде-  
лить функции чтения, записи и удаления для определенного атрибута – де-  
скрипторы также позволяют создавать классы с методами, реализующими   
эти операции, но они обеспечивают дополнительную гибкость, предостав-  
ляя поддержку выполнения более произвольных действий. В действитель-  
ности свойства представляют собой упрощенный способ создания дескрип-  
торов определенного типа – который вызывает функции при попытках   
доступа к атрибуту. Также они отличаются своей реализацией: свойство   
создается с помощью встроенной функции, а дескриптор определяется как   
класс. Кроме того, дескрипторы могут пользоваться всеми преимущества-  
ми ООП, такими как наследование. В дополнение к данным экземпляра де-  
скрипторы могут хранить и использовать собственные данные, что позволя-  
ет избегать конфликтов с именами атрибутов в экземплярах.   
3. Свойства могут определяться с применением синтаксиса декораторов. По-  
скольку встроенная функция property принимает единственный аргумент-  
функцию, она может использоваться как декоратор функций, чтобы опре-  
делить свойство, доступное для чтения. Из-за особенностей поведения де-  
кораторов имя декорированной функции присваивается свойству, методом   
чтения которого назначается оригинальная функция (name = property(name)).   
Наличие методов setter и deleter в свойствах позволяет в дальнейшем до-  
бавлять с применением синтаксиса декораторов методы доступа, реализу-  
ющие запись значения в атрибут и удаление атрибута, – они присваивают   
соответствующий метод доступа атрибуту декорированной функции и воз-  
вращают дополненное свойство.  
4. Методы \_\_getattr\_\_ и \_\_getattribute\_\_ позволяют получить более обобщен-  
ное решение: они могут использоваться для управления доступом к про-  
извольному количеству атрибутов. В противоположность им каждое свой-  
ство или дескриптор обеспечивает управление доступом только к одному,   
определенному атрибуту – мы не сможем перехватить обращения ко всем   
атрибутам с помощью единственного свойства или дескриптора. С другой   
стороны, свойства и дескрипторы позволяют управлять не только чтением   
значений атрибутов, но и присваиванием: методы \_\_getattr\_\_ и \_\_getattrib-  
getattrib-  
ute\_\_ обрабатывают только операции чтения – чтобы перехватить операции

1086   
Глава 37. Управляемые атрибуты   
присваивания, необходимо реализовать метод \_\_setattr\_\_. Кроме того, они   
отличаются своей реализацией: \_\_getattr\_\_ и \_\_getattribute\_\_ – это методы   
перегрузки операторов, тогда как свойства и дескрипторы – это объекты,   
которые вручную присваиваются атрибутам класса.  
5. Нет, нельзя. Цитата из сериала «Monty Python’s Flying Circus» (Летающий   
цирк Монти Пайтона):  
Спор – это последовательность взаимосвязанных утверждений с целью обосновать свою   
позицию.  
Нет, это не так.  
Да, это так! Спор – это не просто противостояние.  
Послушайте, если я спорю с вами, следовательно, я должен занять противоположную   
позицию.  
Да, но это не означает, что вы можете просто сказать: “Нет, это не так.”  
Но это так!  
Нет, не так!  
Да, так!  
Нет, не так. Спор – это интеллектуальный процесс. Противостояние – это всего лишь   
автоматическое высказывание несогласия с любым утверждением, которые делает другой   
человек.  
(короткая пауза)  
Нет, это не так.  
Это так.  
Нисколько.  
Но послушайте...

Глава 38.  
   
Декораторы  
В главе 31, где рассматривались дополнительные возможности классов, мы по-  
знакомились со статическими методами и с методами классов, а также вкратце   
рассмотрели способ их объявления с применением синтаксиса декораторов @.   
Кроме того, мы сталкивались с декораторами в предыдущей главе (глава 37),   
где узнали, что встроенная функция property может использоваться как деко-  
ратор, и в главе 28, когда изучали понятие абстрактного суперкласса.  
В этой главе мы продолжим обсуждение декораторов. Здесь мы детально рас-  
смотрим внутренние особенности декораторов и познакомимся с дополнитель-  
ными способами создания собственных декораторов. Как вы увидите далее,   
многие понятия, с которыми мы познакомились в предыдущих главах, такие   
как сохранение состояния, постоянно будут встречаться в декораторах.  
Декораторы – это достаточно сложная тема, а вопрос разработки декораторов   
представляет интерес скорее для разработчиков инструментальных средств,   
чем для прикладных программистов. Однако, учитывая, что декораторы при-  
обретают все большую популярность в различных фреймворках на языке   
Python, понимание основ декораторов поможет вам приподнять покров таин-  
, понимание основ декораторов поможет вам приподнять покров таин-  
ственности, окутывающий декораторы, даже если вы только пользуетесь ими.  
Помимо изучения особенностей конструирования декораторов в этой главе   
рассматриваются более практичные  примеры использования языка Python.   
Так как демонстрируемые здесь примеры несколько большего объема, чем где-  
либо в книге, они лучше иллюстрируют, как программный код объединяется   
в более завершенные системы и инструменты. И – в качестве бонуса – значи-  
тельная часть программного кода, который вы напишете в этой главе, может   
использоваться в качестве универсальных инструментов в ваших программах.  
Что такое декоратор?  
Декорирование – это способ управления функциями и классами. Сами декора-  
торы имеют форму вызываемых объектов (таких, как функции), которые об-  
рабатывают другие вызываемые объекты. Как мы видели ранее в этой книге,   
декораторы в языке Python имеют две родственные разновидности:

1088   
Глава 38. Декораторы   
 •  
Декораторы функций связывают имя функции с другим вызываемым объ-  
ектом на этапе определения функции, добавляя дополнительный уровень   
логики, которая управляет функциями и методами или выполняет некото-  
рые действия в случае их вызова.  
 •  
Декораторы классов связывают имя класса с другим вызываемым объек-  
том на этапе его определения, добавляя дополнительный уровень логики,   
которая управляет классами или экземплярами, созданными при обраще-  
нии к этим классам.  
В двух словах, декораторы предоставляют возможность в конце инструкции   
def определения функции в случае декораторов функций или в конце инструк-  
ции class определения класса в случае декораторов классов добавить автомати-  
чески вызываемый программный код. Этот программный код может служить   
самым разным целям, как описывается в следующих разделах.  
Управление вызовами и экземплярами  
Например, в типичном случае, автоматически запускаемый программный код   
может использоваться для выполнения дополнительных операций при вызо-  
ве функций и классов. Для достижения этой цели устанавливается объект-  
обертка, который вызывается позднее:   
 •  
Декораторы функций устанавливают объекты-обертки, перехватывающие   
вызовы функций и выполняющие необходимые операции.  
 •  
Декораторы классов устанавливают объекты-обертки, перехватывающие   
попытки создания экземпляров и выполняющие необходимые операции.  
Этот эффект достигается декораторами автоматически, за счет автоматическо-  
го связывания имен функций и классов с другими вызываемыми объектами   
в конце инструкций def и class. При последующих вызовах эти вызываемые   
объекты могут выполнять самые разные операции, такие как трассировка   
и хронометраж вызовов функций, управление доступом к атрибутам экзем-  
пляров классов и так далее.  
Управление функциями и классами  
В большей части примеров этой главы будут демонстрироваться обертки, пере-  
хватывающие вызовы функций, и операции создания экземпляров классов, но   
это не все, что могут делать декораторы:  
 •  
Декораторы  функций могут также управлять не только вызовами функ-  
ций, но и самими объектами функций, например регистрировать функции   
в некотором прикладном интерфейсе. Однако основное наше внимание бу-  
дет уделяться здесь наиболее типичному применению декораторов – управ-  
лению вызовами.  
 •  
Декораторы классов могут также использоваться не только для управле-  
ния вызовами классов с целью создания экземпляров, но и самими объек-  
тами  классов, например, добавлять новые методы в классы. Поскольку   
эта область применения декораторов в значительной степени пересекается   
с областью применения метаклассов (в действительности и декораторы,   
и метаклассы добавляют логику, которая выполняется в конце процедуры   
создания класса), дополнительные примеры такого их использования мы   
увидим в следующей главе.

Что такое декоратор?   
1089  
Другими словами, декораторы функций могут использоваться для управления   
вызовами функций и самими объектами функций, а декораторы классов могут   
использоваться для управления процедурой создания экземпляров классов   
и самих классов. Возвращая сам декорируемый объект вместо обертки, деко-  
раторы могут служить простым способом выполнения дополнительных опера-  
ций после создания функций и классов.  
Независимо от того, какую роль они играют, декораторы предоставляют про-  
стой и очевидный способ реализации инструментов, которые будут полезны и в   
процессе разработки программ, и в действующих системах.   
Определение и использование декораторов  
В зависимости от характера вашей работы, вы можете быть пользователем   
декораторов или их разработчиком. Как вы уже видели, в составе Python по-  
Python по-  
 по-  
ставляется множество встроенных декораторов, которые играют специализи-  
рованные роли, – объявление статических методов, создание свойств и многие   
другие. Кроме того, многие популярные библиотеки на языке Python включа-  
Python включа-  
 включа-  
ют декораторы, позволяющие решать такие задачи, как управление базой дан-  
ных или логикой работы пользовательского интерфейса. В подобных случаях   
мы можем использовать декораторы, даже не зная, как они реализованы.  
Для решения своих задач программисты могут создавать собственные деко-  
раторы. Например, декораторы функций могут использоваться для добавле-  
ния возможности трассировки в другие функции, для проверки допустимости   
значений аргументов в процессе отладки, для автоматического приобретения   
и освобождения блокировок, для хронометража вызовов функций в процессе   
оптимизации и так далее. Выполнение любых операций, которые по вашему   
представлению можно было бы добавить в вызовы функций, можно рассматри-  
вать как функциональность, которую можно реализовать в виде собственного   
декоратора.  
С другой стороны, декораторы функций предназначены только для расшире-  
ния функциональных возможностей отдельных функций или методов, а не ин-  
терфейса объекта в целом. С последней ролью лучше справляются декораторы   
классов – благодаря их возможности перехватывать операции создания эк-  
земпляров, они могут использоваться для расширения или управления интер-  
фейсами объектов. Например, можно создать декораторы классов, способные   
отслеживать или проверять все обращения к атрибутам объекта. Они могут   
использоваться также для создания прокси-объектов, классов, допускающих   
создание единственного экземпляра, и реализации других распространенных   
шаблонов проектирования. Фактически, как вы увидите ниже, многие декора-  
торы классов очень близко напоминают реализацию шаблона делегирования,   
который мы рассматривали в главе 30.  
Зачем нужны декораторы?  
Подобно многим другим дополнительным инструментам языка Python декора-  
Python декора-  
 декора-  
торы никогда не становятся единственным возможным средством, с техниче-  
ской точки зрения: их функциональные возможности часто могут быть реали-  
зованы в виде вспомогательных функций или с использованием других прие-  
мов (в самом простом случае мы всегда можем вручную организовать повторное   
присваивание объектов, которое выполняется декораторами автоматически).

1090   
Глава 38. Декораторы   
Но, как бы то ни было, декораторы обеспечивают явный синтаксис для реше-  
ния таких задач, а это делает намерения программиста более очевидными, по-  
зволяет уменьшить избыточность программного кода и может помочь гаранти-  
ровать корректное использование прикладного интерфейса:  
 •  
Декораторы имеют очень очевидный синтаксис, что делает их более замет-  
ными, чем вызовы вспомогательных функций, которые могут находиться   
очень далеко от функций и классов, к которым они применяются.  
 •  
Декораторы применяются к функции или классу только один раз, когда   
они определяются, – нет никакой необходимости добавлять дополнитель-  
ный программный код (который может изменяться со временем) при каж-  
дом вызове класса или функции.  
 •  
Благодаря двум предыдущим особенностям снижается вероятность, что   
пользователь забудет дополнить функцию или класс декоратором согласно   
требованиям прикладного интерфейса.  
Другими словами, кроме технической модели, декораторы предлагают ряд   
дополнительных преимуществ, с точки зрения простоты сопровождения про-  
граммного кода и его удобочитаемости. Кроме того, как структурированные   
инструменты декораторы естественным образом способствуют инкапсуляции   
программного кода, что снижает его избыточность и упрощает внесение изме-  
нений в будущем.  
Однако у декораторов имеются и недостатки – при добавлении обертываю-  
щей логики они могут изменять типы декорируемых объектов и выполнять   
дополнительные вызовы функций. С другой стороны, все это справедливо для   
любого приема, связанного с добавлением к объектам обертывающей логики.  
Мы исследуем эти «за» и «против» на практичных примерах далее в этой гла-  
ве. Принятие решения об использовании декораторов во многом зависит от   
личных предпочтений, тем не менее, благодаря своим преимуществам они по-  
лучают все более широкое применение в мире Python. Чтобы помочь вам сфор-  
Python. Чтобы помочь вам сфор-  
. Чтобы помочь вам сфор-  
мировать собственное отношение к ним, давайте обратимся к некоторым под-  
робностям.  
Основы  
Начнем рассмотрение декораторов с точки зрения синтаксиса. Вскоре мы на-  
пишем первый действующий программный код, но так как основное волшеб-  
ство декораторов сводится к автоматической операции повторного присваива-  
ния, для начала важно понять, как это происходит.  
Декораторы функций  
Декораторы функций впервые появились в Python 2.5. Как мы уже видели ра-  
Python 2.5. Как мы уже видели ра-  
 2.5. Как мы уже видели ра-  
нее в этой книге, они в значительной степени являются всего лишь синтакси-  
ческим подсластителем – конструкцией, которая передает одну функцию в вы-  
зов другой в конце инструкции def и присваивает результат оригинальному   
имени первой функции.  
Порядок использования  
Декораторы функций являются своего рода объявлениями времени выполне-  
ния для функций, определения которых следуют за декораторами. Декоратор

Закрепление пройденного   
1091  
указывается в отдельной строке непосредственно перед инструкцией def, опре-  
деляющей функцию или метод, и состоит из символа @, за которым следует имя   
метафункции – функции (или другого вызываемого объекта), управляющей   
другой функцией.  
С точки зрения программирования, декоратор функции автоматически ото-  
бражает следующую конструкцию:  
@decorator # Декорирование функции  
def F(arg):  
 ...  
   
F(99) # Вызов функции  
в эквивалентную ей форму, где decorator – это вызываемый объект, принимаю-  
щий единственный аргумент и возвращающий вызываемый объект, который   
принимает тот же набор аргументов, что и оригинальная функция F:  
def F(arg):  
 ...  
F = decorator(F) # Присваивает имени функции результат вызова декоратора  
   
F(99) # Фактически вызывается decorator(F)(99)  
Такое автоматическое повторное присваивание имени оригинальной функции   
может применяться к любым инструкциям def, будь то определение обычной   
функции или определение метода внутри инструкции class. Когда позднее про-  
изводится вызов функции F, фактически вызывается объект, возвращаемый   
декоратором, который может быть или другим объектом, реализующим необ-  
ходимую логику, обертывающую логику оригинальной функции, или самой   
оригинальной функцией.  
Другими словами, декорирование по сути заключается в отображении первого   
вызова из следующих ниже во второй (при том, что декоратор в действитель-  
ности вызывается только один раз – на этапе декорирования):  
func(6, 7)  
decorator(func)(6, 7)  
Такое автоматическое повторное связывание имени объясняет синтаксис объ-  
явления статических методов и свойств, который мы видели ранее в книге:  
class C:  
 @staticmethod  
 def meth(...): ... # meth = staticmethod(meth)  
   
class C:  
 @property  
 def name(self): ... # name = property(name)  
В обоих случаях в конце инструкции def имени метода присваивается резуль-  
тат вызова встроенного декоратора. Вызов оригинального имени позднее при-  
ведет к вызову объекта, который вернул декоратор.  
Реализация  
Декоратор сам по себе является вызываемым объектом, который возвращает   
вызываемый объект. То есть он возвращает объект, который будет вызываться   
при вызове декорируемой функции по ее оригинальному имени, – это может

1092   
Глава 38. Декораторы   
быть объект-обертка, перехватывающий вызовы оригинальной функции, или   
сама оригинальная функция, дополненная новыми возможностями. Фактиче-  
ски декораторы могут быть вызываемыми объектами любого типа и могут воз-  
вращать вызываемые объекты любого типа: могут использоваться любые ком-  
бинации функций и классов, однако некоторые из таких комбинаций лучше   
подходят только для решения определенного круга задач.  
Например, чтобы понять, как действует протокол декорирования, позволяю-  
щий организовать управление функциями сразу после их создания, можно   
создать декоратор, имеющий следующий вид:  
def decorator(F):  
 # Обработка функции F  
 return F  
   
@decorator  
def func(): ... # func = decorator(func)  
Поскольку имени функции опять присваивается оригинальная функция, этот   
декоратор просто выполняет некоторые действия после определения функции.   
Такие декораторы могут использоваться для регистрации функций в приклад-  
ном интерфейсе, для присоединения атрибутов к функциям и тому подобного.  
В более типичном случае, чтобы добавить некоторую логику, которая перехва-  
тывает вызов функции, мы могли бы реализовать декоратор, возвращающий   
другой объект, отличный от оригинальной функции:  
def decorator(F):  
 # Сохраняет или использует функцию F  
 # Возвращает другой вызываемый объект:   
 # вложенная инструкция def, class с методом \_\_call\_\_ и так далее.  
   
@decorator  
def func(): ... # func = decorator(func)  
Этот декоратор вызывается на этапе декорирования, а возвращаемый им вы-  
зываемый объект будет вызываться позднее, при обращении к оригинальному   
имени функции. Сам декоратор принимает декорируемую функцию – возвра-  
щаемый им вызываемый объект принимает любые аргументы, которые только   
могут передаваться оригинальной функции. То же самое происходит и при де-  
корировании методов классов: подразумеваемый объект экземпляра является   
всего лишь первым аргументом возвращаемого вызываемого объекта.  
Ниже приводится типичный шаблон построения декоратора, демонстрирую-  
щий эту идею, – декоратор возвращает обертку, которая сохраняет оригиналь-  
ную функцию в области видимости объемлющей функции:  
def decorator(F): # На этапе декорирования @  
 def wrapper(\*args): # Обертывающая функция  
 # Использование F и аргументов  
 # F(\*args) – вызов оригинальной функции   
 return wrapper  
   
@decorator # func = decorator(func)  
def func(x, y): # func передается декоратору в аргументе F  
 ...  
   
func(6, 7) # 6, 7 передаются функции wrapper в виде \*args

Основы   
1093  
Когда позднее в программе будет встречен вызов функции func, в действитель-  
ности будет вызвана функция wrapper, возвращаемая декоратором� функция   
wrapper в свою очередь может вызвать оригинальную функцию func, которая   
остается доступной ей в области видимости объемлющей функции. При таком   
подходе для каждой декорированной функции создается новая область види-  
мости, в которой сохраняется информация о состоянии.  
Чтобы реализовать то же с помощью классов, мы можем реализовать метод пе-  
регрузки операции вызова и вместо области видимости объемлющей функции   
использовать атрибуты экземпляра:  
class decorator:  
 def \_\_init\_\_(self, func): # На этапе декорирования @  
 self.func = func  
 def \_\_call\_\_(self, \*args): # Обертка вызова функции  
 # Использование self.func и аргументов  
 # self.func(\*args) – вызов оригинальной функции  
   
@decorator  
def func(x, y): # func = decorator(func)  
 ... # func будет передана методу \_\_init\_\_  
   
func(6, 7) # 6, 7 передаются методу \_\_call\_\_ в виде \*args  
Когда позднее в программе будет встречен вызов функции func, в действитель-  
ности будет вызван метод \_\_call\_\_ перегрузки операторов экземпляра, создан-  
ного декоратором� метод \_\_call\_\_, в свою очередь, может вызвать оригиналь-  
ную функцию func, которая доступна ему в виде атрибута экземпляра. При   
таком подходе для каждой декорированной функции создается новый экзем-  
пляр, хранящий информацию о состоянии в своих атрибутах.  
Поддержка декорирования методов  
Здесь следует сделать одно важное замечание, касающееся предыдущего при-  
мера декоратора на основе класса: такой декоратор можно использовать для   
перехвата вызовов простых функций, но он вообще непригоден для декориро-  
вания методов классов:  
class decorator:  
 def \_\_init\_\_(self, func): # func – это метод, не связанный   
 self.func = func # с экземпляром   
 def \_\_call\_\_(self, \*args): # self – это экземпляр декоратора  
 # вызов self.func(\*args) потерпит неудачу!   
 # Экземпляр C отсутствует в args!  
   
class C:  
 @decorator  
 def method(self, x, y): # method = decorator(method)  
 ... # то есть имени method присваивается экземпляр   
 # класса decorator  
При таком подходе имени декорируемого метода присваивается экземпляр   
класса decorator, а не функция.  
Проблема, собственно, состоит в том, что позднее, при вызове метода method,   
в аргументе self методу \_\_call\_\_ декоратора будет передан экземпляр класса   
decorator, а экземпляр класса C не будет включен в список аргументов \*args. Это

1094   
Глава 38. Декораторы   
делает невозможным вызов оригинального метода – объект декоратора сохра-  
няет оригинальную функцию метода, но он не может передать ей ссылку на   
экземпляр.  
Для одновременной поддержки возможности декорирования функций и ме-  
тодов лучше всего подходит прием, основанный на применении вложенных   
функций:  
def decorator(F): # F – функция или метод, не связанный с экземпляром  
 def wrapper(\*args): # для методов - экземпляр класса в args[0]  
 # F(\*args) – вызов функции или метода  
 return wrapper  
   
@decorator  
def func(x, y): # func = decorator(func)  
 ...  
func(6, 7) # В действительности вызовет wrapper(6, 7)  
   
class C:  
 @decorator  
 def method(self, x, y): # method = decorator(method)  
 ... # Присвоит простую функцию  
   
X = C()  
X.method(6, 7) # В действительности вызовет wrapper(X, 6, 7)  
При таком подходе функция wrapper будет принимать экземпляр класса C в виде   
первого аргумента, поэтому она сможет вызвать оригинальный метод и пере-  
дать всю необходимую информацию.  
С технической точки зрения, работоспособность версии на основе вложенной   
функции обусловлена тем, что интерпретатор создает объект связанного ме-  
тода и передает подразумеваемый экземпляр класса в аргументе self, только   
когда атрибут метода ссылается на простую функцию. Когда он ссылается на   
экземпляр вызываемого класса, в аргументе self ему передается экземпляр   
вызываемого класса, чтобы обеспечить доступ к собственным данным. Какое   
значение это может иметь на практике, мы увидим в более реалистичных при-  
мерах ниже в этой главе.  
Обратите также внимание, что вложенные функции обеспечивают, пожалуй,   
самый простой способ поддержки декорирования функций и методов одновре-  
менно, но он не единственный. Дескрипторы, обсуждавшиеся в предыдущей   
главе, например, принимают при вызове сам дескриптор и подразумеваемый   
экземпляр класса. Ниже в этой главе мы увидим, как можно использовать   
дескрипторы для создания декораторов, хотя такое решение выглядит более   
сложным.  
Декораторы классов  
Декораторы функций оказались настолько удобными, что в Python 2.6 и 3.0 эта   
модель была расширена, чтобы обеспечить возможность декорирования клас-  
сов. Декораторы классов тесно связаны с декораторами функций – фактически   
они используют тот же самый синтаксис и очень похожий способ реализации.   
Вместо того чтобы обертывать отдельные функции или методы, декораторы   
классов обеспечивают способ управления классами или обертывания опера-  
ции создания экземпляров дополнительной логикой, управляющей или рас-  
ширяющей логику создания экземпляров класса.

Основы   
1095  
Порядок использования  
Синтаксически декораторы классов располагаются непосредственно перед   
инструкциями class (так же, как декораторы функций располагаются непо-  
средственно перед определениями функций). Если исходить из того, что деко-  
ратор – это функция, принимающая единственный аргумент и возвращающая   
вызываемый объект, то синтаксис декоратора класса:  
@decorator # Декорирование класса  
class C:  
 ...  
   
x = C(99) # Создает экземпляр  
эквивалентен следующей конструкции – класс автоматически передается   
функции-декоратору, а возвращаемый ею результат присваивается оригиналь-  
ному имени класса:  
class C:  
 ...  
C = decorator(C) # Присваивает имени класса результат,   
 # возвращаемый декоратором  
   
x = C(99) # Фактически вызовет decorator(C)(99)  
Суть заключается в том, что позднее, при вызове имени класса, для создания   
экземпляра вместо оригинального класса будет вызван вызываемый объект,   
возвращенный декоратором.  
Реализация  
Для создания декораторов классов используются почти те же приемы, которые   
используются при создании декораторов функций. Поскольку декоратор клас-  
са также является вызываемым объектом, который возвращает вызываемый   
объект, могут быть сконструированы любые комбинации функций и классов.  
При таком подходе результат работы декоратора вызывается, когда позднее   
в программе производится попытка создать экземпляр класса. Например, что-  
бы просто выполнить некоторые операции сразу после создания класса, нужно   
вернуть сам оригинальный класс:  
def decorator(C):  
 # Обработать класс C  
 return C  
   
@decorator  
class C: ... # C = decorator(C)  
Чтобы добавить уровень обертывающей логики, которая позднее будет пере-  
хватывать попытки создания экземпляров, декоратор должен вернуть вызы-  
ваемый объект:  
def decorator(C):  
 # Сохранить или использовать класс C  
 # Возвращает другой вызываемый объект:   
 # вложенная инструкция def, class с методом \_\_call\_\_ и так далее.  
   
@decorator  
class C: ... # C = decorator(C)

1096   
Глава 38. Декораторы   
Вызываемый объект, возвращаемый таким декоратором класса, обычно соз-  
дает и возвращает новый экземпляр оригинального класса, изменяя или до-  
полняя его интерфейс. Например, следующий декоратор, который добавляет   
в объект обработку операций обращения к неопределенным атрибутам:  
def decorator(cls): # На этапе декорирования @  
 class Wrapper:  
 def \_\_init\_\_(self, \*args): # На этапе создании экземпляра  
 self.wrapped = cls(\*args)  
 def \_\_getattr\_\_(self, name): # Вызывается при обращении к атрибуту  
 return getattr(self.wrapped, name)  
 return Wrapper  
   
@decorator  
class C: # C = decorator(C)  
 def \_\_init\_\_(self, x, y): # Вызывается методом Wrapper.\_\_init\_\_  
 self.attr = ‘spam’  
   
x = C(6, 7) # В действительности вызовет Wrapper(6, 7)  
print(x.attr) # Вызовет Wrapper.\_\_getattr\_\_, выведет “spam”  
В этом примере декоратор присвоит оригинальному имени класса другой   
класс, который сохраняет оригинальный класс в области видимости объемлю-  
щей функции, создает и встраивает экземпляр оригинального класса при вы-  
зове. Когда позднее будет выполнена попытка прочитать значение атрибута эк-  
земпляра, она будет перехвачена методом \_\_getattr\_\_ обертки и делегирована   
встроенному экземпляру оригинального класса. Кроме того, для каждого де-  
корированного класса создается новая область видимости объемлющей функ-  
ции, в которой сохраняется оригинальный класс. Мы еще встретимся с этим   
шаблоном далее в этой главе, в составе более реального примера.  
Подобно декораторам функций декораторы классов обычно создаются в виде   
«фабричных» функций, которые создают и возвращают вызываемые объекты,   
классы с методами \_\_init\_\_ или \_\_call\_\_ для перехвата операций вызова или их   
комбинации. Фабричные функции обычно сохраняют информацию о состоя-  
нии в области видимости объемлющей функции, а классы – в атрибутах.  
Поддержка множества экземпляров  
Как и в случае с декораторами функций, одни комбинации типов вызываемых   
объектов лучше подходят для решения определенных задач, чем другие. Рас-  
смотрим следующую альтернативную и ошибочную реализацию декоратора из   
предыдущего примера:  
class Decorator:  
 def \_\_init\_\_(self, C): # На этапе декорирования @  
 self.C = C  
 def \_\_call\_\_(self, \*args): # На этапе создания экземпляра  
 self.wrapped = self.C(\*args)  
 return self  
 def \_\_getattr\_\_(self, attrname): # Вызывается при обращении к атрибуту  
 return getattr(self.wrapped, attrname)  
   
@Decorator  
class C: ... # C = Decorator(C)  
   
x = C()  
y = C() # Затрет x!

Основы   
1097  
Эта версия может использоваться для декорирования множества классов (для   
каждого из них будет создан новый экземпляр класса Decorator) и будет пере-  
хватывать попытки создания экземпляров (каждый раз будет вызываться ме-  
тод \_\_call\_\_). Однако в отличие от предыдущей версии, данная версия не под-  
держивает работу с множеством экземпляров одного класса – операция созда-  
ния очередного экземпляра будет затирать предыдущий экземпляр. Первона-  
чальная версия поддерживает возможность создания множества экземпляров,   
потому что при создании каждого экземпляра создается новый, независимый   
объект-обертка. В более общем смысле, любой из следующих вариантов под-  
держивает возможность создания множества обернутых экземпляров:  
def decorator(C): # На этапе декорирования @  
 class Wrapper:  
 def \_\_init\_\_(self, \*args): # Вызывается при создании экземпляра  
 self.wrapped = C(\*args)  
 return Wrapper  
   
class Wrapper: ...  
def decorator(C): # На этапе декорирования @  
 def onCall(\*args): # На этапе создания экземпляра  
 return Wrapper(C(\*args)) # Встраивает экземпляр в экземпляр  
 return onCall  
Мы изучим это явление на более реалистичном примере ниже в этой главе –   
однако на практике необходимо проявлять осторожность и выбирать комби-  
нации типов вызываемых объектов, которые лучше соответствуют нашим на-  
мерениям.  
Вложение декораторов  
Иногда одного декоратора бывает недостаточно. Для поддержки многоступен-  
чатых расширений синтаксис декораторов позволяет добавлять несколько   
уровней обертывающей логики к декорируемой функции или методу. При ис-  
пользовании такой возможности каждый декоратор должен указываться в от-  
дельной строке. Синтаксическая конструкция следующего вида:  
@A  
@B  
@C  
def f(...):  
 ...  
равноценна следующей:  
def f(...):  
 ...  
f = A(B(C(f)))  
Здесь оригинальная функция передается трем различным декораторам, а по-  
лучившийся в результате вызываемый объект присваивается оригинальному   
имени. Каждый декоратор обрабатывает результат, возвращаемый преды-  
дущим декоратором, который может быть оригинальной функцией или   
объектом-оберткой.  
Если все декораторы возвращают обертки, то при вызове функции по ориги-  
нальному имени будет выполнена логика всех трех обертывающих объектов,   
расширяя возможности функции тремя различными способами. Последний

1098   
Глава 38. Декораторы   
декоратор в списке будет задействован первым и окажется самым глубоко вло-  
женным.  
Так же как и в случае декорирования функций, применение нескольких де-  
кораторов классов приведет к вызову нескольких вложенных функций и, воз-  
можно, к созданию нескольких уровней обертывающей логики вокруг опера-  
ции создания экземпляров. Например, следующий фрагмент:  
@spam  
@eggs  
class C:  
 ...  
   
X = C()  
эквивалентен следующему:  
class C:  
 ...  
C = spam(eggs(C))  
   
X = C()  
Как и прежде, каждый декоратор может возвращать оригинальный класс или   
объект-обертку. Если оба декоратора в примере возвращают обертки, то позд-  
нее, когда будет предпринята попытка создать экземпляр оригинального клас-  
са C, вызов будет направлен обертывающим объектам, созданным обоими де-  
кораторами, spam и eggs, которые могут преследовать совершенно разные цели.  
Например, следующие декораторы просто возвращают декорируемую функ-  
цию, не выполняя никаких дополнительных действий:  
def d1(F): return F  
def d2(F): return F  
def d3(F): return F  
   
@d1  
@d2  
@d3  
def func(): # func = d1(d2(d3(func)))  
 print(‘spam’)  
   
func() # Выведет “spam”  
При применении аналогичных ничего не делающих декораторов к классам ре-  
зультат будет похожим.  
Однако когда декораторы добавляют обертывающие объекты функций, они   
могут расширять возможности оригинальных функций – в следующем при-  
мере происходит объединение результатов в ходе выполнения уровней оберты-  
вающей логики декораторов от внутренней к внешней:  
def d1(F): return lambda: ‘X’ + F()  
def d2(F): return lambda: ‘Y’ + F()  
def d3(F): return lambda: ‘Z’ + F()  
   
@d1  
@d2  
@d3  
def func(): # func = d1(d2(d3(func)))

Основы   
1099  
 return ‘spam’  
   
print(func()) # Выведет “XYZspam”  
Для реализации обертывающей логики мы использовали здесь lambda-функции   
(каждая из них сохраняет обертываемую функцию в области видимости объем-  
лющей функции)� на практике обертки могут быть реализованы в виде функ-  
ций, вызываемых классов и других объектов. В случае удачной конструкции   
декораторов возможность их вложения друг в друга дает нам возможность со-  
ставлять из них самые разнообразные комбинации.  
Аргументы декораторов  
Обе разновидности декораторов, декораторы функций и декораторы классов,   
могут принимать дополнительные аргументы, хотя в действительности эти   
аргументы передаются вызываемому объекту, возвращающему декоратор, ко-  
торый в свою очередь возвращает вызываемый объект. Например, следующий   
программный код:  
@decorator(A, B)  
def F(arg):  
 ...  
   
F(99)  
автоматически будет преобразован в эквивалентную форму, где decorator – это   
вызываемый объект, возвращающий фактический декоратор. Возвращаемый   
фактический декоратор, в свою очередь, возвращает вызываемый объект, кото-  
рый позднее будет использоваться для перехвата вызовов оригинальной функ-  
ции:  
def F(arg):  
 ...  
F = decorator(A, B)(F) # Присваивает имени F результат вызова объекта,   
 # возвращаемого вызываемым объектом decorator  
   
F(99) # Фактически вызывается decorator(A, B)(F)(99)  
Аргументы декораторов интерпретируются еще до того как будет выполнена   
операция декорирования, и обычно в них передаются значения для использо-  
вания в последующих вызовах. В подобных случаях функция decorator, напри-  
мер, может иметь такой вид:  
def decorator(A, B):  
 # Сохранить или использовать A, B  
 def actualDecorator(F):  
 # Сохранить или использовать функцию F  
 # Возвращает другой вызываемый объект:  
 # вложенная инструкция def, class с методом \_\_call\_\_ и так далее.  
 return callable  
 return actualDecorator  
При такой организации внешняя функция обычно сохраняет аргументы деко-  
ратора вместе с другой информацией о состоянии для последующего использо-  
вания внутри фактического декоратора – вызываемого объекта, возвращаемо-  
го функцией. Этот фрагмент сохраняет аргументы в области видимости объ-

1100   
Глава 38. Декораторы   
емлющей функции, но для этих целей можно также использовать атрибуты   
классов.  
Другими словами, аргументы декораторов часто подразумевают три уровня   
вызываемых объектов: вызываемый объект, принимающий аргументы декора-  
тора, возвращает вызываемый объект, который будет играть роль декоратора,   
который в свою очередь возвращает вызываемый объект для обработки вызо-  
вов оригинальной функции или класса. Каждый из этих трех уровней может   
быть функцией или классом и может сохранять информацию в области види-  
мости объемлющей функции или в атрибутах класса. Конкретные примеры ис-  
пользования декораторов с аргументами мы увидим далее в этой главе.  
Декораторы могут управлять функциями   
и классами одновременно  
В оставшейся части главы основное наше внимание будет уделяться оберты-  
ванию вызовов функций и классов, тем не менее, я должен заметить, что ме-  
ханизм декораторов обладает более широкими возможностями – это протокол   
передачи функций и классов вызываемым объектам сразу же после их созда-  
ния. Таким образом, декораторы могут использоваться для выполнения произ-  
вольных операций сразу после создания декорируемых объектов:  
def decorate(O):  
 # Сохраняет или дополняет функцию или класс O  
 return O  
   
@decorator  
def F(): ... # F = decorator(F)  
   
@decorator  
class C: ... # C = decorator(C)  
Если декоратор возвращает не обертку, а оригинальный декорируемый объект,   
как показано выше, он может использоваться для управления самими функ-  
циями и классами. Ниже в этой главе будут представлены более реалистичные   
примеры, которые используют эту идею для регистрации вызываемых объек-  
тов в прикладном интерфейсе с помощью декорирования и присваивания атри-  
бутов функциям после их создания.  
Программирование декораторов функций  
В оставшейся части главы мы будем изучать действующие примеры, демон-  
стрирующие практическое применение концепций декораторов, которые мы   
только что исследовали. В этом разделе будут представлены несколько декора-  
торов функций, а в следующем – несколько декораторов классов. А в заверше-  
ние главы мы рассмотрим некоторые достаточно крупные примеры использо-  
вания декораторов функций и классов.  
Трассировка вызовов  
Для начала мы вернемся к примеру трассировщика вызовов, с которым мы   
встретились в главе 31. Следующий пример демонстрирует определение и ис-  
пользование декоратора функций, который подсчитывает количество вызовов

Программирование декораторов функций   
1101  
декорированной функции и при каждом ее вызове выводит трассировочное со-  
общение:  
class tracer:  
 def \_\_init\_\_(self, func): # На этапе декорирования @:   
 self.calls = 0 # сохраняет оригинальную функцию func  
 self.func = func  
 def \_\_call\_\_(self, \*args): # При последующих вызовах: вызывает   
 self.calls += 1 # оригинальную функцию func  
 print(‘call %s to %s’ % (self.calls, self.func.\_\_name\_\_))  
 self.func(\*args)  
   
@tracer  
def spam(a, b, c): # spam = tracer(spam)  
 print(a + b + c) # Обертывает функцию spam объектом декоратора  
Обратите внимание, что при декорировании каждой функции этим классом   
создается новый экземпляр, который сохраняет объект функции и счетчик ее   
вызовов. Обратите также внимание, как используется синтаксис \*args аргу-  
ментов, позволяющий организовать упаковывание и распаковывание произ-  
вольного количества аргументов. Такой прием позволяет использовать декора-  
тор для обертывания любых функций, принимающих любое число аргументов   
(эта версия декоратора не может применяться к методам классов, но мы испра-  
вим этот недостаток ниже, в этом разделе).  
Если теперь импортировать эту функцию из модуля и протестировать ее в ин-  
терактивной оболочке, мы получим следующие ниже результаты – при каж-  
дом вызове функции будет выводиться трассировочное сообщение благодаря   
тому, что класс декоратора перехватывает их. Эта реализация будет работать   
под управлением обеих версий Python, 2.6 и 3.0, как и все остальные примеры   
в этой главе, если явно не указывается иное:  
>>> from decorator1 import spam  
   
>>> spam(1, 2, 3) # В действительности вызывается объект-обертка  
call 1 to spam  
6  
   
>>> spam(‘a’, ‘b’, ‘c’) # Вызовет метод \_\_call\_\_ класса  
call 2 to spam  
abc  
   
>>> spam.calls # Счетчик вызовов в объекте-обертке  
2  
   
>>> spam  
<decorator1.tracer object at 0x02D9A730>  
При вызове класс tracer сохраняет декорируемую функцию и в последующем   
будет перехватывать ее вызовы, выполняя дополнительные операции по под-  
счету количества вызовов и выводу сообщения для каждого вызова. Обратите   
внимание, что счетчик вызовов реализован в виде атрибута декорированной   
функции – в действительности, после декорирования имя spam – это экземпляр   
класса tracer (что может запутать программы, проверяющие типы объектов,   
но в общем случае не должно порождать проблем).  
Для случаев, когда требуется перехватывать вызовы функций, синтаксис @ де-  
кораторов может оказаться гораздо удобнее, чем добавление логики подсчета

1102   
Глава 38. Декораторы   
в каждый вызов, а кроме того, он позволяет избежать непосредственного вызо-  
ва оригинальной функции по невнимательности. Рассмотрим эквивалентную   
реализацию без использования декоратора:  
calls = 0  
def tracer(func, \*args):  
 global calls  
 calls += 1  
 print(‘call %s to %s’ % (calls, func.\_\_name\_\_))  
 func(\*args)  
   
def spam(a, b, c):  
 print(a, b, c)  
   
>>> spam(1, 2, 3) # Обычный вызов без трассировки: невнимательность?  
1 2 3  
   
>>> tracer(spam, 1, 2, 3) # Вызов с трассировкой без применения декоратора  
call 1 to spam  
1 2 3  
Такое альтернативное решение может быть использовано для любой функции   
без применения специального синтаксиса @, но, в отличие версии с примене-  
нием декоратора, оно требует добавления дополнительных синтаксических   
конструкций везде, где вызывается функция. Кроме того, намерения програм-  
миста в этом случае могут оказаться недостаточно очевидными и не гарантиру-  
ется, что дополнительный уровень логики будет вызываться при обращениях   
к оригинальной функции. Хотя без декораторов всегда можно обойтись (мы   
можем реализовать повторное присваивание вручную), тем не менее, они часто   
предлагают более удобный способ.  
Способы сохранения информации о состоянии  
Последний пример в предыдущем разделе поднимает весьма важную пробле-  
му. Декораторы функций могут использовать самые разные способы сохране-  
ния информации о состоянии, предоставляемой на этапе декорирования, для   
последующего использования в фактических вызовах функции. Обычно де-  
кораторы должны поддерживать возможность создания множества декориро-  
ванных объектов и выполнения множества вызовов, однако достичь этой цели   
можно разными путями: атрибуты экземпляров, глобальные переменные, не-  
локальные переменные и атрибуты функций – все они могут использоваться   
для сохранения информации о состоянии.  
Атрибуты экземпляра класса  
Например, ниже приводится расширенная версия предыдущего примера, в ко-  
торую добавлена поддержка именованных  аргументов и возврат значения,   
полученного от обернутой функции с целью расширить область применение   
декоратора:  
class tracer: # Состояние сохраняется в атрибутах экземпляра  
 def \_\_init\_\_(self, func): # На этапе декорирования @  
 self.calls = 0   
 self.func = func # Cохраняет оригинальную функцию func  
 def \_\_call\_\_(self, \*args, \*\*kwargs): # Вызывается при обращениях

Программирование декораторов функций   
1103  
 self.calls += 1 # к оригинальной функции  
 print(‘call %s to %s’ % (self.calls, self.func.\_\_name\_\_))  
 return self.func(\*args, \*\*kwargs)  
   
@tracer  
def spam(a, b, c): # То же, что и spam = tracer(spam)  
 print(a + b + c) # Вызывает метод tracer.\_\_init\_\_  
   
@tracer  
def eggs(x, y): # То же, что и eggs = tracer(eggs)  
 print(x \*\* y) # Обертывает функцию eggs объектом tracer   
   
spam(1, 2, 3) # В действительности вызывается tracer.\_\_call\_\_  
spam(a=4, b=5, c=6) # spam – атрибут экземпляра  
   
eggs(2, 16) # В действительности вызывается tracer.\_\_call\_\_,   
eggs(4, y=4) # self.func – это функция eggs, self.calls – отдельное   
 # значение для каждой функции   
Как и в предыдущей версии, для сохранения информации о состоянии здесь   
используются атрибуты экземпляра. Обе обернутые функции и счетчики вы-  
зовов сохраняются как индивидуальные данные экземпляров – каждая опе-  
рация декорирования создает собственную копию. Если запустить эту версию   
под управлением Python 2.6 или 3.0, он выведет следующие результаты – об-  
Python 2.6 или 3.0, он выведет следующие результаты – об-  
 2.6 или 3.0, он выведет следующие результаты – об-  
ратите внимание, что для каждой из функций, spam и eggs, имеется свой счет-  
чик вызовов, потому что каждая операция декорирования создает отдельный   
экземпляр:  
call 1 to spam  
6  
call 2 to spam  
15  
call 1 to eggs  
65536  
call 2 to eggs  
256  
Несмотря на удобство этого приема для декорирования функций, он не может   
использоваться для декорирования методов (подробнее об этом рассказывается   
ниже).  
Области видимости объемлющих функций   
и глобальные переменные  
В достижении подобного эффекта часто могут помочь ссылки на переменные   
в области видимости объемлющих функций и вложенные инструкции def,   
особенно для статических данных, таких как ссылка на оригинальную функ-  
цию. Однако в этом случае нам также потребовалось бы сохранить в области   
видимости объемлющей функции счетчик вызовов, изменяющийся при каж-  
дом вызове, что невозможно в Python 2.6. В версии 2.6 мы можем либо исполь-  
Python 2.6. В версии 2.6 мы можем либо исполь-  
 2.6. В версии 2.6 мы можем либо исполь-  
зовать классы и атрибуты, как в примере выше, либо перенести переменную,   
хранящую информацию о состоянии, в глобальную область видимости и ис-  
пользовать объявление global:  
calls = 0 # Информация сохраняется в объемлющей   
def tracer(func): # области видимости и в глобальной переменной

1104   
Глава 38. Декораторы   
 def wrapper(\*args, \*\*kwargs): # Вместо атрибутов класса  
 global calls # calls – глобальная переменная, общая для   
 calls += 1 # всех функций, а не для каждой в отдельности  
 print(‘call %s to %s’ % (calls, func.\_\_name\_\_))  
 return func(\*args, \*\*kwargs)  
 return wrapper  
   
@tracer  
def spam(a, b, c): # То же, что и spam = tracer(spam)  
 print(a + b + c)  
   
@tracer  
def eggs(x, y): # То же, что и eggs = tracer(eggs)  
 print(x \*\* y)  
   
spam(1, 2, 3) # В действительности вызывается wrapper,   
spam(a=4, b=5, c=6) # присвоенная имени функции, wrapper вызывает spam  
   
eggs(2, 16) # В действительности вызывается wrapper,  
eggs(4, y=4) # присвоенная имени eggs. Глобальная переменная   
 # calls – общая для всех функций!  
К сожалению, после того как счетчик был перенесен в глобальную область ви-  
димости, он стал общим для всех обернутых функций. В отличие от атрибу-  
тов экземпляров, глобальные счетчики не могут использоваться для подсчета   
вызовов каждой функции в отдельности – счетчик наращивается при вызове   
любой трассируемой функции. Вы можете сами заметить различия, если срав-  
ните вывод этой версии и предыдущей – единый, общий счетчик вызовов из-  
меняется при вызове любой декорированной функции:  
call 1 to spam  
6  
call 2 to spam  
15  
call 3 to eggs  
65536  
call 4 to eggs  
256  
Области видимости объемлющих функций   
и нелокальные переменные  
В некоторых случаях бывает желательным именно такой способ сохранения   
информации в глобальных переменных. Однако если требуется обеспечить под-  
счет вызовов каждой функции в отдельности, мы можем использовать классы,   
как было показано прежде, или задействовать новую инструкцию nonlocal, по-  
явившуюся в Python 3.0 и описанную в главе 17. Поскольку новая инструкция   
позволяет изменять переменные в области видимости объемлющей функции,   
они могут использоваться для хранения данных, индивидуальных для каждой   
функции:  
def tracer(func): # Информация сохраняется в объемлющей области  
 calls = 0 # видимости и в нелокальной переменной вместо  
 # атрибутов класса и глобальных переменных  
 def wrapper(\*args, \*\*kwargs): # calls – не глобальная переменная   
 nonlocal calls # для каждой функции в отдельности

Программирование декораторов функций   
1105  
 calls += 1  
 print(‘call %s to %s’ % (calls, func.\_\_name\_\_))  
 return func(\*args, \*\*kwargs)  
 return wrapper  
   
@tracer  
def spam(a, b, c): # То же, что и spam = tracer(spam)  
 print(a + b + c)  
   
@tracer  
def eggs(x, y): # То же, что и eggs = tracer(eggs)  
 print(x \*\* y)  
   
spam(1, 2, 3) # В действительности вызывается wrapper,   
spam(a=4, b=5, c=6) # присвоенная имени функции, wrapper вызывает spam  
   
eggs(2, 16) # В действительности вызывается wrapper,  
eggs(4, y=4) # присвоенная имени eggs. Нелокальная переменная   
 # calls теперь отдельная для каждой функции  
Теперь благодаря тому, что переменные в области видимости объемлющей   
функции не являются глобальными, каждая обернутая функция получает   
свой собственный счетчик, как при использовании классов и атрибутов. Ниже   
приводятся результаты работы этого сценария под управлением Python 3.0:  
call 1 to spam  
6  
call 2 to spam  
15  
call 1 to eggs  
65536  
call 2 to eggs  
256  
Атрибуты функций  
Наконец, даже если вы не используете Python 3.X и не можете задействовать   
инструкцию nonlocal, вы все равно имеете возможность отказаться от глобаль-  
ных переменных и классов, использовав атрибуты функций для хранения не-  
которой изменяемой информации о состоянии. В последних версиях Python   
допускается присоединять к функциям произвольные атрибуты и присваивать   
им значения инструкцией вида func.attr=value. В нашем примере мы можем   
определить атрибут wrapper.calls для хранения счетчика. Следующий пример   
работает точно так же, как предыдущая версия, основанная на использовании   
инструкции nonlocal, – благодаря тому, что каждая декорируемая функция   
получает свой собственный счетчик, при этом он также будет работать и под   
управлением Python 2.6:  
def tracer(func): # Информация сохраняется в объемлющей   
 def wrapper(\*args, \*\*kwargs): # области видимости и в атрибутах функции  
 wrapper.calls += 1 # calls – не глобальная переменная,   
 # для каждой функции в отдельности  
 print(‘call %s to %s’ % (wrapper.calls, func.\_\_name\_\_))  
 return func(\*args, \*\*kwargs)  
 wrapper.calls = 0  
 return wrapper

1106   
Глава 38. Декораторы   
Обратите внимание, что этот прием работает, так как имя wrapper сохраняется   
в области видимости объемлющей функции tracer. Когда позднее выполняется   
наращивание счетчика wrapper.calls, сам объект с именем wrapper не изменяет-  
ся, поэтому нам не требуется объявление nonlocal.  
Этот прием приведен последним, потому что он менее очевидный, чем приме-  
нение инструкции nonlocal в версии 3.0 и, вероятно, его следует использовать   
только в случаях, когда другие приемы не могут быть использованы. Однако   
мы будем использовать его в ответе на один из контрольных вопросов в конце   
главы, когда нам потребуется организовать доступ к информации о состоянии   
из-за пределов декоратора. Нелокальные переменные доступны только внутри   
вложенной функции, а атрибуты функций имеют более широкую область ви-  
димости.  
Поскольку декораторы часто могут образовывать несколько уровней вызывае-  
мых объектов, вы можете комбинировать функции с объемлющими областя-  
ми видимости и классы с атрибутами для достижения различных желаемых   
эффектов. Однако, как мы увидим ниже, добиться нужного результата порой   
оказывается сложнее, чем можно было бы ожидать, – когда каждая декорируе-  
мая функция должна обладать собственной информацией о состоянии и каж-  
дый декорируемый класс может требовать хранить информацию как свою соб-  
ственную, так и для каждого экземпляра в отдельности.  
Фактически, как будет рассказываться в следующем разделе, если нам по-  
требуется, чтобы декоратор функций дополнительно поддерживал возмож-  
ность применения к методам классов, нам также потребуется внимательнее   
отнестись к различиям между декораторами, реализованными в виде объектов   
экземпляров вызываемых классов, и декораторами, реализованными в виде   
функций.  
Ошибки при использовании классов I:   
декорирование методов классов  
Когда я писал первый декоратор tracer функций, представленный выше, я наи-  
вно полагал, что он также может применяться к любым методам – декориро-  
ванные методы должны действовать точно так же, а автоматически передавае-  
мый аргумент self со ссылкой на экземпляр в этом случае мог бы просто быть   
добавлен в начало списка аргументов \*args. К сожалению, я ошибался: если   
применить первую версию декоратора tracer к методу класса, он потерпит не-  
удачу, потому что аргумент self будет ссылаться на экземпляр класса декора-  
тора, а ссылка на подразумеваемый экземпляр декорируемого класса не будет   
включена в список \*args. Это справедливо для обеих версий Python, 3.0 и 2.6.  
Я познакомил вас с этим явлением выше в этой главе но теперь вы можете на-  
блюдать его в контексте более реалистичного примера. Допустим, имеется де-  
коратор трассировки вызовов функций, реализованный на основе класса:  
class tracer:  
 def \_\_init\_\_(self, func): # На этапе декорирования @  
 self.calls = 0 # Сохраняет функцию для последующего вызова  
 self.func = func  
 def \_\_call\_\_(self, \*args, \*\*kwargs): # Вызывается при обращениях  
 self.calls += 1 # к оригинальной функции  
 print(‘call %s to %s’ % (self.calls, self.func.\_\_name\_\_))

Программирование декораторов функций   
1107  
 return self.func(\*args, \*\*kwargs)  
Тогда декорирование простых функций будет работать именно так, как описы-  
валось выше:  
@tracer  
def spam(a, b, c): # spam = tracer(spam)  
 print(a + b + c) # Вызовет метод tracer.\_\_init\_\_  
   
spam(1, 2, 3) # Вызовет метод tracer.\_\_call\_\_  
spam(a=4, b=5, c=6) # spam – атрибут экземпляра  
Однако декорирование методов класса не даст желаемого результата (наибо-  
лее внимательные читатели без труда узнают в нем класс Person из пособия по   
объектно-ориентированному программированию в главе 27):  
class Person:  
 def \_\_init\_\_(self, name, pay):  
 self.name = name  
 self.pay = pay  
   
 @tracer  
 def giveRaise(self, percent): # giveRaise = tracer(giverRaise)  
 self.pay \*= (1.0 + percent)  
   
 @tracer  
 def lastName(self): # lastName = tracer(lastName)  
 return self.name.split()[-1]  
   
bob = Person(‘Bob Smith’, 50000) # tracer запоминает метод  
bob.giveRaise(.25) # Вызовет tracer.\_\_call\_\_(???, .25)  
print(bob.lastName()) # Вызовет tracer.\_\_call\_\_(???)  
Корень этой проблемы заключен в аргументе self метода \_\_call\_\_ класса tracer:   
это экземпляр класса tracer или класса Person? В действительности нам нужны   
оба экземпляра: экземпляр класса tracer необходим для сохранения информа-  
ции о состоянии, а экземпляр класса Person – чтобы вызвать оригинальный ме-  
тод. На самом деле аргумент self должен быть ссылкой на объект tracer, чтобы   
обеспечить доступ к информации о состоянии трассировщика� это одинаково   
верно как в случае декорирования функции, так и в случае декорирования ме-  
тода.  
К сожалению, когда имени декорируемого метода повторно присваивается эк-  
земпляр класса с методом \_\_call\_\_, интерпретатор передает в аргументе self   
только экземпляр класса tracer – он вообще не передает в списке аргументов   
подразумеваемый экземпляр класса Person. Кроме того, из-за того, что экзем-  
пляр класса tracer вообще ничего не знает об экземпляре класса Person, необхо-  
димом для вызова метода, мы не можем создать метод, связанный с экземпля-  
ром, и поэтому не можем корректно произвести вызов.  
Фактически в предыдущем примере мы сталкиваемся с ситуацией, когда деко-  
рируемый метод получает недостаточное количество аргументов, что вызывает   
ошибку. Если добавить в метод \_\_call\_\_ вывод всех аргументов, можно будет   
убедиться, что аргумент self представляет экземпляр класса tracer, а экзем-  
пляр класса Person вообще отсутствует:  
<\_\_main\_\_.tracer object at 0x02D6AD90> (0.25,) {}  
call 1 to giveRaise

1108   
Глава 38. Декораторы   
Traceback (most recent call last):  
 File “C:/misc/tracer.py”, line 56, in <module>  
 bob.giveRaise(.25)  
 File “C:/misc/tracer.py”, line 9, in \_\_call\_\_  
 return self.func(\*args, \*\*kwargs)  
TypeError: giveRaise() takes exactly 2 positional arguments (1 given)  
Как уже упоминалось выше, это происходит потому, что интерпретатор пере-  
дает в аргументе self подразумеваемый экземпляр, когда происходит связыва-  
ние имени метода с простой функцией, – когда имени соответствует экземпляр   
вызываемого класса, передается экземпляр этого класса. С технической точки   
зрения, интерпретатор создает объект связанного метода, содержащий подраз-  
умеваемый экземпляр, только когда метод является простой функцией.  
Использование вложенных функций   
для декорирования методов  
Если вам необходимо, чтобы декоратор мог применяться и к простым функци-  
ям, и к методам классов, наиболее простое решение заключается в использова-  
нии одного из других способов организации сохранения информации о состоя-  
нии, описанных выше. Реализуйте свой декоратор в виде вложенной функции,   
чтобы не зависеть от единственного аргумента self для представления экзем-  
пляра класса-обертки и подразумеваемого экземпляра класса.  
Следующее альтернативное решение решает описанную проблему с помощью   
нелокальных переменных, которые могут использоваться в Python 3.0. По-  
Python 3.0. По-  
 3.0. По-  
скольку теперь имени декорируемого метода присваивается не экземпляр   
класса, а простая функция, интерпретатор будет корректно передавать ей эк-  
земпляр класса Person в первом аргументе, а декоратор передаст его в виде пер-  
вого элемента в списке \*args настоящему методу:  
# Этот декоратор может применяться к функциям и к методам  
   
def tracer(func): # Вместо класса с методом \_\_call\_\_ используется функция,  
 calls = 0 # иначе “self” будет представлять экземпляр декоратора!  
 def onCall(\*args, \*\*kwargs):  
 nonlocal calls  
 calls += 1  
 print(‘call %s to %s’ % (calls, func.\_\_name\_\_))  
 return func(\*args, \*\*kwargs)  
 return onCall  
   
# Может применяться к простым функциям  
   
@tracer  
def spam(a, b, c): # spam = tracer(spam)  
 print(a + b + c) # onCall сохранит ссылку на spam  
   
spam(1, 2, 3) # Вызовет onCall(1, 2, 3)  
spam(a=4, b=5, c=6)  
   
# Может применяться к методам классов!  
   
class Person:  
 def \_\_init\_\_(self, name, pay):  
 self.name = name  
 self.pay = pay

Программирование декораторов функций   
1109  
 @tracer  
 def giveRaise(self, percent): # giveRaise = tracer(giverRaise)  
 self.pay \*= (1.0 + percent) # onCall сохранит ссылку на giveRaise  
   
 @tracer  
 def lastName(self): # lastName = tracer(lastName)  
 return self.name.split()[-1]  
   
print(‘methods...’)  
bob = Person(‘Bob Smith’, 50000)  
sue = Person(‘Sue Jones’, 100000)  
print(bob.name, sue.name)  
sue.giveRaise(.10) # Выховет onCall(sue, .10)  
print(sue.pay)  
print(bob.lastName(), sue.lastName()) # Вызовет onCall(bob), lastName – в   
 # области видимости объемлющей функции  
Эта версия действует одинаково хорошо с функциями и с методами:  
call 1 to spam  
6  
call 2 to spam  
15  
methods...  
Bob Smith Sue Jones  
call 1 to giveRaise  
110000.0  
call 1 to lastName  
call 2 to lastName  
Smith Jones  
Использование дескрипторов для декорирования методов  
Решение на основе вложенных функций, продемонстрированное в предыду-  
щем разделе, является наиболее простым, когда необходимо обеспечить под-  
держку применения декоратора к функциям и к методам классов, однако воз-  
можны и другие решения. Дескрипторы, особенности которых мы исследовали   
в предыдущей главе, например, также могут помочь нам.  
Вспомните, как в той главе говорилось, что дескрипторы могут быть атрибу-  
тами классов, которым присваиваются объекты с методом \_\_get\_\_, который   
вызывается автоматически при попытке сослаться на атрибут и получить его   
значение (наследование класса object необходимо в Python 2.6, но не в 3.0):  
class Descriptor(object):  
 def \_\_get\_\_(self, instance, owner): ...  
   
class Subject:  
 attr = Descriptor()  
   
X = Subject()  
X.attr # Приблизительно вызов будет выглядеть так:   
 # Descriptor.\_\_get\_\_(Subject.attr, X, Subject)  
Кроме того, дескрипторы могут иметь методы \_\_set\_\_ и \_\_del\_\_ управления   
доступом, но в данном случае они нам не потребуются. Теперь, учитывая, что   
метод \_\_get\_\_ дескриптора принимает класс дескриптора и подразумеваемый   
экземпляр класса, он отлично подходит для декорирования методов, когда для

1110   
Глава 38. Декораторы   
выполнения вызова требуется иметь доступ к информации декоратора и к ори-  
гинальному экземпляру класса. Рассмотрим следующий альтернативный де-  
 следующий альтернативный де-  
следующий альтернативный де-  
 альтернативный де-  
альтернативный де-  
 де-  
де-  
коратор трассировки, который одновременно является дескриптором:  
class tracer(object):  
 def \_\_init\_\_(self, func): # На этапе декорирования @  
 self.calls = 0 # Сохраняет функцию для последующего вызова  
 self.func = func  
 def \_\_call\_\_(self, \*args, \*\*kwargs): # Вызывается при обращениях к  
 self.calls += 1 # оригинальной функции  
 print(‘call %s to %s’ % (self.calls, self.func.\_\_name\_\_))  
 return self.func(\*args, \*\*kwargs)  
 def \_\_get\_\_(self, instance, owner): # Вызывается при обращении к атрибуту  
 return wrapper(self, instance)  
   
class wrapper:  
 def \_\_init\_\_(self, desc, subj): # Сохраняет оба экземпляра  
 self.desc = desc # Делегирует вызов дескриптору  
 self.subj = subj  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 return self.desc(self.subj, \*args, \*\*kwargs) # Вызовет tracer.\_\_call\_\_  
   
@tracer  
def spam(a, b, c): # spam = tracer(spam)  
 ...то же, что и прежде... # Использует только \_\_call\_\_  
   
class Person:  
 @tracer  
 def giveRaise(self, percent): # giveRaise = tracer(giverRaise)  
 ...то же, что и прежде... # Создаст дескриптор giveRaise  
Этот пример действует точно так же, как и предыдущий, основанный на вло-  
женной функции. При вызове декорированной функции вызывается только   
метод \_\_call\_\_, тогда как при вызове декорированных методов сначала вызыва-  
ется метод \_\_get\_\_, операцией получения метода (в выражении instance.method)�   
объект, возвращаемый методом \_\_get\_\_, хранит подразумеваемый экземпляр   
класса, а затем выполняется выражение вызова, что приводит к вызову метода   
\_\_call\_\_ (в выражении (args...)). Например, в тестовом программном коде вы-  
полняется вызов  
sue.giveRaise(.10) # Вызовет \_\_get\_\_, затем \_\_call\_\_  
который сначала вызовет метод tracer.\_\_get\_\_, потому что атрибуту giveRaise   
в классе Person был присвоен дескриптор декоратора функций. Затем выраже-  
ние вызова произведет вызов метода \_\_call\_\_ объекта wrapper, который в свою   
очередь вызовет метод tracer.\_\_call\_\_.  
Объект wrapper хранит оба экземпляра, дескриптора и класса, поэтому он мо-  
жет передать управление обратно оригинальному экземпляру класса дескрип-  
тора/декоратора. В результате объект wrapper сохраняет подразумеваемый эк-  
земпляр класса, доступный в процессе получения атрибута, и добавляет его   
к списку аргументов, который затем передается методу \_\_call\_\_. Делегирова-  
ние вызова обратно экземпляру дескриптора необходимо в данном случае, что-  
бы для всех вызовов обернутого метода использовался один и тот же счетчик   
вызовов, который находится в экземпляре дескриптора.  
Для достижения того же эффекта можно было бы использовать вложенную   
функцию и хранить информацию о состоянии в области видимости объемлю-

Программирование декораторов функций   
1111  
щей функции. Следующая версия действует точно так же, как и предыдущая,   
но в ней класс и атрибуты объекта заменили вложенная функция и перемен-  
ные в области видимости объемлющей функции, при этом новая версия оказа-  
лась существенно короче:  
class tracer(object):  
 def \_\_init\_\_(self, func): # На этапе декорирования @  
 self.calls = 0 # Сохраняет функцию для последующего вызова  
 self.func = func  
 def \_\_call\_\_(self, \*args, \*\*kwargs): # Вызывается при обращениях к  
 self.calls += 1 # оригинальной функции  
 print(‘call %s to %s’ % (self.calls, self.func.\_\_name\_\_))  
 return self.func(\*args, \*\*kwargs)  
 def \_\_get\_\_(self, instance, owner): # Вызывается при обращении к методу  
 def wrapper(\*args, \*\*kwargs): # Сохраняет оба экземпляра  
 return self(instance, \*args, \*\*kwargs) # Вызовет \_\_call\_\_  
 return wrapper  
Добавьте инструкции print в методы этой версии, чтобы проследить, как вы-  
полняется двухэтапный процесс get/call, и запустите ее, добавив в конец тот   
же программный код самопроверки, как в представленном ранее примере,   
основанном на вложенной функции. В любом случае данный прием, основан-  
ный на дескрипторе, является также более сложным, чем версия с вложенной   
функцией, и поэтому может рассматриваться во вторую очередь� однако он мо-  
жет оказаться полезным в некоторых случаях.  
В оставшейся части этой главы мы достаточно произвольно будем подходить   
к выбору классов или функций для реализации наших декораторов функций,   
когда они будут применяться только к функциям. Некоторые декораторы мо-  
гут не требовать доступа к экземпляру оригинального класса, и в этом случае   
они прекрасно будут работать с функциями и с методами при реализации их   
в виде класса. Примером такого декоратора может служить декоратор static-  
method, имеющийся в языке Python, которому не требуется доступ к подразуме-  
Python, которому не требуется доступ к подразуме-  
, которому не требуется доступ к подразуме-  
ваемому экземпляру класса (в действительности его задача состоит в том, что-  
бы вообще убрать ссылку на экземпляр из вызова метода).  
Суть всего вышеизложенного заключается в том, что, если вам потребуется   
написать декоратор, который может применяться и к функциям, и к методам   
классов, лучше использовать описанный здесь прием на основе вложенной   
функции, а не класс с методом \_\_call\_\_.  
Хронометраж вызовов  
Чтобы полнее ощутить возможности, предлагаемые декораторами функций,   
обратимся к другому примеру их использования. Наш следующий декоратор   
реализует хронометраж выполнения декорируемых функций – время един-  
ственного вызова и накопленное время всех вызовов. В нашем примере деко-  
ратор будет применяться к двум функциям с целью сравнить скорость работы   
генератора списков и встроенной функции map (для сравнения, в главе 20 при-  
водится еще один пример хронометража итерационных альтернатив, не свя-  
занный с декораторами):  
import time  
   
class timer:  
 def \_\_init\_\_(self, func):

1112   
Глава 38. Декораторы   
 self.func = func  
 self.alltime = 0  
 def \_\_call\_\_(self, \*args, \*\*kargs):  
 start = time.clock()  
 result = self.func(\*args, \*\*kargs)  
 elapsed = time.clock() - start  
 self.alltime += elapsed  
 print(‘%s: %.5f, %.5f’ % (self.func.\_\_name\_\_, elapsed, self.alltime))  
 return result  
   
@timer  
def listcomp(N):  
 return [x \* 2 for x in range(N)]  
   
@timer  
def mapcall(N):  
 return map((lambda x: x \* 2), range(N))  
   
result = listcomp(5) # Хронометраж данного вызова, всех вызовов,   
listcomp(50000) # возвращаемое значение  
listcomp(500000)  
listcomp(1000000)  
print(result)  
print(‘allTime = %s’ % listcomp.alltime) # Общее время всех вызовов listcomp  
   
print(‘’)  
result = mapcall(5)  
mapcall(50000)  
mapcall(500000)  
mapcall(1000000)  
print(result)  
print(‘allTime = %s’ % mapcall.alltime) # Общее время всех вызовов mapcall  
   
print(‘map/comp = %s’ % round(mapcall.alltime / listcomp.alltime, 3))  
В данном случае подход без использования декоратора позволил бы использо-  
вать испытуемые функции с проведением хронометража или без него, но если   
бы мы задумали выполнить хронометраж, это усложнило бы сигнатуру вызо-  
ва (вместо того чтобы добавить декоратор перед инструкцией def в одном ме-  
сте, нам потребовалось бы добавить дополнительный программный код везде,   
где выполняется вызов) и у нас не было бы другого простого способа гаранти-  
ровать, что все вызовы испытуемой функции в программе выполняются под   
управлением логики хронометража, кроме как вручную отыскать и изменить   
все вызовы.  
Если запустить этот пример под управлением Python 2.6, он выведет результа-  
Python 2.6, он выведет результа-  
 2.6, он выведет результа-  
ты, как показано ниже:  
listcomp: 0.00002, 0.00002  
listcomp: 0.00910, 0.00912  
listcomp: 0.09105, 0.10017  
listcomp: 0.17605, 0.27622  
[0, 2, 4, 6, 8]  
allTime = 0.276223304917  
   
mapcall: 0.00003, 0.00003  
mapcall: 0.01363, 0.01366  
mapcall: 0.13579, 0.14945

Программирование декораторов функций   
1113  
mapcall: 0.27648, 0.42593  
[0, 2, 4, 6, 8]  
allTime = 0.425933533452  
map/comp = 1.542  
Важное примечание: я не запускал этот пример под управлением Python 3.0,   
потому что, как описывается в главе 14, в этой версии встроенная функция map   
возвращает итератор, а не список, как в версии 2.6. По этой причине нельзя   
напрямую сравнивать производительность функции map в 3.0 с производитель-  
ностью генератора списков (в действительности, тестирование функции map   
в Python 3.0 практически не занимает времени!).  
Если вы пожелаете запустить этот пример под управлением Python 3.0, ис-  
Python 3.0, ис-  
 3.0, ис-  
пользуйте конструкцию list(map()), чтобы принудительно создать список, по-  
добный тому, что возвращает генератор списков, иначе вы будете сравнивать   
теплое с мягким. Однако этого не следует делать в Python 2.6, в противном слу-  
Python 2.6, в противном слу-  
 2.6, в противном слу-  
чае при тестировании функции map будет создаваться два списка вместо одного.  
Следующая версия программного кода позволяет выполнять тестирование   
в обеих версиях Python, 2.6 и 3.0. Однако обратите внимание: хотя эта версия   
позволяет сравнивать производительность генератора списков и функции map   
в любой из версий, 2.6 или 3.0, так как в 3.0 функция range также возвращает   
итератор, тем не менее, результаты, полученные в 2.6 и 3.0, не могут сравни-  
ваться непосредственно:  
...  
import sys  
   
@timer  
def listcomp(N):  
 return [x \* 2 for x in range(N)]  
   
if sys.version\_info[0] == 2:  
 @timer  
 def mapcall(N):  
 return map((lambda x: x \* 2), range(N))  
else:  
 @timer  
 def mapcall(N):  
 return list(map((lambda x: x \* 2), range(N)))  
 ...  
Наконец, как мы узнали в части V, посвященной модулям, если вам потребу-  
ется повторно использовать этот декоратор в других модулях, необходимо бу-  
дет заключить программный код самопроверки, находящийся в конце файла,   
в условную инструкцию, выполняющую проверку \_\_name\_\_ == ‘\_\_main\_\_’, чтобы   
он выполнялся только при запуске файла как сценария, а не во время его им-  
портирования. Однако мы не будем делать этого, потому что собираемся доба-  
вить в нашу реализацию еще одну особенность.  
Добавление аргументов декоратора  
Декоратор timer из предыдущего раздела действует, но было бы неплохо, если   
бы его можно было настраивать, – например, для такого универсального ин-  
струмента, как этот, было бы весьма полезно определять метку для вывода,   
а также включать и отключать вывод трассировочных сообщений. Для этого

1114   
Глава 38. Декораторы   
мы можем использовать аргументы декоратора: при надлежащем использо-  
вании аргументы могут использоваться, как параметры настройки, которые   
могут принимать индивидуальные значения для каждой декорируемой функ-  
ции. Определить метку для вывода, например, можно, как показано ниже:  
def timer(label=’’):  
 def decorator(func):  
 def onCall(\*args): # Аргументы args передаются функции  
 ... # func сохраняется в объемлющей области  
 print(label, ... # label сохраняется в объемлющей области  
 return onCall  
 return decorator # Возвращает фактический декоратор  
   
@timer(‘==>’) # То же, что и listcomp = timer(‘==>’)(listcomp)  
def listcomp(N): ... # Имени listcomp присваивается декоратор  
   
listcomp(...) # В действительности вызывается функция decorator  
Здесь была добавлена область видимости объемлющей функции, в которой   
сохраняется аргумент декоратора, который будет использоваться в последую-  
щих вызовах. После того как функция listcomp будет определена, при обраще-  
нии к ней в действительности будет вызываться функция decorator (результат,   
возвращаемый функцией timer, которая вызывается непосредственно перед   
тем, как будет выполнено декорирование) со значением label, доступным в объ-  
емлющей области видимости. То есть функция timer возвращает декоратор,   
который запоминает аргумент декоратора и оригинальную функцию, и возвра-  
щает вызываемый объект, который, в свою очередь, при последующих вызовах   
будет вызывать оригинальную функцию.  
Мы можем использовать этот подход в нашем декораторе timer, чтобы обеспе-  
чить возможность передачи метки и флага, управляющего выводом трассиро-  
вочных сообщений на этапе декорирования. Следующий пример демонстри-  
рует, как это выглядит в программном коде. Он находится в файле модуля   
с именем mytools.py, благодаря чему его можно импортировать, как обычный   
инструмент:  
import time  
   
def timer(label=’’, trace=True): # Аргументы декоратора: сохраняются  
 class Timer:  
 def \_\_init\_\_(self, func): # На этапе декорирования сохраняется   
 self.func = func # декорируемая функция  
 self.alltime = 0  
 def \_\_call\_\_(self, \*args, \*\*kargs): # При вызове: вызывается оригинал  
 start = time.clock()  
 result = self.func(\*args, \*\*kargs)  
 elapsed = time.clock() - start  
 self.alltime += elapsed  
 if trace:  
 format = ‘%s %s: %.5f, %.5f’  
 values = (label, self.func.\_\_name\_\_, elapsed, self.alltime)  
 print(format % values)  
 return result  
 return Timer  
Почти все, что мы проделали в этом примере, – это заключили оригинальный   
класс Timer в объемлющую функцию, чтобы создать область видимости, в кото-

Программирование декораторов функций   
1115  
рой будут сохраняться аргументы декоратора. Внешняя функция timer вызы-  
вается непосредственно перед операцией декорирования, она просто возвраща-  
ет класс Timer, который будет играть роль фактического декоратора. В момент   
декорирования создается экземпляр класса Timer, который запоминает саму   
декорируемую функцию, но при этом ему остаются доступными аргументы де-  
коратора, находящиеся в области видимости объемлющей функции.  
На этот раз вместо того чтобы добавлять программный код самотестирования   
в данный файл, мы организуем вызов декоратора из другого файла. Ниже при-  
водится исходный текст клиентского модуля, использующего наш декоратор   
timer� этот модуль сохранен в файле testseqs.py и применяет декоратор к группе   
итерационных альтернатив:   
from mytools import timer  
   
@timer(label=’[CCC]==>’)  
def listcomp(N): # То же, что и listcomp = timer(...)(listcomp)  
 return [x \* 2 for x in range(N)] # listcomp(...) вызовет Timer.\_\_call\_\_  
   
@timer(trace=True, label=’[MMM]==>’)  
def mapcall(N):  
 return map((lambda x: x \* 2), range(N))  
   
for func in (listcomp, mapcall):  
 print(‘’)  
 result = func(5) # Хронометраж вызова, всех вызовов, возвращаемое значение  
 func(50000)   
 func(500000)  
 func(1000000)  
 print(result)  
 print(‘allTime = %s’ % func.alltime) # Общее время всех вызовов  
   
print(‘map/comp = %s’ % round(mapcall.alltime / listcomp.alltime, 3))  
Напомню еще раз: если вы предполагаете запускать этот пример под управ-  
лением Python 3.0, оберните вызов функции map вызовом функции list. Если   
запустить этот пример под управлением Python 2.6, он выведет следующие ре-  
Python 2.6, он выведет следующие ре-  
 2.6, он выведет следующие ре-  
зультаты – для каждой декорированной функции теперь выводится своя мет-  
ка, определенная как аргумент декоратора:  
[CCC]==> listcomp: 0.00003, 0.00003  
[CCC]==> listcomp: 0.00640, 0.00643  
[CCC]==> listcomp: 0.08687, 0.09330  
[CCC]==> listcomp: 0.17911, 0.27241  
[0, 2, 4, 6, 8]  
allTime = 0.272407666337  
   
[MMM]==> mapcall: 0.00004, 0.00004  
[MMM]==> mapcall: 0.01340, 0.01343  
[MMM]==> mapcall: 0.13907, 0.15250  
[MMM]==> mapcall: 0.27907, 0.43157  
[0, 2, 4, 6, 8]  
allTime = 0.431572169089  
map/comp = 1.584  
Кроме того, мы можем протестировать декоратор в интерактивном режиме,   
чтобы увидеть, как действуют параметры настройки:

1116   
Глава 38. Декораторы   
>>> from mytools import timer  
>>> @timer(trace=False) # Вывод сообщений отключен,   
... def listcomp(N): # накапливается общее время  
... return [x \* 2 for x in range(N)]  
...  
>>> x = listcomp(5000)  
>>> x = listcomp(5000)  
>>> x = listcomp(5000)  
>>> listcomp  
<mytools.Timer instance at 0x025C77B0>  
>>> listcomp.alltime  
0.0051938863738243413  
   
>>> @timer(trace=True, label=’\t=>’) # Вывод сообщений включен  
... def listcomp(N):  
... return [x \* 2 for x in range(N)]  
...  
>>> x = listcomp(5000)  
 => listcomp: 0.00155, 0.00155  
>>> x = listcomp(5000)  
 => listcomp: 0.00156, 0.00311  
>>> x = listcomp(5000)  
 => listcomp: 0.00174, 0.00486  
>>> listcomp.alltime  
0.0048562736325408196  
Этот декоратор, реализующий хронометраж вызовов функций, может приме-  
няться к любым функциям, внутри модулей или в интерактивной оболочке.   
Другими словами, он автоматически превращается в универсальный инстру-  
мент хронометража функций в наших сценариях. Другой пример использо-  
вания аргументов в декораторах вы найдете в разделе «Реализация частных   
атрибутов» ниже, и еще один – в разделе «Простой декоратор проверки значе-  
ний позиционных аргументов на вхождение в заданный диапазон» ниже.  
Методы хронометрирования: Декоратор timer, представленный   
в этом разделе, может применяться к любым функциям, а чтобы   
его можно было применять к методам классов, требуются совсем   
незначительные переделки. В двух словах: как было показано   
в разделе «Ошибки при использовании классов I: декорирование   
методов классов», выше, для этого нужно заменить вложенный   
класс вложенной функцией. Однако поскольку это тема одного   
из контрольных вопросов в конце главы, я не буду здесь приво-  
дить полное решение.  
Программирование декораторов классов  
До сих пор мы создавали декораторы, которые управляют вызовами функций,   
но, как мы уже знаем, в Python 2.6 и 3.0 возможности декораторов были рас-  
Python 2.6 и 3.0 возможности декораторов были рас-  
 2.6 и 3.0 возможности декораторов были рас-  
ширены, и теперь они могут применяться и к классам. Как описывалось выше,   
в отличие от декораторов функций, декораторы классов применяются к клас-  
сам – они могут использоваться для управления самими классами или для пе-  
рехвата операций создания экземпляров и реализации управления ими. Кроме

Программирование декораторов классов   
1117  
того, подобно декораторам функций декораторы классов в действительности   
являются всего лишь синтаксическим подсластителем, хотя принято считать,   
что они делают намерения программиста более очевидными и минимизируют   
количество ошибочных вызовов.  
Классы одиночных экземпляров  
Поскольку декораторы классов способны перехватывать операции создания   
экземпляров, они могут использоваться для управления всеми экземплярами   
класса или расширять интерфейс каждого экземпляра в отдельности. Для де-  
монстрации сказанного ниже приводится первый пример декоратора класса,   
который реализует первый случай – управление всеми экземплярами клас-  
са. Этот пример реализует классический шаблон проектирования singleton   
(одиночка1), когда в каждый конкретный момент во время работы программы   
может существовать не более одного экземпляра класса. Функция singleton   
в примере определяет и возвращает другую функцию, которая управляет эк-  
земплярами, а применение синтаксиса @ автоматически обертывает декорируе-  
мый класс этой функцией:  
instances = {}  
def getInstance(aClass, \*args): # Управляет глобальной таблицей  
 if aClass not in instances: # Добавьте \*\*kargs, чтобы обрабатывать   
 # именованные аргументы  
 instances[aClass] = aClass(\*args) # По одному элементу словаря для   
 return instances[aClass] # каждого класса  
   
def singleton(aClass): # На этапе декорирования  
 def onCall(\*args): # На этапе создания экземпляра  
 return getInstance(aClass, \*args)  
 return onCall  
Чтобы воспользоваться этой функцией, достаточно применить ее как декора-  
тор к классу, количество экземпляров которого должно быть не больше одного:  
@singleton # Person = singleton(Person)  
class Person: # Присвоит onCall имени Person   
 def \_\_init\_\_(self, name, hours, rate): # onCall сохранит Person  
 self.name = name  
 self.hours = hours  
 self.rate = rate  
 def pay(self):  
 return self.hours \* self.rate  
   
@singleton # Spam = singleton(Spam)  
class Spam: # Присвоит onCall имени Spam   
 def \_\_init\_\_(self, val): # onCall сохранит Spam  
 self.attr = val  
   
bob = Person(‘Bob’, 40, 10) # В действительности вызовет onCall  
print(bob.name, bob.pay())  
   
sue = Person(‘Sue’, 50, 20) # Тот же самый единственный объект  
1   
В русскоязычной литературе название шаблона «singleton» часто подается без пере-  
вода, иногда используется название «синглетон», иногда – «одиночка». – Прим.   
перев.

1118   
Глава 38. Декораторы   
print(sue.name, sue.pay())  
   
X = Spam(42) # Один экземпляр Person, один – Spam  
Y = Spam(99)  
print(X.attr, Y.attr)  
Теперь, когда позднее классы Person и Spam будут использоваться для создания   
экземпляров, логика декоратора, обертывающая операцию создания экзем-  
пляров, передаст управление функции onCall, которая в свою очередь вызовет   
функцию getInstance, создающую и возвращающую единственный экземпляр   
класса независимо от количества попыток создать экземпляр. Ниже приводят-  
ся результаты работы этого примера:  
Bob 400  
Bob 400  
42 42  
Интересно отметить, что здесь можно было бы предложить более автономное   
решение, если бы имелась возможность использовать инструкцию nonlocal   
(доступную в версии Python 3.0 и выше), позволяющую изменять переменные   
в области видимости объемлющей функции. Следующее альтернативное ре-  
шение достигает желаемого эффекта за счет использования отдельной области   
видимости для каждого класса вместо единой глобальной таблицы, в которой   
отводится по одному элементу для каждого класса:  
def singleton(aClass): # На этапе декорирования  
 instance = None  
 def onCall(\*args): # На этапе создания экземпляра  
 nonlocal instance # nonlocal доступна в 3.0 и выше  
 if instance == None:  
 instance = aClass(\*args) # По одной области видимости   
 return instance # на каждый класс  
 return onCall  
Эта версия действует точно так же, но она уже не зависит от имен в глобальной   
области видимости за пределами декоратора. Имеется возможность реализо-  
вать подобное автономное решение, которое будет действовать в обеих верси-  
ях Python, 2.6 и 3.0, – с применением класса. В следующем примере вместо   
областей видимости или глобальной таблицы создается по одному экземпляру   
на каждый класс. Он действует точно так же, как и две другие версии (Фак-  
тически он опирается на тот же самый шаблон проектирования, который мы   
увидим далее в разделе «Ошибки при использовании классов II: сохранение   
множества экземпляров» ниже. Здесь нам требуется получить единственный   
экземпляр, но это лишь частный случай):  
class singleton:  
 def \_\_init\_\_(self, aClass): # На этапе декорирования  
 self.aClass = aClass  
 self.instance = None  
 def \_\_call\_\_(self, \*args): # На этапе создания экземпляра  
 if self.instance == None:  
 self.instance = self.aClass(\*args) # По одному экземпляру на класс  
 return self.instance  
Чтобы превратить этот декоратор в полноценный универсальный инструмент,   
сохраните его в виде файла модуля, оберните программный код самопроверки

Программирование декораторов классов   
1119  
условной инструкцией, проверяющей значение \_\_name\_\_, и добавьте поддержку   
именованных аргументов в методе создания экземпляров с помощью синтак-  
сической конструкции \*\*kargs (я оставляю это вам в качестве самостоятельного   
упражнения).  
Изменение интерфейсов объектов  
Пример в предыдущем разделе продемонстрировал возможность использова-  
ния декоратора классов для управления всеми экземплярами класса. Другой   
распространенный случай использования декораторов классов – расширение   
интерфейса каждого отдельного экземпляра. Декораторы классов могут добав-  
лять к экземплярам обертывающий уровень логики, которая некоторым спо-  
собом организует доступ к интерфейсам.  
Например, в главе 30 было продемонстрирован способ обертывания всего ин-  
терфейса встроенного экземпляра с помощью метода \_\_getattr\_\_ перегрузки   
операторов с целью реализовать шаблон делегирования. В предыдущей главе   
мы видели похожие примеры управления атрибутами. Напомню, что метод   
\_\_getattr\_\_ вызывается при попытке получить значение неопределенного атри-  
бута, – мы могли бы использовать его, чтобы перехватывать вызовы методов   
в управляющем классе и делегировать их встроенному объекту.  
Для справки ниже приводится оригинальный пример, не использующий деко-  
раторы, где демонстрируется работа с объектами двух встроенных типов:  
class Wrapper:  
 def \_\_init\_\_(self, object):  
 self.wrapped = object # Сохранить объект  
 def \_\_getattr\_\_(self, attrname):  
 print(‘Trace:’, attrname) # Сообщить о попытке получить значение  
 return getattr(self.wrapped, attrname) # Делегировать выполнение   
 # операции  
   
>>> x = Wrapper([1,2,3]) # Обернуть список  
>>> x.append(4) # Делегирует операцию методу списка  
Trace: append  
>>> x.wrapped # Вывести элементы  
[1, 2, 3, 4]  
   
>>> x = Wrapper({“a”: 1, “b”: 2}) # Обернуть словарь  
>>> list(x.keys()) # Делегирует операцию методу словаря  
Trace: keys # Используйте list() в 3.0  
[‘a’, ‘b’]  
В этом примере класс Wrapper перехватывает попытки обращения к любым   
атрибутам обернутого объекта, выводит трассировочные сообщения и исполь-  
зует встроенную функцию getattr для передачи запросов обернутому объекту.   
В частности, он отслеживает попытки доступа к атрибутам, которые выпол-  
няются за пределами класса обернутого объекта, – обращения, выполняемые   
внутри обернутого объекта, не перехватываются и выполняются, как пред-  
усмотрено реализацией. Такая модель обертывания всего интерфейса отли-  
чается от поведения декораторов функций, которые обертывают один опреде-  
ленный метод.  
Декораторы классов обеспечивают альтернативный и не менее удобный спо-  
соб реализовать прием, основанный на применении метода \_\_getattr\_\_, для

1120   
Глава 38. Декораторы   
обертывания всего интерфейса. В Python 2.6 и 3.0, например, предыдущий   
пример класса может быть реализован в виде декоратора класса, который вме-  
шивается в процесс создания экземпляра, вместо того чтобы передавать пред-  
варительно созданный экземпляр конструктору класса-обертки (а также до-  
полнен конструкцией \*\*kargs, дающей возможность принимать именованные   
аргументы, и счетчиком общего числа обращений):  
def Tracer(aClass): # На этапе декорирования @  
 class Wrapper:  
 def \_\_init\_\_(self, \*args, \*\*kargs): # На этапе создания экземпляра  
 self.fetches = 0  
 self.wrapped = aClass(\*args, \*\*kargs) # Использует имя в   
 def \_\_getattr\_\_(self, attrname): # объемлющей области   
 print(‘Trace: ‘ + attrname) # Перехватывает обращения ко   
 self.fetches += 1 # всем атриб-м, кроме своих   
 return getattr(self.wrapped, attrname) # Делегирует обращения   
 return Wrapper # обернутому объекту  
   
@Tracer  
class Spam: # Spam = Tracer(Spam)  
 def display(self): # Имени Spam присваивается экземпляр Wrapper  
 print(‘Spam!’ \* 8)  
   
@Tracer  
class Person: # Person = Tracer(Person)  
 def \_\_init\_\_(self, name, hours, rate): # Wrapper сохраняет Person  
 self.name = name  
 self.hours = hours  
 self.rate = rate  
 def pay(self): # Доступ извне перехватывается  
 return self.hours \* self.rate # Изнутри – не перехватывается  
   
food = Spam() # Вызовет Wrapper()  
food.display() # Вызовет \_\_getattr\_\_  
print([food.fetches])  
   
bob = Person(‘Bob’, 40, 50) # bob – в действительности экземпляр Wrapper  
print(bob.name) # экземпляр Person встраивается во Wrapper  
print(bob.pay())  
   
print(‘’)  
sue = Person(‘Sue’, rate=100, hours=60) # sue – другой экземпляр Wrapper  
print(sue.name) # с другим экземпляром Person  
print(sue.pay())  
   
print(bob.name) # Состояние bob отличается  
print(bob.pay())  
print([bob.fetches, sue.fetches]) # Атрибуты Wrapper не отслеживаются  
Важно отметить, что этот пример существенно отличается от декоратора   
tracer, с которым мы встречались выше. В разделе «Программирование деко-  
раторов функций» выше мы рассматривали декораторы, которые позволяют   
отслеживать и хронометрировать отдельные функции или методы. В отличие   
от него представленный здесь декоратор класса, перехватывающий операции   
создания экземпляров, позволяет отслеживать попытки обращения ко всему   
интерфейсу объекта, то есть попытки доступа к любым его атрибутам.

Программирование декораторов классов   
1121  
Ниже приводятся результаты работы этого сценария под управлением обеих   
версий Python, 2.6 и 3.0: попытки получить значения атрибутов экземпляров   
обоих классов, Spam и Person, запускают логику метода \_\_getattr\_\_ в классе Wrap-  
per, потому что в действительности объекты food и bob являются экземплярами   
класса Wrapper благодаря тому, что операции создания экземпляров были пере-  
хвачены декоратором:  
Trace: display  
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!  
[1]  
Trace: name  
Bob  
Trace: pay  
2000  
   
Trace: name  
Sue  
Trace: pay  
6000  
Trace: name  
Bob  
Trace: pay  
2000  
[4, 2]  
Обратите внимание, что в предыдущем примере декорировался пользователь-  
ский класс. Однако точно так же, как и в оригинальном примере в главе 30, мы   
можем использовать декоратор для обертывания встроенных типов, таких как   
list, если при этом мы создадим подкласс, что позволит нам применить син-  
таксис декораторов, или выполним декорирование вручную – синтаксис деко-  
раторов требует, чтобы вслед за строкой с именем декоратора, начинающегося   
символом @, следовала инструкция class.  
В следующем примере x в действительности является экземпляром класса   
Wrapper из-за того, что было выполнено косвенное декорирование (я переместил   
реализацию декоратора класса в файл модуля tracer.py, чтобы его можно было   
повторно использовать, как показано ниже):  
>>> from tracer import Tracer # Декоратор был перемещен в файл модуля  
>>> @Tracer  
... class MyList(list): pass # MyList = Tracer(MyList)  
>>> x = MyList([1, 2, 3]) # Вызовет Wrapper()  
>>> x.append(4) # Вызовет \_\_getattr\_\_, append  
Trace: append  
>>> x.wrapped  
[1, 2, 3, 4]  
   
>>> WrapList = Tracer(list) # Декорирование выполняется вручную  
>>> x = WrapList([4, 5, 6]) # В противном случае потребовалось бы   
>>> x.append(7) # определить подкласс  
Trace: append  
>>> x.wrapped  
[4, 5, 6, 7]  
Прием на основе декоратора позволяет поместить процедуру создания экзем-  
пляра непосредственно в декоратор и тем самым избавиться от необходимости

1122   
Глава 38. Декораторы   
создавать экземпляр предварительно. Даже при том, что разница на первый   
взгляд кажется незначительной, тем не менее, декораторы позволяют сохра-  
нить привычный синтаксис создания экземпляров и одновременно пользо-  
ваться всеми преимуществами декораторов. Вместо того чтобы передавать все   
созданные экземпляры конструктору класса-обертки вручную, нам достаточно   
всего лишь добавить декоратор перед определением класса:  
@Tracer # Подход на основе декоратора  
class Person: ...  
bob = Person(‘Bob’, 40, 50)  
sue = Person(‘Sue’, rate=100, hours=60)  
   
class Person: ... # Подход без использования декоратора  
bob = Wrapper(Person(‘Bob’, 40, 50))  
sue = Wrapper(Person(‘Sue’, rate=100, hours=60))  
Представьте, что вам требуется создать множество экземпляров класса. При-  
, что вам требуется создать множество экземпляров класса. При-  
что вам требуется создать множество экземпляров класса. При-  
 вам требуется создать множество экземпляров класса. При-  
вам требуется создать множество экземпляров класса. При-  
 требуется создать множество экземпляров класса. При-  
требуется создать множество экземпляров класса. При-  
 создать множество экземпляров класса. При-  
создать множество экземпляров класса. При-  
 множество экземпляров класса. При-  
множество экземпляров класса. При-  
 экземпляров класса. При-  
экземпляров класса. При-  
 класса. При-  
класса. При-  
. При-  
При-  
менение декораторов в подобных ситуациях принесет вам чистую победу как   
в смысле размера программного кода, так и в смысле простоты его сопрово-  
ждения.  
Примечание,  касающееся  различий  между  версиями: Как мы   
узнали в главе 37, метод \_\_getattr\_\_ будет перехватывать обра-  
щения к методам перегрузки операторов, таким как \_\_str\_\_ и \_\_  
repr\_\_, в Python 2.6, но не в Python 3.0.  
В Python 3.0 экземпляры классов по умолчанию наследуют не-  
Python 3.0 экземпляры классов по умолчанию наследуют не-  
 3.0 экземпляры классов по умолчанию наследуют не-  
которые (но не все) из этих имен от класса (в действительности,   
от суперкласса object, который назначается автоматически), по-  
тому что в этой версии Python все классы являются классами   
«нового стиля». Кроме того, в версии 3.0 неявные обращения   
к атрибутам, которые производятся встроенными операциями,   
такими как вывод и +, не приводят к вызову метода \_\_getattr\_\_   
(или родственного ему метода \_\_getattribute\_\_). Для классов но-  
вого стиля поиск таких методов начинается с классов, при этом   
обычный этап поиска в экземпляре пропускается полностью.  
В данном случае это означает, что трассировщик, основанный   
на использовании метода \_\_getattr\_\_, будет автоматически пере-  
хватывать и делегировать вызовы методов перегрузки операто-  
ров в версии 2.6, но не будет в версии 3.0. Чтобы заметить это,   
попробуйте вывести объект x непосредственно в конце предыду-  
щего интерактивного сеанса – в версии 2.6 будет отмечено обра-  
щение к атрибуту \_\_repr\_\_ и интерпретатор выведет список, как   
и ожидается, но в версии 3.0 обращение к атрибуту \_\_repr\_\_ за-  
мечено не будет, а для вывода списка будет использован метод   
вывода по умолчанию для класса Wrapper:  
>>> x # 2.6  
Trace: \_\_repr\_\_  
[4, 5, 6, 7]  
>>> x # 3.0  
<tracer.Wrapper object at 0x026C07D0>  
Чтобы получить те же результаты в Python 3.0, методы пере-  
Python 3.0, методы пере-  
 3.0, методы пере-  
грузки операторов требуется переопределить в классе-обертке

Программирование декораторов классов   
1123  
вручную, с помощью каких-либо инструментов или за счет их   
определения в суперклассах. Одинаково обрабатываться в обеих   
версиях будут только атрибуты с простыми именами. Мы еще   
раз столкнемся с этим различием между версиями ниже в этой   
же главе, когда будем рассматривать декоратор Private.  
Ошибки при использовании классов II:   
сохранение множества экземпляров  
Любопытно отметить, что функция декоратора в этом примере практически   
полностью реализована не как функция, а как класс с соответствующим мето-  
дом перегрузки операторов. Ниже приводится несколько упрощенная альтер-  
натива, действующая похожим способом, потому что ее метод \_\_init\_\_ вызыва-  
ется в момент применения декоратора @ к классу, а метод \_\_call\_\_ вызывается   
при создании экземпляра декорируемого класса. На этот раз наши объекты   
в действительности являются экземплярами класса Tracer, и здесь мы, по сути,   
лишь заменили объемлющую область видимости атрибутами экземпляра:  
class Tracer:  
 def \_\_init\_\_(self, aClass): # На этапе декорирования @  
 self.aClass = aClass # Использует атрибуты экземпляра  
 def \_\_call\_\_(self, \*args): # На этапе создания экземпляра  
 self.wrapped = self.aClass(\*args) # ЕДИНСТВЕННЫЙ (ПОСЛЕДНИЙ)   
 return self # ЭКЗЕМПЛЯР ДЛЯ КАЖДОГО КЛАССА!  
 def \_\_getattr\_\_(self, attrname):  
 print(‘Trace: ‘ + attrname)  
 return getattr(self.wrapped, attrname)  
   
@Tracer # Вызовет \_\_init\_\_  
class Spam: # То же, что и Spam = Tracer(Spam)  
 def display(self):  
 print(‘Spam!’ \* 8)  
   
...  
food = Spam() # Вызовет \_\_call\_\_  
food.display() # Вызовет \_\_getattr\_\_  
Однако как мы уже видели ранее, эта альтернатива, основанная исключитель-  
но на использовании класса, может применяться к множеству классов, как   
и прежде, но она не обеспечивает возможность создания множества экземпля-  
ров для каждого конкретного класса: каждая операция создания экземпляра   
будет вызывать метод \_\_call\_\_, который будет затирать ссылку на предыдущий   
экземпляр. В результате экземпляр класса Tracer будет сохранять единствен-  
ный экземпляр декорируемого класса, созданный последним. Поэксперимен-  
тируйте с этим примером самостоятельно, чтобы понять, как это происходит,   
а ниже приводится пример, иллюстрирующий проблему:  
@Tracer  
class Person: # Person = Tracer(Person)  
 def \_\_init\_\_(self, name): # Имени Person присваивается Wrapper  
 self.name = name  
   
bob = Person(‘Bob’) # bob – экземпляр класса Wrapper  
print(bob.name) # экземпляр класса Wrapper содержит экземпляр класса Person  
Sue = Person(‘Sue’)

1124   
Глава 38. Декораторы   
print(sue.name) # объект sue затер объект bob  
print(bob.name) # ОЙ: теперь Боба зовут ‘Sue’!  
Если запустить этот пример, он выведет следующие результаты – поскольку   
в этой реализации трассировщик может сохранять только один экземпляр, по-  
пытка создать второй экземпляр затрет первый:  
Trace: name  
Bob  
Trace: name  
Sue  
Trace: name  
Sue  
Проблема здесь заключается в некорректной реализации сохранения  инфор-  
мации о состоянии – интерпретатор создает по одному экземпляру декоратора   
для каждого класса, а не для каждого экземпляра класса, поэтому сохраняется   
только последний созданный экземпляр. Решение этой проблемы точно такое   
же, как и в предыдущем разделе, описывающем ошибки при использовании   
классов для декорирования методов, – отказаться от реализации декораторов   
в виде классов.  
Предыдущая версия декоратора Tracer на основе функции поддерживает воз-  
можность создания множества экземпляров, потому что для каждого созда-  
ваемого экземпляра создается новый экземпляр Wrapper, что исключает воз-  
можность затирания единственного общего экземпляра Tracer. Оригинальная   
версия примера, не использующая декоратор, обеспечивает возможность соз-  
дания множества экземпляров по той же причине. Декораторы могут быть не   
только чрезвычайно удобными, но и порождать трудноуловимые ошибки!  
Декораторы и управляющие функции  
Независимо от подобных тонкостей пример декоратора классов Tracer все так   
же целиком опирается на использование метода \_\_getattr\_\_ для перехвата об-  
ращений к обернутому и встроенному объекту экземпляра. Как мы уже по-  
няли ранее, все, чего мы достигли, – это лишь перенесли операцию создания   
экземпляра в класс, вместо того, чтобы передавать уже готовый экземпляр   
управляющей функции. В оригинальном примере трассировщика, не исполь-  
зующем декоратор, мы могли бы просто немного иначе реализовать создание   
экземпляров:  
class Spam: # Версия, не использующая декоратор  
 ... # Это может быть любой класс  
food = Wrapper(Spam()) # Специальный синтаксис создания экземпляра  
   
@Tracer  
class Spam: # Версия на основе декоратора  
 ... # Обязательный синтаксис @ определения класса  
food = Spam() # Обычный синтаксис создания экземпляра  
По сути декораторы классов перекладывают требования к синтаксису оформ-  
ления с операции создания экземпляра на саму инструкцию class. То же спра-  
ведливо и для примера реализации шаблона singleton, приводившегося выше   
в этом разделе, – вместо того, чтобы декорировать класс и использовать обыч-  
ную конструкцию создания экземпляра, мы могли бы просто передать класс   
и аргументы конструктора управляющей функции:

Программирование декораторов классов   
1125  
instances = {}  
def getInstance(aClass, \*args):  
 if aClass not in instances:  
 instances[aClass] = aClass(\*args)  
 return instances[aClass]  
   
bob = getInstance(Person, ‘Bob’, 40, 10) # Вместо: bob = Person(‘Bob’, 40, 10)  
Как один из вариантов мы могли бы воспользоваться возможностями интро-  
спекции, чтобы определить класс из уже созданного экземпляра (предполага-  
ется допустимость такого способа создания экземпляра):  
instances = {}  
def getInstance(object):  
 aClass = object.\_\_class\_\_  
 if aClass not in instances:  
 instances[aClass] = object  
 return instances[aClass]  
   
bob = getInstance(Person(‘Bob’, 40, 10)) # Вместо: bob = Person(‘Bob’, 40, 10)  
То же остается верным и для декораторов функций, таких как tracer, который   
был написан нами ранее: вместо того, чтобы декорировать функцию дополни-  
тельной логикой, перехватывающей последующие вызовы, мы могли бы про-  
сто передать требуемую функцию и ее аргументы управляющей функции, ко-  
торая будет переадресовывать вызовы:  
def func(x, y): # Версия, не использующая декоратор  
 ... # def tracer(func, args): ... func(\*args)  
result = tracer(func, (1, 2)) # Специальный синтаксис вызова  
   
@tracer  
def func(x, y): # Версия на основе декоратора  
 ... # Присваивает имени func = tracer(func)  
result = func(1, 2) # Обычный синтаксис вызова  
Подход, основанный на применении управляющих функций, как в данном   
примере, переносит бремя использования синтаксиса декораторов с определе-  
ний функций и классов на конструкции вызова.  
Зачем нужны декораторы? (Еще раз)  
Итак, почему я только что показал вам способы реализации шаблона singleton   
без применения декораторов? Как я уже упоминал в начале этой главы, декора-  
торы дарят нам компромиссные решения. Безусловно синтаксис имеет немало-  
важное значение, но все мы слишком часто забываем задать вопрос «зачем»,   
когда сталкиваемся с новыми инструментами. Теперь, когда мы увидели, как   
действуют декораторы, давайте отступим на шаг назад, чтобы окинуть взгля-  
дом получившуюся картину.  
Подобно большинству других особенностей языка декораторы имеют свои «за»   
и «против». Например, в колонке «против» декораторов классов можно указать   
два потенциальных недостатка:  
Изменение типа  
Как мы уже видели, после обертывания декорируемая функция или класс   
не сохраняет свой первоначальный  тип – имени оригинальной функции   
или класса присваивается объект-обертка, что может иметь значение в про-

1126   
Глава 38. Декораторы   
граммах, где выполняется проверка типов объектов. В примере реализации   
шаблона singleton оба подхода, на основе декоратора и на основе управляю-  
singleton оба подхода, на основе декоратора и на основе управляю-  
 оба подхода, на основе декоратора и на основе управляю-  
щей функции, сохраняют оригинальный тип экземпляров, а в примере реа-  
лизации трассировщика ни один из подходов не обеспечивает сохранение   
типа, потому что требуется обертывание оригинальных объектов.  
Дополнительные вызовы  
Дополнительный уровень обертывающей логики, добавляемый при декори-  
ровании, привносит дополнительную нагрузку, отрицательно влияющую   
на производительность, в виде дополнительных вызовов, которые произво-  
дятся при каждом обращении к декорированному объекту. Вызовы функ-  
ций – это достаточно затратные операции, поэтому декорирование обертка-  
ми может привести к снижению производительности программы. В приме-  
ре реализации трассировщика доступ к любым атрибутам осуществляется   
через обертывающую логику – в примере реализации шаблона singleton   
дополнительные вызовы не производятся благодаря тому, что сохраняется   
оригинальный тип класса.  
Похожие проблемы наблюдаются и в декораторах функций: в обоих случаях,   
как при декорировании, так и при использовании управляющих функций, вы-  
полняются дополнительные вызовы. Кроме того, при декорировании обычно   
изменяется тип оригинальной функции (но он не изменяется при использова-  
нии управляющей функции).  
И все же, ни одна из этих проблем не является слишком серьезной. Для боль-  
шинства программ проблема изменения типа вряд ли будет иметь хоть какое-  
то значение, а потеря скорости за счет дополнительных вызовов будет ничтож-  
ной. Кроме того, последняя проблема наблюдается только при использовании   
оберток и ее часто легко ликвидировать, просто убрав декоратор, когда требу-  
ется обеспечить максимальную производительность. Этой же проблеме под-  
вержены решения, не использующие декораторы, которые добавляют оберты-  
вающую логику (включая метаклассы, как мы увидим в главе 39).  
Напротив, как мы видели в начале этой главы, декораторы обладают тремя   
важными достоинствами. В сравнении с решениями на основе управляющих   
(или «вспомогательных») функций, представленными в предыдущем разделе,   
декораторы могут предложить:  
Явный синтаксис  
Декораторы делают наращивание функциональных возможностей более   
явным и очевидным. Синтаксис @ декораторов заметнее, чем специальный   
программный код в вызовах, которые могут быть разбросаны по всему фай-  
 код в вызовах, которые могут быть разбросаны по всему фай-  
код в вызовах, которые могут быть разбросаны по всему фай-  
 в вызовах, которые могут быть разбросаны по всему фай-  
в вызовах, которые могут быть разбросаны по всему фай-  
 вызовах, которые могут быть разбросаны по всему фай-  
вызовах, которые могут быть разбросаны по всему фай-  
, которые могут быть разбросаны по всему фай-  
которые могут быть разбросаны по всему фай-  
 могут быть разбросаны по всему фай-  
могут быть разбросаны по всему фай-  
 быть разбросаны по всему фай-  
быть разбросаны по всему фай-  
 разбросаны по всему фай-  
разбросаны по всему фай-  
 по всему фай-  
по всему фай-  
 всему фай-  
всему фай-  
 фай-  
фай-  
лу. В наших примерах реализации трассировщика и шаблона singleton, на-  
. В наших примерах реализации трассировщика и шаблона singleton, на-  
В наших примерах реализации трассировщика и шаблона singleton, на-  
singleton, на-  
, на-  
пример, строка с декоратором выглядит более заметной, чем дополнитель-  
ный программный код в вызовах. Кроме того, декораторы позволяют ис-  
пользовать обычный синтаксис вызовов функций и создания экземпляров,   
знакомый всем программистам на языке Python.  
Сопровождение программного кода  
Декораторы позволяют избежать использования избыточного программно-  
го кода при оформлении каждого вызова функции или класса. Благодаря   
тому, что декораторы появляются в программном коде всего один раз, не-  
посредственно перед определением класса или функции, они устраняют   
избыточность и упрощают сопровождение программы в будущем. При ис-

Непосредственное управление функциями и классами   
1127  
пользовании управляющих функций в наших примерах реализации трас-  
сировщика и шаблона singleton нам потребовалось бы использовать специ-  
singleton нам потребовалось бы использовать специ-  
 нам потребовалось бы использовать специ-  
альный программный код для оформления каждого вызова, что потребова-  
ло бы от нас лишних действий при создании программы и при ее изменении   
в будущем.  
Последовательность  
Декораторы снижают вероятность того, что программист забудет задей-  
ствовать требуемую обертывающую логику. Это следует в основном из   
двух предыдущих достоинств – благодаря тому, что декораторы являются   
более очевидными и появляются в программном коде всего один раз, не-  
посредственно перед декорируемыми объектами, они обеспечивают более   
непротиворечивый и единообразный способ, чем специализированный про-  
граммный код, который должен включаться в каждый вызов. В примере   
реализации шаблона singleton, например, легко можно забыть добавить   
специальный программный код во все операции создания экземпляров, что   
приведет к невозможности гарантировать существование единственного эк-  
земпляра класса.  
Кроме того, декораторы обеспечивают инкапсуляцию программного кода, сни-  
жая его избыточность и минимизируя усилия по его сопровождению в буду-  
щем. Тот же эффект позволяют получить и другие средства структурирования   
программного кода, однако декораторы являются естественным средством при   
решении задач расширения функциональных возможностей.  
Однако ни одно из этих преимуществ не делает декораторы обязательными   
к применению. В конечном счете, использовать или не использовать декорато-  
ры – это в значительной степени вопрос выбора стиля. И, тем не менее, боль-  
шинство программистов считают их важным приобретением, особенно когда   
они рассматриваются как средство корректного использования библиотек   
и прикладных интерфейсов.  
Я могу напомнить похожие аргументы, приводившиеся за и против исполь-  
зования методов-конструкторов в классах – до появления метода \_\_init\_\_ тот   
же самый эффект можно было получить, вызывая метод экземпляра вручную   
в процессе его создания (например, X=Class().init()). Однако со временем, не-  
взирая на устоявшийся стиль оформления, синтаксис \_\_init\_\_ приобрел боль-  
шую популярность благодаря своей очевидности, последовательности и про-  
стоте в сопровождении. Судить конечно вам, но, как мне кажется, декораторы   
несут в себе множество похожих преимуществ.  
Непосредственное управление   
функциями и классами  
Большинство примеров в этой главе разрабатывалось с целью обеспечить   
перехват вызовов функций и операций создания экземпляров. Это наиболее   
типичная роль декораторов, но она не единственная. Поскольку декораторы   
запускаются, когда создается новая функция или класс, они могут также ис-  
пользоваться не только для перехвата обращений к этим функциям и классам,   
но и для управления самими объектами функций и классов.  
Представьте, например, что вам требуется зарегистрировать методы или клас-  
сы, используемые приложением, для последующей обработки (возможно, эти

1128   
Глава 38. Декораторы   
объекты будут вызываться приложением в ответ на какие-либо события). Вы   
могли бы написать функцию регистрации, которую можно было бы вызывать   
вручную после того, как объекты будут определены, однако декораторы позво-  
ляют сделать ваши намерения более явными.  
Следующий простой пример реализует эту идею, определяя декоратор, кото-  
рый может применяться к функциям или к классам, чтобы добавить объект   
в реестр на основе словаря. Так как декоратор возвращает сам объект, а не   
обертку, он не оказывает влияния на последующие вызовы этого объекта:  
# Регистрация декорируемых объектов  
   
registry = {}  
def register(obj): # Декоратор функций и классов  
 registry[obj.\_\_name\_\_] = obj # Добавить в реестр  
 return obj # Возвращает сам объект obj, а не обертку  
   
@register  
def spam(x):  
 return(x \*\* 2) # spam = register(spam)  
   
@register  
def ham(x):  
 return(x \*\* 3)  
   
@register  
class Eggs: # Eggs = register(Eggs)  
 def \_\_init\_\_(self, x):  
 self.data = x \*\* 4  
 def \_\_str\_\_(self):  
 return str(self.data)  
   
print(‘Registry:’)  
for name in registry:  
 print(name, ‘=>’, registry[name], type(registry[name]))  
   
print(‘\nManual calls:’)  
print(spam(2)) # Вызов объекта вручную  
print(ham(2)) # Вызовы не перехватываются декоратором  
X = Eggs(2)  
print(X)  
   
print(‘\nRegistry calls:’)  
for name in registry:  
 print(name, ‘=>’, registry[name](3)) # Вызов из реестра  
Если запустить этот пример, он добавит декорированные объекты в реестр по   
их именам, однако они по-прежнему будут действовать, как предусмотрено ре-  
ализацией, при последующих вызовах, а вызовы не будут переадресовываться   
обертывающей логике. Фактически эти объекты могут вызываться непосред-  
ственно и с помощью реестра:  
Registry:  
Eggs => <class ‘\_\_main\_\_.Eggs’> <class ‘type’>  
ham => <function ham at 0x02CFB738> <class ‘function’>  
spam => <function spam at 0x02CFB6F0> <class ‘function’>  
   
Manual calls:  
4

Непосредственное управление функциями и классами   
1129  
8  
16  
   
Registry calls:  
Eggs => 81  
ham => 27  
spam => 9  
Этот прием можно использовать при создании пользовательского интерфейса,   
например чтобы зарегистрировать обработчики действий пользователя. Обра-  
ботчики могут регистрироваться по имени функции или класса, как сделано   
здесь, или можно было бы определять обрабатываемые события с помощью   
аргументов декоратора – для сохранения этих аргументов можно было бы ис-  
пользовать дополнительную инструкцию def, обертывающую декоратор.  
Этот пример достаточно искусственный, но сам прием используется очень   
широко. Например, декораторы функций могут также использоваться для   
обработки атрибутов функций, а декораторы классов могут динамически до-  
бавлять новые атрибуты классов и даже методы. Взгляните на следующие де-  
кораторы функций, – они присваивают функциям новые атрибуты, в которых   
сохраняется информация для последующего использования приложением, но   
не добавляют новый слой обертывающей логики, перехватывающей вызовы   
этих функций:  
# Непосредственное расширение декорируемых объектов  
   
>>> def decorate(func):  
... func.marked = True # Присваивает функции атрибут   
... return func # для последующего использования  
...  
>>> @decorate  
... def spam(a, b):  
... return a + b  
...  
>>> spam.marked  
True  
   
>>> def annotate(text): # То же самое, но значение атрибута   
... def decorate(func): # передается в аргументе декоратора  
... func.label = text  
... return func  
... return decorate  
...  
>>> @annotate(‘spam data’)  
... def spam(a, b): # spam = annotate(...)(spam)  
... return a + b  
...  
>>> spam(1, 2), spam.label  
(3, ‘spam data’)  
Такие декораторы расширяют сами функции и классы и не перехватывают по-  
следующие попытки их вызова. В следующей главе мы увидим дополнитель-  
ные примеры декораторов классов, которые непосредственно управляют клас-  
сами, потому что это, как оказывается, пересекается с областью применения   
метаклассов, а в оставшейся части главы давайте обратимся к двум объемным   
примерам декораторов.

1130   
Глава 38. Декораторы   
Пример: «частные» и «общедоступные»   
атрибуты  
В последних двух разделах главы будут представлены крупные примеры ис-  
пользования декораторов. Оба примера сопровождаются минимумом поясне-  
ний, отчасти потому, что эта глава и так уже превысила отведенный ей раз-  
мер, но в основном потому, что вы должны уже достаточно хорошо понимать   
основы декораторов, чтобы разобраться в примерах самостоятельно. Будучи   
по сути универсальными инструментами, эти примеры дают нам возможность   
увидеть, как концепции декораторов могут объединяться в программном коде,   
имеющем практическую ценность.  
Реализация частных атрибутов  
Следующий декоратор  классов реализует объявление Private для атрибутов   
экземпляров классов – то есть атрибутов, которые хранятся в экземпляре или   
наследуются от одного из его классов. Этот декоратор препятствует возмож-  
ности получения или изменения значений таких атрибутов из-за пределов де-  
корированного класса, но позволяет свободно обращаться к этим атрибутам   
внутри методов класса. Это объявление действует не совсем так, как в языках   
C++ или Java, но обеспечивает похожий способ управления доступом в языке   
Python.  
Мы уже видели первую неполную реализацию частных атрибутов экземпляра,   
доступных за пределами класса только для чтения, в главе 29. Версия, пред-  
ставленная здесь, распространяет эту концепцию также на операцию чтения,   
и для реализации этой модели использует прием делегирования вместо насле-  
дования. В определенном смысле это всего лишь расширенная версия декора-  
тора классов, реализующего трассировку обращений к атрибутам, с которым   
мы встречались выше.  
Несмотря на то, что для реализации частных атрибутов в этом примере исполь-  
зуется синтаксический подсластитель в виде декоратора классов, тем не менее,   
обработка обращений к атрибутам в нем полностью основана на использова-  
нии методов \_\_getattr\_\_ и \_\_setattr\_\_ перегрузки операторов, которые мы рас-  
сматривали в предыдущих главах. Когда обнаруживается попытка обращения   
к частному атрибуту, эта версия возбуждает исключение с помощью инструк-  
ции raise, которому передается текст сообщения об ошибке. Можно обработать   
исключение с помощью инструкции try или позволить ему завершить работу   
сценария.  
Ниже приводятся реализация декоратора и программный код самотестирова-  
ния в конце файла. Этот декоратор будет работать под управлением обеих вер-  
сий Python, 2.6 и 3.0, потому что в нем используется синтаксис вызова функ-  
Python, 2.6 и 3.0, потому что в нем используется синтаксис вызова функ-  
, 2.6 и 3.0, потому что в нем используется синтаксис вызова функ-  
ции print и инструкции raise, допустимый в версии 3.0, однако обращения   
к методам перегрузки операторов будут перехватываться только в версии 2.6   
(подробнее об этом чуть ниже):  
“””  
Ограничение на чтение значений частных атрибутов экземпляров классов.  
Примеры использования приводятся в программном коде самопроверки, в конце.  
Декоратор действует как: Doubler = Private(‘data’, ‘size’)(Doubler).  
Функция Private возвращает onDecorator, onDecorator возвращает onInstance,  
а в каждый экземпляр onInstance встраивается экземпляр Doubler.

Пример: «частные» и «общедоступные» атрибуты   
1131  
“””  
   
traceMe = False  
def trace(\*args):  
 if traceMe: print(‘[‘ + ‘ ‘.join(map(str, args)) + ‘]’)  
   
def Private(\*privates): # privates – в объемлющей области видимости  
 def onDecorator(aClass): # aClass – в объемлющей области видимости  
 class onInstance: # обертывает экземпляр атрибута  
 def \_\_init\_\_(self, \*args, \*\*kargs):  
 self.wrapped = aClass(\*args, \*\*kargs)  
 def \_\_getattr\_\_(self, attr): # Для собственных атрибутов getattr  
 # не вызывается  
 trace(‘get:’, attr) # Другие, как предполагается,   
 if attr in privates: # принадлежат обернутому объекту  
 raise TypeError(‘private attribute fetch: ‘ + attr)  
 else:  
 return getattr(self.wrapped, attr)  
 def \_\_setattr\_\_(self, attr, value): # Доступ извне  
 trace(‘set:’, attr, value) # Другие обрабатываются нормально  
 if attr == ‘wrapped’: # Разрешить доступ к своим атр.  
 self.\_\_dict\_\_[attr] = value # Избежать зацикливания  
 elif attr in privates:  
 raise TypeError(‘private attribute change: ‘ + attr)  
 else:  
 setattr(self.wrapped, attr, value) # Атрибуты обернутого   
 # объекта  
 return onInstance # Или использовать \_\_dict\_\_  
 return onDecorator  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 traceMe = True  
   
 @Private(‘data’, ‘size’) # Doubler = Private(...)(Doubler)  
 class Doubler:  
 def \_\_init\_\_(self, label, start):  
 self.label = label # Доступ изнутри класса  
 self.data = start # Не перехватывается: обрабатывается как обычно  
 def size(self):  
 return len(self.data) # Методы выполняются без проверки, потому  
 def double(self): # что ограничение доступа не наследуется  
 for i in range(self.size()):  
 self.data[i] = self.data[i] \* 2  
 def display(self):  
 print(‘%s => %s’ % (self.label, self.data))  
   
 X = Doubler(‘X is’, [1, 2, 3])  
 Y = Doubler(‘Y is’, [-10, -20, -30])  
   
 # Все следующие попытки оканчиваются успехом  
   
 print(X.label) # Доступ извне класса  
 X.display(); X.double(); X.display() # Перехватывается: проверяется,   
 # делегируется  
 print(Y.label)  
 Y.display(); Y.double()  
 Y.label = ‘Spam’  
 Y.display()

1132   
Глава 38. Декораторы   
 # Все следующие попытки терпят неудачу  
 “””  
 print(X.size()) # Выведет “TypeError: private attribute fetch: size”  
 print(X.data)  
 X.data = [1, 1, 1]  
 X.size = lambda S: 0  
 print(Y.data)  
 print(Y.size())  
 “””  
Когда переменная traceMe получает значение True, программный код самопро-  
верки модуля выводит следующие результаты. Обратите внимание, что декора-  
тор перехватывает и проверяет допустимость обеих операций, чтения и запи-  
си, над атрибутами, когда они выполняются за пределами обернутого класса,   
но не перехватывает попытки доступа изнутри самого класса:  
[set: wrapped <\_\_main\_\_.Doubler object at 0x02B2AAF0>]  
[set: wrapped <\_\_main\_\_.Doubler object at 0x02B2AE70>]  
[get: label]  
X is  
[get: display]  
X is => [1, 2, 3]  
[get: double]  
[get: display]  
X is => [2, 4, 6]  
[get: label]  
Y is  
[get: display]  
Y is => [-10, -20, -30]  
[get: double]  
[set: label Spam]  
[get: display]  
Spam => [-20, -40, -60]  
Подробности реализации I  
Реализация получилась немного сложной и вам, вероятно, лучше всего самим   
поэкспериментировать с ней, чтобы разобраться с тем, как она действует. Од-  
нако, чтобы помочь вам в изучении примера, я упомяну несколько важных мо-  
ментов.  
Наследование и делегирование  
В первой прикидочной реализации частных атрибутов, которая приводилась   
в главе 29, для перехвата попыток изменить значение атрибута использовал-  
ся механизм наследования и метод \_\_setattr\_\_. Однако наследование только   
осложняет ситуацию, потому что отличить, откуда была произведена попыт-  
ка доступа, изнутри или снаружи, не так-то просто (попытки доступа изнутри   
должны выполняться как обычно, а попытки доступа снаружи должны огра-  
ничиваться). Чтобы обойти эту проблему, пример в главе 29 требует, чтобы для   
изменения значений атрибутов наследующие их классы использовали словарь   
\_\_dict\_\_, что в лучшем случае можно признать неполным решением.  
Данная версия вместо наследования использует прием делегирования (прием   
встраивания одного объекта в другой) – он лучше подходит для решения на-  
шей задачи, так как позволяет легко определить, откуда была выполнена по-

Пример: «частные» и «общедоступные» атрибуты   
1133  
пытка доступа – изнутри класса или снаружи. Попытки доступа к атрибутам   
снаружи перехватываются методами перегрузки операторов обертки и делеги-  
руются классу, если они допустимы. Попытки доступа изнутри самого класса   
(то есть из методов класса через аргумент self) не перехватываются и выполня-  
ются как обычно, без всяких проверок, потому что ограничение доступа в дан-  
ной реализации не наследуется.  
Аргументы декоратора  
Декоратор класса, используемый здесь, принимает произвольное число аргу-  
ментов – имен частных атрибутов. Однако, обратите внимание, что аргументы   
передаются функции Private, а функция Private возвращает функцию декора-  
тора, которая применяется к классу. То есть аргументы используются еще до   
того, как будет выполнена операция декорирования – функция Private возвра-  
щает декоратор, который в свою очередь «запоминает» список имен частных   
атрибутов в виде ссылки в область видимости объемлющей функции.  
Сохранение информации   
и объемлющие области видимости  
Говоря об объемлющих областях видимости, следует отметить, что в действи-  
тельности в этом примере существует три уровня, где сохраняется информация   
о состоянии:  
 •  
Аргументы функции Private используются еще до того, как будет выполне-  
на операция декорирования, и сохраняются в области видимости объемлю-  
щей функции для последующего использования в функции onDecorator и в   
классе onInstance.  
 •  
Аргумент aClass функции onDecorator используется на этапе декорирования   
и сохраняется в объемлющей области видимости для последующего ис-  
пользования на этапе создания экземпляра.  
 •  
Обернутый объект экземпляра сохраняется в атрибуте экземпляра класса   
onInstance для последующего использования при обработке попыток досту-  
па к атрибутам из-за пределов класса.  
Все эти механизмы действуют вполне естественно, в полном соответствии   
с правилами областей видимости в языке Python.  
Использование \_\_dict\_\_ и \_\_slots\_\_  
Метод \_\_setattr\_\_ в этом примере опирается на использование словаря \_\_dict\_\_   
пространства имен объекта экземпляра, с помощью которого он изменяет зна-  
чение собственного атрибута wrapped экземпляра класса onInstance. Как мы   
узнали в предыдущей главе, в этом методе нельзя выполнять прямое присваи-  
вание значения атрибуту из-за опасности зацикливания. Однако для присваи-  
вания значений атрибутам обернутого объекта вместо словаря \_\_dict\_\_ исполь-  
зуется встроенная функция setattr. Кроме того, для получения значений атри-  
бутов обернутого объекта используется функция getattr, потому что атрибуты   
не только могут храниться в самом объекте, но и наследоваться от класса.  
Благодаря этому данная реализация декоратора может применяться практи-  
чески к любым классам. Вы можете вспомнить, как в главе 31 говорилось, что   
классы нового стиля с атрибутом \_\_slots\_\_ могут вообще не хранить атрибуты   
в словаре \_\_dict\_\_. Однако так как мы используем словарь \_\_dict\_\_ только на

1134   
Глава 38. Декораторы   
уровне экземпляра класса onInstance и не используем его на уровне обернутого   
экземпляра, а также потому, что функции setattr и getattr могут применяться   
к атрибутам, хранящимся в \_\_dict\_\_ и \_\_slots\_\_, наш декоратор может приме-  
няться к классам, использующим любой механизм хранения атрибутов.  
Обобщение на случай объявления   
общедоступных атрибутов  
Теперь, когда у нас имеется реализация декоратора Private частных атрибутов,   
будет совсем несложно обобщить ее и реализовать декоратор Public для объяв-  
ления общедоступных атрибутов – атрибуты этого типа по сути являются про-  
тивоположностью частным атрибутам, поэтому в их реализации нам достаточ-  
но будет просто инвертировать условие во внутренней проверке. Пример, кото-  
рый приводится в этом разделе, реализует декораторы классов Private и Public,   
которые дают возможность определять либо множество общедоступных, либо   
множество частных атрибутов (атрибуты, хранящиеся в экземпляре или на-  
следуемые ими от своих классов) со следующей семантикой:  
 •  
Декоратор Private объявляет атрибуты экземпляров класса, которые недо-  
ступны, кроме как изнутри методов класса. То есть ни один атрибут, имя   
которого было объявлено с помощью декоратора Private, не будет доступен   
за пределами класса, при этом все остальные атрибуты будут доступны   
внешнему программному коду как для чтения, так и для записи.  
 •  
Декоратор Public объявляет атрибуты экземпляров класса, которые будут   
доступны не только внутри методов класса, но и снаружи. То есть все атри-  
буты, имена которых были объявлены с помощью декоратора Public, будут   
доступны из любой точки программы, однако атрибуты, которые не были   
объявлены общедоступными, будут доступны только внутри класса.  
Объявления Private и Public являются взаимоисключающими: при использо-  
вании декоратора Private все необъявленные атрибуты будут считаться обще-  
доступными и наоборот, при использовании декоратора Public все необъявлен-  
ные атрибуты будут считаться частными. По сути эти объявления являются   
полными противоположностями друг другу. Однако в случае необъявленных   
имен атрибутов, которые не создаются внутри методов класса, они ведут себя   
немного по-разному – сохраняется возможность создавать присваиванием   
атрибуты за пределами класса, когда к классу применяется декоратор Private,   
(все необъявленные имена доступны), но не тогда, когда к классу применяется   
декоратор Public (все необъявленные имена недоступны).  
И снова вам следует изучить этот пример самостоятельно, чтобы получить   
представление о том, как он действует. Обратите внимание, что на этот раз до-  
бавился еще один, четвертый уровень, где сохраняется информация, вдоба-  
вок к описанным в предыдущем разделе: функции проверки, реализованные   
в виде lambda-функций, сохраняются в дополнительной области видимости   
объемлющей функции. Этот пример может выполняться под управлением лю-  
бой версии Python, 2.6 или 3.0, но с некоторыми оговорками, которые отно-  
Python, 2.6 или 3.0, но с некоторыми оговорками, которые отно-  
, 2.6 или 3.0, но с некоторыми оговорками, которые отно-  
сятся к версии 3.0 (краткие пояснения приводятся в строке документирования   
модуля, а более подробные пояснения приводятся после листинга примера):  
“””  
Декораторы Private и Public для объявления частных и общедоступных атрибутов.  
Управляют доступом к атрибутам, хранящимся в экземпляре или наследуемым  
от классов. Декоратор Private объявляет атрибуты, которые недоступны за

Пример: «частные» и «общедоступные» атрибуты   
1135  
пределами декорируемого класса, а декоратор Public объявляет все атрибуты,   
которые, наоборот, будут доступны. Внимание: в Python 3.0 эти декораторы   
оказывают воздействие только на атрибуты с обычными именами – вызовы методов  
перегрузки операторов с именами вида \_\_X\_\_, которые неявно производятся   
встроенными операциями, не перехватываются методами \_\_getattr\_\_ и \_\_getattribute\_\_   
в классах нового стиля.  
Добавьте здесь реализации методов вида \_\_X\_\_ и с их помощью делегируйте выполнение   
операций встроенным объектам.  
“””  
   
traceMe = False  
def trace(\*args):  
 if traceMe: print(‘[‘ + ‘ ‘.join(map(str, args)) + ‘]’)  
   
def accessControl(failIf):  
 def onDecorator(aClass):  
 class onInstance:  
 def \_\_init\_\_(self, \*args, \*\*kargs):  
 self.\_\_wrapped = aClass(\*args, \*\*kargs)  
 def \_\_getattr\_\_(self, attr):  
 trace(‘get:’, attr)  
 if failIf(attr):  
 raise TypeError(‘private attribute fetch: ‘ + attr)  
 else:  
 return getattr(self.\_\_wrapped, attr)  
 def \_\_setattr\_\_(self, attr, value):  
 trace(‘set:’, attr, value)  
 if attr == ‘\_onInstance\_\_wrapped’:  
 self.\_\_dict\_\_[attr] = value  
 elif failIf(attr):  
 raise TypeError(‘private attribute change: ‘ + attr)  
 else:  
 setattr(self.\_\_wrapped, attr, value)  
 return onInstance  
 return onDecorator  
   
def Private(\*attributes):  
 return accessControl(failIf=(lambda attr: attr in attributes))  
   
def Public(\*attributes):  
 return accessControl(failIf=(lambda attr: attr not in attributes))  
Чтобы выяснить, как использовать эти декораторы, загляните в программный   
код самопроверки в предыдущем примере. Ниже приводится короткий пример   
использования этих декораторов классов в интерактивной оболочке (он одина-  
ково работает под управлением Python 2.6 и 3.0). Как и следовало ожидать,   
нечастные, то есть общедоступные, атрибуты доступны за пределами декори-  
руемого класса, а частные, то есть не общедоступные, атрибуты, – нет:  
>>> from access import Private, Public  
   
>>> @Private(‘age’) # Person = Private(‘age’)(Person)  
... class Person: # Person = onInstance с информацией о состоянии  
... def \_\_init\_\_(self, name, age):  
... self.name = name  
... self.age = age # Внутри доступ к атрибутам не ограничивается  
...  
>>> X = Person(‘Bob’, 40)

1136   
Глава 38. Декораторы   
>>> X.name # Попытки доступа снаружи проверяются  
‘Bob’  
>>> X.name = ‘Sue’  
>>> X.name  
‘Sue’  
>>> X.age  
TypeError: private attribute fetch: age  
>>> X.age = ‘Tom’  
TypeError: private attribute change: age  
   
>>> @Public(‘name’)  
... class Person:  
... def \_\_init\_\_(self, name, age):  
... self.name = name  
... self.age = age  
...  
>>> X = Person(‘bob’, 40) # X – экземпляр onInstance  
>>> X.name # экземп. Person встраивается в экземп. onInstance  
‘bob’  
>>> X.name = ‘Sue’  
>>> X.name  
‘Sue’  
>>> X.age  
TypeError: private attribute fetch: age  
>>> X.age = ‘Tom’  
TypeError: private attribute change: age  
Подробности реализации II  
Чтобы помочь вам в изучении примера, я сделаю несколько заключительных   
замечаний, касающихся этой версии. Поскольку этот пример является всего   
лишь обобщением примера из предыдущего раздела, большинство замечаний,   
которые были сделаны там, в равной степени справедливы и здесь.  
Использование псевдочастных имен вида \_\_X  
Помимо обобщения в этой версии также используется механизм искажения   
псевдочастных имен вида \_\_X (который мы обсуждали в главе 30), чтобы обе-  
спечить локальность атрибута \_\_wrapped в управляющем классе за счет добав-  
ления имени класса в начало имени атрибута. Это позволяет избежать кон-  
фликта с возможным именем wrapped в обернутом классе, и вообще этот прием   
полезно использовать в подобных универсальных инструментах. Тем не менее   
псевдочастные атрибуты не являются по-настоящему «частными», потому что   
искаженное имя доступно за пределами класса. Обратите также внимание,   
что в методе \_\_setattr\_\_ используется строка с полностью развернутым име-  
нем (‘\_onInstance\_\_wrapped’), потому что именно такое имя создается интерпре-  
татором.  
Нарушение ограничений  
Несмотря на то, что этот пример действительно ограничивает доступ к атрибу-  
там экземпляров и их классов, тем не менее, существует возможность обойти   
эти ограничения несколькими способами. Например, с использованием ис-  
каженной версии имени атрибута \_\_wrapped (попытка обратиться к атрибуту

Пример: «частные» и «общедоступные» атрибуты   
1137  
bob.pay может оказаться неудачной, но обращение к атрибуту bob.\_onInstance\_\_  
wrapped.pay увенчается успехом!). Однако если для обхода ограничений требу-  
ется явно использовать подобные имена, вероятно, этих ограничений вполне   
достаточно для большинства применений. Конечно, ограничения на доступ   
к частным атрибутам можно преодолеть в любом языке, если действовать до-  
статочно грубо (в некоторых реализациях языка C++ может помочь объявле-  
C++ может помочь объявле-  
++ может помочь объявле-  
ние #define private public). Механизмы управления доступом могут помочь сни-  
зить вероятность случайных ошибок, которые совершаются программистами   
на любом языке, но всегда, когда имеется возможность изменить исходные тек-  
сты, управление доступом оказывается недостижимой мечтой.  
Преимущества декораторов  
Те же результаты мы могли бы получить без применения декораторов, ис-  
пользуя управляющие функции или реализуя повторное присваивание име-  
нам вручную. Однако синтаксис декораторов делает программный код более   
последовательным и немного более очевидным. Главный недостаток декора-  
торов, как и любого другого подхода, основанного на обертывании исходных   
объектов, заключается в дополнительных вызовах, которые производятся   
при обращении к атрибутам, а также в том, что экземпляры декорированных   
классов в действительности не являются экземплярами оригинальных, де-  
корируемых классов. Если, к примеру, проверить тип с помощью X.\_\_class\_\_   
или isinstance(X, C), можно убедиться, что они являются экземплярами обе-  
ртывающего класса. Однако если вы не планируете выполнять проверку ти-  
пов объектов, то проблема изменения типа, вероятно, не будет иметь никакого   
значения.  
Нерешенные проблемы  
В текущем своем виде этот пример действует именно так, как и предусматри-  
валось, в обеих версиях Python, 2.6 и 3.0 (если методы перегрузки операторов   
будут переопределены в классе-обертке, чтобы делегировать выполнение опе-  
раций обернутому классу). Тем не менее в любом программном обеспечении   
всегда есть что улучшить.  
Внимание: вызовы методов перегрузки операторов в 3.0   
выполняются иначе  
Как и все другие классы, использующие прием делегирования на основе ме-  
тода \_\_getattr\_\_, этот декоратор будет работать в любой версии Python только   
в случае применения к атрибутам с обычными именами – обработка методов   
перегрузки операторов, таких как \_\_str\_\_ и \_\_add\_\_, в классах нового стиля вы-  
полняется иначе. Поэтому в версии 3.0 попытки перехватить таким способом   
обращения к этим методам во встроенном объекте будут терпеть неудачу.  
Как мы узнали в предыдущей главе, поиск методов перегрузки операторов   
в классических классах начинается с экземпляра, но в классах нового сти-  
ля это не так – интерпретатор вообще пропускает этап поиска в экземплярах   
и производит поиск таких методов, начиная с классов. Поэтому при неявном   
обращении к методам перегрузки операторов с именами вида \_\_X\_\_, в процессе   
выполнения встроенных операций интерпретатор не вызывает ни метод \_\_ge-  
ge-  
tattr\_\_, ни метод \_\_getattribute\_\_ при работе с классами нового стиля в вер-

1138   
Глава 38. Декораторы   
сии 2.6 и со всеми классами в версии 3.0. Обращение к таким атрибутам про-  
исходит без вызова метода onInstance.\_\_getattr\_\_, вследствие чего декоратор не   
имеет возможности ни управлять доступом к ним, ни делегировать их вызовы.  
Класс нашего декоратора определен не как класс нового стиля (он не наследует   
явно суперкласс object), поэтому в версии 2.6 он будет перехватывать обраще-  
ния к методам перегрузки операторов. Но, поскольку в версии 3.0 все классы   
автоматически считаются классами нового стиля, обращения к таким методам   
встроенного объекта перехватываться не будут. Чтобы обойти эту проблему   
в версии 3.0, проще всего будет переопределить в классе onInstance все методы   
перегрузки операторов, которые могут иметься в обертываемых объектах. Та-  
кие дополнительные методы можно добавить вручную, с помощью инструмен-  
тов, частично автоматизирующих эту задачу (например, с помощью декорато-  
ров классов или метаклассов, которые обсуждаются в следующей главе), или   
за счет определения этих методов в суперклассах.  
Чтобы понять разницу, попробуем применить декоратор к классу, который реа-  
лизует метод перегрузки оператора, в Python 2.6. Управление доступом будет   
работать, как и прежде, – обращения к обоим методам, \_\_str\_\_ – при выводе   
и \_\_add\_\_ – при выполнении операции +, будут приводить к вызову метода \_\_ge-  
ge-  
tattr\_\_ декоратора, благодаря чему он сможет проверить и делегировать выпол-  
нение операции экземпляру класса Person:  
C:\misc> c:\python26\python  
>>> from access import Private  
>>> @Private(‘age’)  
... class Person:  
... def \_\_init\_\_(self):  
... self.age = 42  
... def \_\_str\_\_(self):  
... return ‘Person: ‘ + str(self.age)  
... def \_\_add\_\_(self, yrs):  
... self.age += yrs  
...  
>>> X = Person()  
>>> X.age # Проверка обычного имени выполняется корректно  
TypeError: private attribute fetch: age  
>>> print(X) # \_\_getattr\_\_ => вызовет Person.\_\_str\_\_  
Person: 42  
>>> X + 10 # \_\_getattr\_\_ => вызовет Person.\_\_add\_\_  
>>> print(X) # \_\_getattr\_\_ => вызовет Person.\_\_str\_\_  
Person: 52  
Однако если то же самое выполнить под управлением Python 3.0, неявные вы-  
Python 3.0, неявные вы-  
 3.0, неявные вы-  
зовы методов \_\_str\_\_ и \_\_add\_\_ будут выполняться, минуя метод \_\_getattr\_\_   
декоратора, при этом поиск определений методов будет производиться в клас-  
се самого декоратора и выше в дереве наследования. Функция print выведет   
информацию по умолчанию, вызвав метод отображения, унаследованный от   
типа класса (технически, от подразумеваемого в версии 3.0 суперкласса object),   
а операция + сгенерирует ошибку, потому что отсутствует наследуемый метод   
\_\_add\_\_ по умолчанию:  
C:\misc> c:\python30\python  
>>> from access import Private  
>>> @Private(‘age’)

Пример: «частные» и «общедоступные» атрибуты   
1139  
... class Person:  
... def \_\_init\_\_(self):  
... self.age = 42  
... def \_\_str\_\_(self):  
... return ‘Person: ‘ + str(self.age)  
... def \_\_add\_\_(self, yrs):  
... self.age += yrs  
...  
>>> X = Person()   
>>> X.age # Проверка обычного имени по-прежнему действует  
TypeError: private attribute fetch: age  
>>> print(X) # Но в 3.0 делегирование встроенных операций   
 # не выполняется!  
<access.onInstance object at 0x025E0790>  
>>> X + 10  
TypeError: unsupported operand type(s) for +: ‘onInstance’ and ‘int’  
>>> print(X)  
<access.onInstance object at 0x025E0790>  
Использование альтернативного метода \_\_getattribute\_\_ не дает положитель-  
ных результатов – несмотря на то, что он должен перехватывать обращения   
к любым атрибутам (не только к именам, которые не определены), он также   
не вызывается встроенными операциями. Здесь не поможет и использование   
свойств, с которыми мы встречались в главе 37, – напомню, что свойства авто-  
матически выполняют программный код, ассоциированный с определенными   
атрибутами, объявленными в классе, и не предназначены для обработки про-  
извольных атрибутов в обернутых объектах.  
Как уже упоминалось ранее, самое простое решение этой проблемы в Py-  
Py-  
thon 3.0 заключается в том, чтобы в классах, реализующих шаблон делеги-  
 3.0 заключается в том, чтобы в классах, реализующих шаблон делеги-  
рования, как наш декоратор, переопределить методы перегрузки операторов,   
которые могут присутствовать во встраиваемых объектах. Это далеко не иде-  
альное решение, так как при этом создается избыточный программный код,   
особенно если сравнить с решением в версии 2.6. Однако реализация такого   
решения не требует большого количества усилий и до некоторой степени мо-  
жет быть автоматизирована с помощью дополнительных инструментов или су-  
перклассов. Такого решения вполне достаточно, чтобы обеспечить корректную   
работу нашего декоратора под управлением Python 3.0 и возможность объявле-  
Python 3.0 и возможность объявле-  
 3.0 и возможность объявле-  
ния методов перегрузки операторов частными или общедоступными (предпо-  
лагается, что каждый метод перегрузки операторов будет выполнять проверку   
failIf):  
def accessControl(failIf):  
 def onDecorator(aClass):  
 class onInstance:  
 def \_\_init\_\_(self, \*args, \*\*kargs):  
 self.\_\_wrapped = aClass(\*args, \*\*kargs)  
   
 # Перехват и делегирование методов перегрузки операторов  
 def \_\_str\_\_(self):  
 return str(self.\_\_wrapped)  
 def \_\_add\_\_(self, other):  
 return self.\_\_wrapped + other  
 def \_\_getitem\_\_(self, index):  
 return self.\_\_wrapped[index] # Если необходимо

1140   
Глава 38. Декораторы   
 def \_\_call\_\_(self, \*args, \*\*kargs):  
 return self.\_\_wrapped(\*arg, \*kargs) # Если необходимо  
 ...плюс любые другие необходимые методы...  
   
 # Перехват и делегирование обращений к обычным атрибутам  
 def \_\_getattr\_\_(self, attr):  
 ...  
 def \_\_setattr\_\_(self, attr, value):  
 ...  
 return onInstance  
 return onDecorator  
После добавления методов перегрузки операторов предыдущий пример с мето-  
дами \_\_str\_\_ и \_\_add\_\_ будет работать одинаково в обеих версиях Python, 2.6   
и 3.0. Однако чтобы полностью адаптировать декоратор для работы в вер-  
сии 3.0, может потребоваться добавить существенный объем избыточного про-  
граммного кода. В принципе, в таких универсальных инструментах, как этот   
декоратор, в версии 3.0 необходимо переопределить каждый метод перегрузки   
операторов, доступ к которому не может управляться автоматически (имен-  
но по этой причине данное расширение было опущено в нашем программном   
коде). Поскольку в Python 3.0 все классы автоматически считаются классами   
нового стиля, реализация шаблона делегирования в этой версии оказывается   
более сложной (хотя и не невозможной).  
С другой стороны, классы-обертки, использующие прием делегирования, мо-  
гут просто наследовать общий суперкласс, который переопределяет методы   
перегрузки операторов, реализующие стандартный способ делегирования.   
Кроме того, такие инструменты, как дополнительные декораторы классов   
или метаклассы, могут автоматизировать какую-то часть работы, связанной   
с добавлением таких методов (дополнительные подробности вы найдете в при-  
мерах расширения классов в главе 39). Такие приемы могут помочь сделать   
классы-обертки в Python 3.0 более обобщенными, хотя решение не будет таким   
же простым, как в Python 2.6.  
Альтернативные реализации: вставка метода   
\_\_getattribute\_\_, проверка стека вызовов   
Хотя определение избыточных методов перегрузки операторов в классах-  
обертках является наиболее простым решением проблемы в Python 3.0, обо-  
Python 3.0, обо-  
 3.0, обо-  
значенной в предыдущем разделе, тем не менее, оно не единственное. Мы не   
имеем возможности продолжить здесь детальное исследование этой пробле-  
мы, поэтому исследование других потенциальных решений предлагается вы-  
полнить самостоятельно, в качестве упражнения. Однако одна из тупиковых   
альтернатив настолько показательна для классов, что я считаю ее достойной   
краткого упоминания.  
Один из недостатков этого примера заключается в том, что объекты экземпля-  
ров в действительности не являются экземплярами оригинального класса –   
они являются экземплярами класса-обертки. В некоторых программах, кото-  
рые при принятии решений опираются на проверку типов, это обстоятельство   
может иметь большое значение. Чтобы обеспечить поддержку таких проверок,   
можно было бы попробовать добиться похожего эффекта, вставив метод \_\_ge-  
ge-  
tattribute\_\_ в оригинальный класс, чтобы перехватывать все попытки обраще-  
ния к атрибутам его экземпляров. Этот добавленный метод мог бы передавать   
запросы суперклассу, чтобы избежать зацикливания, используя приемы, ко-

Пример: «частные» и «общедоступные» атрибуты   
1141  
торые мы изучили в предыдущей главе. Ниже приводятся возможные измене-  
ния в классе нашего декоратора:  
# Поддерживает трассировку, как и прежде  
   
def accessControl(failIf):  
 def onDecorator(aClass):  
 def getattributes(self, attr):  
 trace(‘get:’, attr)  
 if failIf(attr):  
 raise TypeError(‘private attribute fetch: ‘ + attr)  
 else:  
 return object.\_\_getattribute\_\_(self, attr)  
 aClass.\_\_getattribute\_\_ = getattributes  
 return aClass  
 return onDecorator  
   
def Private(\*attributes):  
 return accessControl(failIf=(lambda attr: attr in attributes))  
   
def Public(\*attributes):  
 return accessControl(failIf=(lambda attr: attr not in attributes))  
Эта альтернативная реализация решает проблему проверки типа, но страдает   
от двух других проблем. Например, она обрабатывает только попытки получе-  
ния атрибутов – в таком своем виде эта версия позволяет выполнять присваи-  
вание значений частным атрибутам. Чтобы перехватить операции присваи-  
вания, по-прежнему необходимо использовать метод \_\_setattr\_\_ либо добавив   
объект-обертку, либо вставив метод в оригинальный класс. Добавление оберт-  
ки, перехватывающей операции присваивания, снова приведет к изменению   
типа, а добавляя метод в оригинальный класс, можно затереть уже существую-  
щий в нем метод \_\_setattr\_\_ (или \_\_getattribute\_\_ в данном случае!). Добавляе-  
мый метод \_\_setattr\_\_ должен также учитывать возможное наличие атрибута   
\_\_slots\_\_ в клиентском классе.  
Кроме того, эта реализация не решает проблему встроенных операций, описан-  
ную в предыдущем разделе, потому что в таких случаях метод \_\_getattribute\_\_   
не вызывается. Если бы наш класс Person имел собственный метод \_\_str\_\_, опе-  
рация вывода вызывала бы его, но только потому, что он присутствует в дан-  
ном конкретном классе. Однако, как и прежде, попытка обращения к атрибуту   
\_\_str\_\_ не привела бы к вызову вставленного метода \_\_getattribute\_\_ – опера-  
ция вывода обойдет этот метод и напрямую вызовет метод \_\_str\_\_ класса.  
Хотя такое решение, возможно, лучше, чем полный отказ от поддержки ме-  
тодов перегрузки операторов в обернутом объекте (исключая переопределение,   
по крайней мере), тем не менее, этот прием по-прежнему не обеспечивает воз-  
можность перехватывать и проверять допустимость вызовов методов \_\_X\_\_, де-  
лая невозможным ограничить доступ к ним. Большинство методов перегрузки   
операторов, как предполагается, должны быть общедоступными, но для неко-  
торых бывает желательно сделать исключение.  
Хуже того, так как этот прием, не использующий обертку, добавляет метод   
\_\_getattribute\_\_ в декорируемый класс, этот метод будет перехватывать попыт-  
ки обращения к атрибутам, выполняемые самим классом, и проверять их так,   
как если бы они выполнялись из-за пределов класса, – это означает, что методы   
класса тоже лишатся возможности использовать частные атрибуты!

1142   
Глава 38. Декораторы   
Фактически вставка методов таким способом функционально эквивалентна их   
наследованию, что предполагает наличие ограничений, свойственных ориги-  
нальной версии реализации частных атрибутов в главе 29. Чтобы узнать, отку-  
да была выполнена попытка доступа к атрибуту, изнутри класса или снаружи,   
нашему методу могло бы потребоваться проверить объекты кадров стека вызо-  
вов. В конечном счете, это помогло бы решить проблему (например, можно было   
бы заменить частные атрибуты свойствами или дескрипторами, проверяющи-  
ми стек вызовов), но это существенно замедлило бы операции доступа, а кроме   
того, для нас реализация такого решения слишком отдает шаманством, чтобы   
рассматривать ее здесь.  
Несмотря на оригинальность и пригодность для использования в некоторых   
случаях, этот прием вставки метода не достигает поставленных нами целей.   
У нас нет возможности продолжить исследование этого приема, потому что   
приемы расширения классов мы будем рассматривать в следующей главе,   
в соединении с метаклассами. Как мы увидим там, метаклассы не являются   
единственной возможностью изменения классов таким способом, потому что   
нередко эту же роль могут играть декораторы классов.  
Для языка Python не характерно управлять доступом  
Теперь, когда я прошел такой длинный путь, объясняя, как реализовать объ-  
явления Private и Public для атрибутов, я должен снова напомнить, что это не   
совсем в стиле программирования на Python – ограничивать доступ к классам.   
Фактически большинство программистов на языке Python сочтут эти примеры   
нужными разве что для демонстрации декораторов в действии. Даже очень боль-  
шие программы на языке Python благополучно обходятся вообще без каких-  
либо средств управления доступом. Если вам действительно необходимо регу-  
лировать доступ к атрибутам, чтобы уменьшить число ошибок при программи-  
ровании, или так случилось, что вы – бывший-программист-на-языке-C++-или-  
Java, то и тогда большую часть того, что вам действительно необходимо, можно   
реализовать с помощью инструментов интроспекции и перегрузки операторов.  
Пример: проверка аргументов функций  
В качестве заключительного примера, демонстрирующего удобство декора-  
торов, в этом разделе мы создадим декоратор, который автоматически прове-  
ряет числовые аргументы, передаваемые функции или методу, на вхождение   
в определенный диапазон. Он предназначен для использования на стадии раз-  
работки или эксплуатации и может рассматриваться как шаблон для решения   
похожих задач (например, для проверки типов аргументов, если это потребу-  
ется). Эта глава по своему объему превысила все мыслимые размеры, поэтому   
программный код примера сопровождается весьма кратким описанием и вам   
придется изучать его практически самостоятельно – как обычно, подробности   
ищите в программном коде.  
Цель  
В пособии по объектно-ориентированному программированию в главе 27 мы   
написали класс, метод giveRaise которого используется для увеличения на   
указанный процент зарплаты сотрудникам, которых представляют объекты   
класса:

Пример: проверка аргументов функций   
1143  
class Person:  
 ...  
 def giveRaise(self, percent):  
 self.pay = int(self.pay \* (1 + percent))  
Тогда мы отметили, что для повышения надежности программного кода было   
бы неплохо проверить аргумент percent, чтобы убедиться, что он имеет не   
слишком большое и не слишком маленькое значение. Мы могли бы реализо-  
вать такую проверку с помощью инструкции if или assert внутри самого мето-  
да, используя встроенную проверку:  
class Person:  
 def giveRaise(self, percent): # Проверка с помощью встроенного прогр. кода  
 if percent < 0.0 or percent > 1.0:  
 raise TypeError, ‘percent invalid’  
 self.pay = int(self.pay \* (1 + percent))  
   
class Person: # Проверка с помощью инструкции assert  
 def giveRaise(self, percent):  
 assert percent >= 0.0 and percent <= 1.0, ‘percent invalid’  
 self.pay = int(self.pay \* (1 + percent))  
Однако такой подход загромождает метод встроенными проверками, которые,   
скорее всего, будут иметь смысл только во время разработки. В более сложных   
случаях такой подход может стать утомительным (попробуйте представить,   
как мог бы выглядеть программный код встроенной проверки для управления   
доступом к атрибуту, которое было реализовано в предыдущем разделе в виде   
декоратора). Хуже того, если логику проверки придется когда-нибудь изменить,   
может потребоваться найти и изменить множество копий встроенной проверки.  
Гораздо удобнее и интереснее было бы создать альтернативу в виде универ-  
сального инструмента, который мог бы автоматически выполнять проверку на   
вхождение значения любого аргумента любой функции или метода в заданный   
диапазон. Наиболее очевидным и удобным представляется решение на основе   
декоратора:  
class Person:  
 @rangetest(percent=(0.0, 1.0)) # Проверка с помощью декоратора  
 def giveRaise(self, percent):  
 self.pay = int(self.pay \* (1 + percent))  
Реализация логики проверки в виде декоратора упрощает реализацию клиен-  
тов и сопровождение программы в будущем.  
Обратите внимание, что наша цель здесь отличается от проверки атрибутов,   
реализованной в последнем примере предыдущей главы. Здесь мы собираемся   
проверять значения аргументов в вызовах функций, а не значения атрибутов   
при выполнении операции присваивания. Декораторы и средства интроспек-  
ции в языке Python позволяют легко реализовать эту задачу.  
Простой декоратор проверки значений позиционных   
аргументов на вхождение в заданный диапазон  
Начнем с реализации простой проверки на вхождение в диапазон. Для просто-  
ты мы сначала создадим декоратор, который работает только с позиционны-  
ми аргументами и предполагает, что они всегда передаются в одной и той же   
позиции во всех вызовах. Они не могут передаваться в виде именованных ар-

1144   
Глава 38. Декораторы   
гументов, и нам не требуется обеспечивать поддержку передачи именованных   
аргументов \*\*args, потому что в этом случае порядок следования аргументов   
может не соответствовать позициям, объявленным в декораторе. Ниже приво-  
дится программный код, сохраненный в файле devtools.py:  
def rangetest(\*argchecks): # Проверяет позиционные аргументы на вхождение   
 def onDecorator(func): # в заданный диапазон  
 if not \_\_debug\_\_: # True – если “python -O main.py args...”  
 return func # Ничего не выполняет: просто возвращает   
 # оригинальную функцию  
 else: # Иначе, на этапе отладки, возвращает обертку  
 def onCall(\*args):  
 for (ix, low, high) in argchecks:  
 if args[ix] < low or args[ix] > high:  
 errmsg = ‘Argument %s not in %s..%s’ % (ix, low, high)  
 raise TypeError(errmsg)  
 return func(\*args)  
 return onCall  
 return onDecorator  
Как можно заметить, этот программный код в основном реализует шаблоны   
проектирования, исследованные нами ранее: здесь используются аргументы   
декоратора, вложенные области видимости для сохранения информации о со-  
стоянии и так далее.  
Кроме того, опираясь на полученные ранее знания, мы использовали вложен-  
ные инструкции def, чтобы гарантировать возможность применения декорато-  
ра и к простым функциям, и к методам. Когда декоратор применяется к методу   
класса, функция onCall будет принимать подразумеваемый экземпляр класса   
в первом элементе списка \*args и передавать его в виде аргумента self ориги-  
нальному методу – нумерация проверяемых аргументов в этом случае начина-  
ется с 1, а не с 0.   
Обратите также внимание, что здесь используется встроенная переменная \_\_  
debug\_\_ – интерпретатор присваивает ей значение True, если только он не был   
запущен с параметром –O командной строки, включающим режим оптимиза-  
ции (например, python –O main.py). Если переменная \_\_debug\_\_ имеет значение   
False, декоратор возвращает оригинальную функцию, чтобы избежать допол-  
нительных вызовов и связанной с ними потери производительности.  
Эта первая версия решения используется, как показано ниже:  
# Файл devtools\_test.py  
   
from devtools import rangetest  
print(\_\_debug\_\_) # False, если “python –O main.py”  
   
@rangetest((1, 0, 120)) # persinfo = rangetest(...)(persinfo)  
def persinfo(name, age): # Значение age должно быть в диапазоне 0..120  
 print(‘%s is %s years old’ % (name, age))  
   
@rangetest([0, 1, 12], [1, 1, 31], [2, 0, 2009])  
def birthday(M, D, Y):  
 print(‘birthday = {0}/{1}/{2}’.format(M, D, Y))  
   
class Person:  
 def \_\_init\_\_(self, name, job, pay):  
 self.job = job

Пример: проверка аргументов функций   
1145  
 self.pay = pay  
 @rangetest([1, 0.0, 1.0]) # giveRaise = rangetest(...)(giveRaise)  
 def giveRaise(self, percent): # Аргумент 0 – ссылка self на экземпляр  
 self.pay = int(self.pay \* (1 + percent))  
   
# Закомментированные строки возбуждают исключение TypeError, если сценарий   
# не был запущен командой “python -O”   
   
persinfo(‘Bob Smith’, 45) # В действительности вызывает onCall(...)  
#persinfo(‘Bob Smith’, 200) # или person, если был использован аргумент –O   
 # командной строки  
birthday(5, 31, 1963)   
#birthday(5, 32, 1963)  
   
sue = Person(‘Sue Jones’, ‘dev’, 100000)  
sue.giveRaise(.10) # В действительности вызывает onCall(self, .10)  
print(sue.pay) # или giveRaise(self, .10), если использован –O  
#sue.giveRaise(1.10)  
#print(sue.pay)  
Если запустить этот сценарий, допустимые вызовы выведут следующие резуль-  
таты (все примеры в этом разделе одинаково работают под управлением обеих   
версий Python, 2.6 и 3.0, потому что декораторы функций поддерживаются   
в обеих версиях� мы не используем прием делегирования атрибутов и исполь-  
зуем вызовы функции print и конструкции возбуждения исключений в стиле   
Python 3.0):  
C:\misc> C:\python30\python devtools\_test.py  
True  
Bob Smith is 45 years old  
birthday = 5/31/1963  
110000  
Если раскомментировать любой из недопустимых вызовов, декоратор будет   
возбуждать исключение TypeError. Ниже приводятся результаты запуска сце-  
нария, в котором были раскомментированы две последние строки (как обычно,   
я опустил часть сообщения об ошибке для экономии места):  
C:\misc> C:\python30\python devtools\_test.py  
True  
Bob Smith is 45 years old  
birthday = 5/31/1963  
110000  
TypeError: Argument 1 not in 0.0..1.0  
Если запустить интерпретатор с аргументом –O командной строки, проверка на   
вхождение в диапазон выполняться не будет, но при этом не будет и падения   
производительности, вызванного логикой обертки, – сценарий будет вызывать   
оригинальные, недекорированные функции. Так как мы решили, что этот де-  
коратор является инструментом отладки, мы можем использовать аргумент –O   
для оптимизации программы при работе в эксплуатационном режиме:  
C:\misc> C:\python30\python –O devtools\_test.py  
False  
Bob Smith is 45 years old  
birthday = 5/31/1963  
110000  
231000

1146   
Глава 38. Декораторы   
Обобщение на именованные аргументы   
и аргументы со значениями по умолчанию  
Предыдущая версия представляет собой основу, которую мы можем использо-  
вать, но она весьма ограниченна – она поддерживает проверку только позици-  
онных аргументов и не предусматривает проверку именованных аргументов   
(фактически она предполагает, что именованные аргументы вообще не будут   
передаваться, так как при их использовании может нарушаться порядок сле-  
дования позиционных аргументов). Кроме того, она ничего не делает с аргу-  
ментами, имеющими значения по умолчанию, которые могут быть опущены   
в том или ином вызове. Эта версия отлично подходит для случаев, когда все   
аргументы передаются по позициям и не используются аргументы со значени-  
ями по умолчанию, но она далека от идеала, если рассматривать ее как универ-  
сальный инструмент. Интерпретатор поддерживает множество более гибких   
режимов передачи аргументов, к которым мы пока не обращаемся.  
Ниже приводится измененная версия нашего примера, расширенная в этом от-  
ношении. Сопоставляя аргументы, ожидаемые обернутой функцией, с факти-  
ческими аргументами в вызове, она обеспечивает проверку аргументов, кото-  
рые могут передаваться по позиции или по имени, и пропускает проверку аргу-  
ментов со значениями по умолчанию, которые отсутствуют в вызове. Проще го-  
воря, проверяемые аргументы определяются в декораторе в виде именованных   
аргументов, которые затем будут отыскиваться в кортеже \*pargs позиционных   
аргументов и в словаре \*\*kargs именованных аргументов.  
“””  
Файл devtools.py: декоратор функций, выполняющий проверку аргументов на   
вхождение в заданный диапазон. Проверяемые аргументы передаются декоратору в   
виде именованных аргументов. В фактическом вызове функции аргументы могут   
передаваться как в виде позиционных, так и в виде именованных аргументов,  
при этом аргументы со значениями по умолчанию могут быть опущены.  
Примеры использования приводятся в файле devtools\_test.py.  
“””  
   
trace = True  
   
def rangetest(\*\*argchecks): # Проверяемые аргументы с диапазонами  
 def onDecorator(func): # onCall сохраняет func и argchecks  
 if not \_\_debug\_\_: # True – если “python –O main.py args...”  
 return func # Обертывание только при отладке; иначе   
 else: # возвращается оригинальная функция  
 import sys  
 code = func.\_\_code\_\_  
 allargs = code.co\_varnames[:code.co\_argcount]  
 funcname = func.\_\_name\_\_  
   
 def onCall(\*pargs, \*\*kargs):  
 # Все аргументы в кортеже pargs сопоставляются с первыми N   
 # ожидаемыми аргументами по позиции  
 # Остальные либо находятся в словаре kargs, либо опущены, как   
 # аргументы со значениями по умолчанию  
 positionals = list(allargs)  
 positionals = positionals[:len(pargs)]  
   
 for (argname, (low, high)) in argchecks.items():  
 # Для всех аргументов, которые должны быть проверены

Пример: проверка аргументов функций   
1147  
 if argname in kargs:  
 # Аргумент был передан по имени  
 if kargs[argname] < low or kargs[argname] > high:  
 errmsg = ‘{0} argument “{1}” not in {2}..{3}’  
 errmsg = errmsg.format(funcname, argname,   
 low, high)  
 raise TypeError(errmsg)  
   
 elif argname in positionals:  
 # Аргумент был передан по позиции  
 position = positionals.index(argname)  
 if pargs[position] < low or pargs[position] > high:  
 errmsg = ‘{0} argument “{1}” not in {2}..{3}’  
 errmsg = errmsg.format(funcname, argname,   
 low, high)  
 raise TypeError(errmsg)  
 else:  
 # Аргумент не был передан: предполагается, что он   
 # имеет значение по умолчанию  
 if trace:  
 print(‘Argument “{0}” defaulted’.format(argname))  
 return func(\*pargs, \*\*kargs) # OK: вызвать оригинальную   
 # функцию  
 return onCall  
 return onDecorator  
Следующий тестовый сценарий демонстрирует порядок использования декора-  
тора – аргументы, которые требуется проверить, передаются декоратору в виде   
именованных аргументов, а в фактическом вызове функции аргументы могут   
передаваться по имени или по позиции. Аргументы со значениями по умолча-  
нию могут быть опущены, даже если они определены для проверки:  
# Файл devtools\_test.py  
# Закомментированные строки возбуждают исключение TypeError, если сценарий   
# не был запущен командой “python -O”   
from devtools import rangetest  
   
# Тест вызовов функций с позиционными и именованными аргументами  
   
@rangetest(age=(0, 120)) # persinfo = rangetest(...)(persinfo)  
def persinfo(name, age):  
 print(‘%s is %s years old’ % (name, age))  
   
@rangetest(M=(1, 12), D=(1, 31), Y=(0, 2009))  
def birthday(M, D, Y):  
 print(‘birthday = {0}/{1}/{2}’.format(M, D, Y))  
   
persinfo(‘Bob’, 40)  
persinfo(age=40, name=’Bob’)  
birthday(5, D=1, Y=1963)  
#persinfo(‘Bob’, 150)  
#persinfo(age=150, name=’Bob’)  
#birthday(5, D=40, Y=1963)  
   
# Тест вызовов методов с позиционными и именованными аргументами  
   
class Person:  
 def \_\_init\_\_(self, name, job, pay):

1148   
Глава 38. Декораторы   
 self.job = job  
 self.pay = pay  
 # giveRaise = rangetest(...)(giveRaise)  
 @rangetest(percent=(0.0, 1.0)) # Аргумент percent передается по имени   
 def giveRaise(self, percent): # или по позиции  
 self.pay = int(self.pay \* (1 + percent))  
   
bob = Person(‘Bob Smith’, ‘dev’, 100000)  
sue = Person(‘Sue Jones’, ‘dev’, 100000)  
bob.giveRaise(.10)  
sue.giveRaise(percent=.20)  
print(bob.pay, sue.pay)  
#bob.giveRaise(1.10)  
#bob.giveRaise(percent=1.20)  
   
# Тест вызовов функций с опущенными аргументами по умолчанию  
   
@rangetest(a=(1, 10), b=(1, 10), c=(1, 10), d=(1, 10))  
def omitargs(a, b=7, c=8, d=9):  
 print(a, b, c, d)  
   
omitargs(1, 2, 3, 4)  
omitargs(1, 2, 3)  
omitargs(1, 2, 3, d=4)  
omitargs(1, d=4)  
omitargs(d=4, a=1)  
omitargs(1, b=2, d=4)  
omitargs(d=8, c=7, a=1)  
   
#omitargs(1, 2, 3, 11) # Недопустимое значение аргумента d  
#omitargs(1, 2, 11) # Недопустимое значение аргумента c  
#omitargs(1, 2, 3, d=11) # Недопустимое значение аргумента d  
#omitargs(11, d=4) # Недопустимое значение аргумента a  
#omitargs(d=4, a=11) # Недопустимое значение аргумента a  
#omitargs(1, b=11, d=4) # Недопустимое значение аргумента b  
#omitargs(d=8, c=7, a=11) # Недопустимое значение аргумента a  
Если запустить этот сценарий, передача значений вне заданного диапазона   
будет приводить к исключению, как и прежде, но теперь аргументы могут   
передаваться по имени или по позиции, а отсутствующие аргументы со значе-  
ниями по умолчанию не проверяются. Этот пример работает в обеих версиях   
Python, 2.6 и 3.0, но при выводе результатов в 2.6 они окружаются дополни-  
тельными круглыми скобками. Поэкспериментируйте с этим примером само-  
стоятельно и изучите полученные результаты – этот декоратор действует так   
же, как и предыдущая версия, но область его применения стала намного шире:  
C:\misc> C:\python30\python devtools\_test.py  
Bob is 40 years old  
Bob is 40 years old  
birthday = 5/1/1963  
110000 120000  
1 2 3 4  
Argument “d” defaulted  
1 2 3 9  
1 2 3 4  
Argument “c” defaulted  
Argument “b” defaulted

Пример: проверка аргументов функций   
1149  
1 7 8 4  
Argument “c” defaulted  
Argument “b” defaulted  
1 7 8 4  
Argument “c” defaulted  
1 2 8 4  
Argument “b” defaulted  
1 7 7 8  
Если раскомментировать любой из недопустимых вызовов, обнаруженная при   
проверке ошибка приведет к исключению, как и прежде (если интерпретатор   
не был запущен с параметром –O командной строки):  
TypeError: giveRaise argument “percent” not in 0.0..1.0  
Подробности реализации  
Эта реализация декоратора опирается на использование механизма интро-  
спекции и на ограничения механизма передачи аргументов. Для достижения   
полной обобщенности мы могли бы имитировать логику сопоставления аргу-  
ментов в языке Python, чтобы определить, какие аргументы передаются и в ка-  
Python, чтобы определить, какие аргументы передаются и в ка-  
, чтобы определить, какие аргументы передаются и в ка-  
ких режимах, но это было бы слишком сложно для нашего инструмента. Было   
бы проще сопоставлять аргументы, переданные по имени, с множеством всех   
имен ожидаемых аргументов, чтобы определить, какие позиционные аргумен-  
ты фактически присутствуют в данном вызове.  
Интроспекция функций  
Как оказывается, механизм интроспекции позволяет исследовать объекты   
функций, а связанные с функциями объекты программного кода обладают   
именно теми инструментами, которые нам необходимы. Эти инструменты   
были кратко представлены в главе 19, а здесь мы находим им практическое   
применение. Множество имен ожидаемых аргументов – это просто первые N   
имен переменных, присоединенных к объекту программного кода функции:  
# В Python 3.0 (и 2.6 для совместимости):  
>>> def func(a, b, c, d):  
... x = 1  
... y = 2  
...  
>>> code = func.\_\_code\_\_ # Объект с программным кодом,   
>>> code.co\_nlocals # принадлежащий объекту функции  
6  
>>> code.co\_varnames # Все имена локальных переменных  
(‘a’, ‘b’, ‘c’, ‘d’, ‘x’, ‘y’)  
>>> code.co\_varnames[:code.co\_argcount] # Первые N локальных имен – это  
(‘a’, ‘b’, ‘c’, ‘d’) # ожидаемые аргументы  
   
>>> import sys # Для обратной совместимости  
>>> sys.version\_info # [0] – старший номер версии  
(3, 0, 0, ‘final’, 0)  
>>> code = func.\_\_code\_\_ if sys.version\_info[0] == 3 else func.func\_code  
Аналогичные методы доступны и в более ранних версиях Python, но атрибут   
func.\_\_code\_\_ в версии 2.5 и ниже имеет имя func.func\_code (более новый атри-  
бут \_\_code\_\_ доступен также в версии 2.6 для обеспечения переносимости). До-

1150   
Глава 38. Декораторы   
полнительные сведения можно получить с помощью функции dir, передав ей   
объект функции или объект с программным кодом.  
Допущения относительно аргументов  
Помимо множества имен ожидаемых аргументов решение опирается также на   
два ограничения, которые накладываются интерпретатором на порядок следо-  
вания передаваемых аргументов (эти ограничения действуют в обеих версиях   
Python, 2.6 и 3.0):  
 •  
В вызове функции все позиционные аргументы предшествуют всем имено-  
ванным аргументам.  
 •  
В инструкции def все аргументы, не имеющие значений по умолчанию, ука-  
зываются перед всеми аргументами, имеющими значения по умолчанию.  
То есть в вызове функции позиционные аргументы не могут следовать за име-  
нованными аргументами, а аргументы, не имеющие значений по умолчанию,   
не могут следовать за аргументами, имеющими значения по умолчанию. Все   
определения аргументов вида name=value должны следовать за простыми аргу-  
ментами вида name.  
Чтобы упростить реализацию, мы можем также допустить, что сам вызов во-  
обще является допустимым, то есть либо все аргументы получат значения (по   
имени или по позиции), либо какие-то аргументы будут опущены преднаме-  
ренно, чтобы им были присвоены значения по умолчанию. Это допущение не   
обязательно будет соблюдаться, потому что на момент выполнения проверки   
оригинальная функция фактически еще не была вызвана – вызов функции мо-  
жет потерпеть неудачу, когда позднее она будет вызвана оберткой, из-за некор-  
ректной передачи аргументов. Однако подобные неудачи не имеют отрицатель-  
ных последствий для работы обертки, поэтому мы можем просто игнорировать   
допустимость самого вызова. Полная проверка допустимости вызова еще до   
того, как он будет произведен, могла бы потребовать от нас реализовать полную   
имитацию алгоритма сопоставления аргументов в языке Python, что слишком   
сложно для нашего инструмента.  
Алгоритм сопоставления  
Теперь с учетом этих ограничений и допущений мы можем реализовать алго-  
ритм обработки именованных аргументов и опущенных аргументов со значе-  
ниями по умолчанию. Перехватив вызов функции, мы можем исходить из сле-  
дующих предположений:  
 •  
Все N переданных позиционных аргументов в \*pargs должны соответство-  
вать первым N ожидаемым аргументам, перечисленным в объекте с про-  
граммным кодом функции. Это следует из правил, определяющих порядок   
следования аргументов в вызовах функций, обозначенных выше, посколь-  
ку все позиционные аргументы предшествуют именованным.  
 •  
Чтобы получить имена аргументов, которые фактически были переданы   
в виде позиционных аргументов, можно извлечь срез из списка всех ожи-  
даемых аргументов длиной N, равной длине кортежа \*pargs с позиционными   
аргументами.  
 •  
Любые аргументы, следующие за первыми N ожидаемыми аргументами, бу-  
дут либо именованными, либо аргументами со значениями по умолчанию,   
которые могут быть опущены в вызове функции.

Пример: проверка аргументов функций   
1151  
 •  
Для каждого имени аргумента из числа тех, что требуется проверить: если   
имя присутствует в словаре \*\*kargs, следовательно, аргумент был передан   
по имени, а если имя попадает в число первых N ожидаемых аргументов,   
следовательно, соответствующий аргумент был передан как позиционный   
(в этом случае относительная позиция в списке ожидаемых аргументов со-  
ответствует относительной позиции в \*pargs). В противном случае можно   
предположить, что аргумент был опущен при вызове функции и имеет зна-  
чение по умолчанию, которое не требуется проверять.  
Другими словами, мы можем пропустить проверку аргументов, которые были   
опущены при вызове функции, исходя из того, что первые N фактически пере-  
данных позиционных аргументов в \*pargs должны соответствовать первым N   
именам аргументов в списке всех ожидаемых аргументов и что все остальные   
аргументы либо должны передаваться по именам и потому попасть в \*\*kargs,   
либо имеют значения по умолчанию и могут быть опущены. Следуя этой схеме,   
декоратор просто будет пропускать проверку любых аргументов, которые на-  
ходятся между самым правым позиционным аргументом и самым левым име-  
нованным аргументом, между именованными аргументами или после самого   
правого позиционного аргумента вообще. Изучите исходные тексты декоратора   
и тестового сценария, чтобы понять, как это воплощается в программный код.  
Нерешенные проблемы  
Несмотря на то, что наш инструмент проверки аргументов действует так, как   
и задумывалось, тем не менее, остаются нерешенными две проблемы. Во-  
первых, как уже упоминалось выше, если вызов оригинальной функции ока-  
жется недопустимым, он будет терпеть неудачу в конце декоратора. Например,   
следующие два вызова приводят к исключению:  
omitargs()  
omitargs(d=8, c=7, b=6)  
Однако они терпят неудачу только в тот момент, когда производится вызов ори-  
гинальной функции, в конце обертки. Мы могли бы попытаться сымитировать   
алгоритм сопоставления аргументов в языке Python, чтобы избежать этого,   
однако для этого нет никаких веских причин – поскольку так или иначе вы-  
зов все равно потерпел бы неудачу, мы можем доверить интерпретатору самому   
обнаружить проблемы, которые не удалось обнаружить нам.  
Наконец, хотя наша последняя версия обрабатывает позиционные и именован-  
ные аргументы, а также опущенные аргументы со значениями по умолчанию,   
тем не менее, она никак не обрабатывает аргументы \*args и \*\*args, которые   
могут использоваться в декорируемой функции для приема произвольного ко-  
личества аргументов. Впрочем, учитывая поставленную цель, мы не должны   
беспокоиться об этом:  
 •  
При передаче дополнительного именованного аргумента его имя попадет   
в словарь \*\*kargs и будет проверено, если оно упоминается в декораторе.   
 •  
Если дополнительный именованный аргумент не будет передан, его имя не   
попадет ни в словарь \*\*kargs, ни в срез со списком ожидаемых позицион-  
ных аргументов и потому просто не будет проверяться. Этот аргумент будет   
интерпретироваться как аргумент со значением по умолчанию, даже если   
в действительности он является необязательным дополнительным аргу-  
ментом.

1152   
Глава 38. Декораторы   
 •  
При передаче дополнительного позиционного аргумента у нас нет никакого   
способа сослаться на него в декораторе – его имя не попадет ни в словарь   
\*\*kargs, ни в срез со списком ожидаемых аргументов, поэтому он просто бу-  
дет пропущен. Поскольку такие аргументы не перечисляются в определе-  
нии функции, нет никакой возможности отобразить в декораторе его имя   
на относительную позицию в списке ожидаемых аргументов.  
Другими словами, этот декоратор поддерживает возможность проверки произ-  
вольных именованных аргументов, но он не поддерживает возможность про-  
верки произвольных позиционных аргументов, не имеющих имени и потому   
не имеющих определенной позиции в сигнатуре функции.  
В принципе, мы могли бы расширить интерфейс декоратора поддержкой кон-  
струкции \*args в декорируемой функции для тех редких случаев, когда это   
может оказаться полезным (например, добавив специальное имя аргумента   
с диапазоном, на соответствие которому проверялись бы все аргументы в спи-  
ске \*pargs обертки, с позициями, превышающими длину списка ожидаемых   
аргументов). Однако мы уже исчерпали объем книги, отведенный для этого   
примера, поэтому, если данное улучшение представляет для вас интерес, вы   
можете реализовать его в качестве самостоятельного упражнения.  
Аргументы декораторов и аннотации функций  
Интересно отметить, что возможность создания аннотаций функций, появив-  
шаяся в Python 3.0, может использоваться как альтернатива аргументам деко-  
Python 3.0, может использоваться как альтернатива аргументам деко-  
 3.0, может использоваться как альтернатива аргументам деко-  
ратора в нашем примере проверки значений аргументов на вхождение в задан-  
ный диапазон. Как мы узнали в главе 19, аннотации позволяют ассоциировать   
выражения с аргументами и возвращаемыми значениями за счет внедрения их   
в строку заголовка инструкции def. Интерпретатор собирает аннотации в сло-  
варь и присоединяет его к аннотированной функции.  
Мы могли бы использовать эту возможность в нашем примере, определив гра-  
ницы диапазонов в строке заголовка, вместо аргументов декоратора. Правда,   
при этом нам все еще будет необходим декоратор для обертывания функции,   
чтобы с помощью обертки перехватывать вызовы, но по сути мы лишь заменим   
декоратор с аргументами:  
@rangetest(a=(1, 5), c=(0.0, 1.0))  
def func(a, b, c): # func = rangetest(...)(func)  
 print(a + b + c)  
на простой декоратор для аннотированной функции:  
@rangetest  
def func(a:(1, 5), b, c:(0.0, 1.0)):  
 print(a + b + c)  
То есть теперь границы диапазона определяются уже не во внешнем программ-  
ном коде, а в самой функции. Следующий сценарий иллюстрирует структуру   
декораторов для каждого из двух случаев с неполной, схематической реализа-  
цией. Шаблон декоратора с аргументами представляет полное решение, пока-  
занное выше. Альтернатива, основанная на использовании аннотаций, имеет   
на один уровень вложенности меньше, потому что ей не требуется сохранять   
аргументы декоратора:

Пример: проверка аргументов функций   
1153  
# С использованием декоратора с аргументами  
   
def rangetest(\*\*argchecks):  
 def onDecorator(func):  
 def onCall(\*pargs, \*\*kargs):  
 print(argchecks)  
 for check in argchecks: pass # Добавьте проверку сюда  
 return func(\*pargs, \*\*kargs)  
 return onCall  
 return onDecorator  
   
@rangetest(a=(1, 5), c=(0.0, 1.0))  
def func(a, b, c): # func = rangetest(...)(func)  
 print(a + b + c)  
   
func(1, 2, c=3) # Вызовет onCall, argchecks – в области   
 #видимости объемлющей функции  
   
# С использованием аннотаций функций  
   
def rangetest(func):  
 def onCall(\*pargs, \*\*kargs):  
 argchecks = func.\_\_annotations\_\_  
 print(argchecks)  
 for check in argchecks: pass # Добавьте проверку сюда  
 return func(\*pargs, \*\*kargs)  
 return onCall  
   
@rangetest  
def func(a:(1, 5), b, c:(0.0, 1.0)): # func = rangetest(func)  
 print(a + b + c)  
   
func(1, 2, c=3) # Вызовет onCall, аннотации в функции func  
В процессе работы обе обертки получают доступ к одной и той информации   
о проверяемых аргументах и диапазонах, но хранится она в разных формах:   
в версии декоратора с аргументами информация сохраняется в аргументах,   
в области видимости объемлющей функции, а в версии с аннотациями инфор-  
мация сохраняется в атрибутах самой функции:  
{‘a’: (1, 5), ‘c’: (0.0, 1.0)}  
6  
{‘a’: (1, 5), ‘c’: (0.0, 1.0)}  
6  
Я не буду подробно излагать реализацию версии, основанной на использовании   
аннотаций, и оставлю ее вам в качестве самостоятельного упражнения – она   
будет идентична реализации полного решения, представленного выше, потому   
что в этой версии информация о диапазонах просто переместилась из области   
видимости объемлющей функции в аргументы самой функции. В действитель-  
ности, все, что мы получили в результате, – это лишь другой интерфейс для   
нашего инструмента. Он по-прежнему должен сопоставлять имена аргументов   
с именами ожидаемых аргументов для получения относительных позиций.  
Фактически использование аннотаций вместо аргументов декоратора в этом   
примере лишь ограничивает  область  его  применения. С одной стороны, ан-  
нотации доступны только в Python 3.0, поэтому в версии 2.6 этот декоратор

1154   
Глава 38. Декораторы   
не может использоваться. С другой стороны, декораторы с аргументами могут   
применяться в обеих версиях.  
Еще более важно, что переместив определения границ в заголовок инструкции   
def, мы по сути отводим функции единственную роль – аннотации позволяют   
указать для каждого аргумента только одно выражение, которое может пре-  
следовать только одну цель. Например, мы не сможем использовать аннота-  
ции, чтобы определить для функции другие роли.  
С другой стороны, аргументы декоратора определяются за пределами самой   
функции, их проще удалить и они обеспечивают более  широкие возможно-  
сти – реализация самой функции не подразумевает единственное назначение   
декоратора. Фактически за счет вложения декораторов с аргументами мы мо-  
жем предусмотреть несколько уровней декорирования одной и той же функ-  
ции – аннотации поддерживают только один. Кроме того, при использовании   
декоратора с аргументами сама функция сохраняет простое и привычное   
оформление.  
Однако если вы отводите функции единственную роль и можете ограничиться   
поддержкой только Python 3.X, то выбор между аннотациями и декоратора-  
Python 3.X, то выбор между аннотациями и декоратора-  
 3.X, то выбор между аннотациями и декоратора-  
X, то выбор между аннотациями и декоратора-  
, то выбор между аннотациями и декоратора-  
ми с аргументами остается лишь делом вкуса. Как часто бывает в жизни, для   
одного аннотации являются благом, для другого – синтаксическим мусором....  
Другие области применения:   
проверка типов (если настаиваете!)  
Шаблон реализации обработки аргументов в декораторах, который мы вырабо-  
тали выше, можно было бы использовать и в других целях. Например, совсем   
несложно реализовать проверку типов аргументов, которую можно было бы ис-  
пользовать на этапе разработки:  
def typetest(\*\*argchecks):  
 def onDecorator(func):  
 ...  
 def onCall(\*pargs, \*\*kargs):  
 positionals = list(allargs)[:len(pargs)]  
 for (argname, type) in argchecks.items():  
 if argname in kargs:  
 if not isinstance(kargs[argname], type):  
 ...  
 raise TypeError(errmsg)  
 elif argname in positionals:  
 position = positionals.index(argname)  
 if not isinstance(pargs[position], type):  
 ...  
 raise TypeError(errmsg)  
 else:  
 # Предполагается, что аргумент был опущен:   
 # аргумент со значением по умолчанию  
 return func(\*pargs, \*\*kargs)  
 return onCall  
 return onDecorator  
   
@typetest(a=int, c=float)  
def func(a, b, c, d): # func = typetest(...)(func)

В заключение   
1155  
 ...  
   
func(1, 2, 3.0, 4) # Корректный вызов  
func(‘spam’, 2, 99, 4) # Возбудит исключение, как и ожидалось  
Фактически мы могли бы пойти по пути обобщения еще дальше и организовать   
передачу функции проверки, как мы сделали это ранее, при реализации деко-  
ратора Public, – единственной копии программного кода такого рода было бы   
достаточно, чтобы организовать и проверку значений аргументов, и проверку   
их типов. Если в этом декораторе вместо аргументов использовать аннотации,   
как описано в предыдущем разделе, это сделало бы аннотации похожими на   
объявления типов в других языках программирования:  
@typetest  
def func(a: int, b, c: float, d): # func = typetest(func)  
 ... # Ух ты!...  
Однако, как вы уже узнали в этой книге, на практике вообще не принято про-  
верять типы объектов, это не характерно для языка Python (на практике та-  
Python (на практике та-  
 (на практике та-  
кого рода проверки выявляют программистов на языке C++, начинающих   
изучать Python).  
Проверки типов ограничивают область применения ваших функций узким   
кругом объектов определенных типов и не позволяют им взаимодействовать   
с объектами других типов, имеющими совместимые интерфейсы. В результа-  
те это накладывает ненужные ограничения на программный код и снижает   
его гибкость. С другой стороны, из каждого правила есть исключения – про-  
верка типов может пригодиться в отдельных случаях при отладке и при ор-  
ганизации взаимодействий с программами, написанными на более строгих   
языках программирования, таких как C++. Данный обобщенный шаблон об-  
C++. Данный обобщенный шаблон об-  
++. Данный обобщенный шаблон об-  
работки аргументов также можно было бы использовать для решения самых   
разно образных задач.  
В заключение  
В этой главе мы исследовали обе разновидности декораторов – декораторы   
функций и классов. Как мы узнали, декораторы предоставляют возможность   
добавлять программный код, который будет вызываться автоматически при   
определении функции или класса. Когда применяется декоратор, интерпре-  
татор присваивает имени функции или класса вызываемый объект, который   
возвращается декоратором. Этот прием позволяет нам добавлять в вызовы   
функций или в операции создания новых экземпляров классов дополнитель-  
ный слой обертывающей логики, управляющей функциями и экземплярами.   
Кроме того, мы видели, что использование управляющих функций и присваи-  
вание возвращаемых вызываемых объектов именам функций и классов вруч-  
ную позволяет достичь того же эффекта, но декораторы предлагают более оче-  
видное и единообразное решение.  
Как вы увидите в следующей главе, декораторы классов могут также использо-  
ваться для управления не только экземплярами, но и самими классами. Одна-  
ко данная возможность тесно пересекается с метаклассами, темой следующей   
главы, поэтому вам придется двинуться дальше, чтобы узнать окончание этой   
истории. Но прежде ответьте на контрольные вопросы, следующие ниже. По-

1156   
Глава 38. Декораторы   
скольку основная часть этой главы была посвящена крупным примерам, в кон-  
трольных вопросах вам будет предложено внести некоторые изменения в их   
реализацию с целью закрепить полученные знания.  
Закрепление пройденного  
Контрольные вопросы  
1. Как упоминалось в одном из примечаний в этой главе, декоратор с аргу-  
ментами, реализующий хронометраж функций, который мы написали   
в разделе «Добавление аргументов декоратора» выше, может применяться   
только к простым функциям, потому что для перехвата вызовов функций   
он использует вложенный класс с методом \_\_call\_\_ перегрузки операторов.   
Этот декоратор не может применяться к методам класса, потому что в этом   
случае в аргументе self будет передаваться экземпляр декоратора, а не под-  
разумеваемый экземпляр класса. Перепишите этот декоратор так, чтобы его   
можно было применять и к простым функциям, и к методам класса, и про-  
тестируйте его с функциями и методами. (Подсказка: прочитайте раздел   
«Ошибки при использовании классов I: декорирование методов классов»   
выше, где приводятся некоторые рекомендации.) Обратите внимание, что   
для сохранения общего времени выполнения можно использовать атрибуты   
объекта функции, так как в вашем распоряжении уже не будет вложенного   
класса, где можно было бы сохранять информацию, и не будет возможности   
обращаться к нелокальным переменным из-за пределов декоратора.  
2. Декораторы классов Public/Private, которые мы написали в этой главе, до-  
бавляют дополнительную нагрузку к операциям извлечения атрибутов де-  
корируемых классов. Конечно, для повышения скорости мы можем просто   
удалить строку с объявлением декоратора @, но точно так же можно было бы   
дополнить сам декоратор проверкой переменной \_\_debug\_\_ и не выполнять   
обертывание вообще, когда интерпретатор запускается с флагом –O (как мы   
сделали это в декораторах, реализующих проверку значений аргументов).   
При таком подходе мы сможем поднять скорость выполнения программы   
с помощью аргументов командной строки (python –O main.py...), не изменяя   
исходных текстов. Реализуйте и протестируйте это расширение.  
Ответы  
1. Ниже приводится один из способов решения этой задачи и результаты, по-  
лученные при его выполнении (методы представленного класса выполня-  
ются очень быстро). Вся хитрость состоит в том, чтобы заменить вложен-  
ный класс вложенной функцией и тем самым обеспечить передачу в аргу-  
менте self экземпляра самого класса, и сохранить общее время выполнения   
в атрибутах функции, благодаря чему оно сможет быть извлечено позднее,   
обращением к оригинальному имени (дополнительные подробности вы най-  
дете в разделе «Способы сохранения информации о состоянии» выше, в этой   
же главе� функции поддерживают возможность присоединения к ним лю-  
бых атрибутов, а имя функции в данном случае является ссылкой на объ-  
емлющую область видимости).  
import time  
   
def timer(label=’’, trace=True): # Декоратор с аргументами: сохраняет арг.

Закрепление пройденного   
1157  
 def onDecorator(func): # На этапе декорирования @: сохраняет   
 # декорируемую функцию  
 def onCall(\*args, \*\*kargs): # При вызове: вызывает оригинал  
 start = time.clock() # Информация в области видимости +   
 result = func(\*args, \*\*kargs) # атрибуты функции  
 elapsed = time.clock() – start  
 onCall.alltime += elapsed  
 if trace:  
 format = ‘%s%s: %.5f, %.5f’  
 values = (label, func.\_\_name\_\_, elapsed, onCall.alltime)  
 print(format % values)  
 return result  
 onCall.alltime = 0  
 return onCall  
 return onDecorator  
   
# Проверка на функциях  
   
@timer(trace=True, label=’[CCC]==>’)  
def listcomp(N): # listcomp = timer(...)(listcomp)  
 return [x \* 2 for x in range(N)] # listcomp(...) вызовет onCall  
   
@timer(trace=True, label=’[MMM]==>’)  
def mapcall(N):  
 return list(map((lambda x: x \* 2), range(N))) # list() для представлений   
 # в 3.0   
for func in (listcomp, mapcall):  
 result = func(5) # Время этого вызова, всех вызовов и возвр. значение  
 func(5000000)  
 print(result)  
 print(‘allTime = %s\n’ % func.alltime) # Общее время всех вызовов  
   
# Проверка на методах  
   
class Person:  
 def \_\_init\_\_(self, name, pay):  
 self.name = name  
 self.pay = pay  
   
 @timer()  
 def giveRaise(self, percent): # giveRaise = timer()(giveRaise)  
 self.pay \*= (1.0 + percent) # декоратор сохранит giveRaise  
   
 @timer(label=’\*\*’)  
 def lastName(self): # lastName = timer(...)(lastName)  
 return self.name.split()[-1] # общее время для класса, а не экз.  
   
bob = Person(‘Bob Smith’, 50000)  
sue = Person(‘Sue Jones’, 100000)  
bob.giveRaise(.10)  
sue.giveRaise(.20) # runs onCall(sue, .10)  
print(bob.pay, sue.pay)  
print(bob.lastName(), sue.lastName()) # runs onCall(bob), remembers lastName  
print(‘%.5f %.5f’ % (Person.giveRaise.alltime, Person.lastName.alltime))  
   
# Ожидаемые результаты  
   
[CCC]==>listcomp: 0.00002, 0.00002  
[CCC]==>listcomp: 1.19636, 1.19638

1158   
Глава 38. Декораторы   
[0, 2, 4, 6, 8]  
allTime = 1.19637775192  
[MMM]==>mapcall: 0.00002, 0.00002  
[MMM]==>mapcall: 2.29260, 2.29262  
[0, 2, 4, 6, 8]  
allTime = 2.2926232943  
   
giveRaise: 0.00001, 0.00001  
giveRaise: 0.00001, 0.00002  
55000.0 120000.0  
\*\*lastName: 0.00001, 0.00001  
\*\*lastName: 0.00001, 0.00002  
Smith Jones  
0.00002 0.00002  
2. Следующая реализация декоратора удовлетворяет условиям второго во-  
проса – при выполнении в оптимизированном режиме (-O) она возвращает   
оригинальный класс, поэтому операции обращения к атрибутам будут вы-  
полняться без потери скорости. В действительности, все, что я сделал, – это   
добавил инструкции проверки работы в отладочном режиме и увеличил   
отступы в определении класса. Добавьте определения методов перегрузки   
операторов в класс-обертку, если вам потребуется обеспечить их делегиро-  
вание клиентскому классу в Python 3.0 (обращения к этим методам в вер-  
сии 2.6 приводят к вызову метода \_\_getattr\_\_, но в версии 3.0 и при исполь-  
зовании классов нового стиля в версии 2.6 – нет).  
traceMe = False  
def trace(\*args):  
 if traceMe: print(‘[‘ + ‘ ‘.join(map(str, args)) + ‘]’)  
   
def accessControl(failIf):  
 def onDecorator(aClass):  
 if not \_\_debug\_\_:  
 return aClass  
 else:  
 class onInstance:  
 def \_\_init\_\_(self, \*args, \*\*kargs):  
 self.\_\_wrapped = aClass(\*args, \*\*kargs)  
 def \_\_getattr\_\_(self, attr):  
 trace(‘get:’, attr)  
 if failIf(attr):  
 raise TypeError(‘private attribute fetch: ‘ + attr)  
 else:  
 return getattr(self.\_\_wrapped, attr)  
 def \_\_setattr\_\_(self, attr, value):  
 trace(‘set:’, attr, value)  
 if attr == ‘\_onInstance\_\_wrapped’:  
 self.\_\_dict\_\_[attr] = value  
 elif failIf(attr):  
 raise TypeError(‘private attribute change: ‘ + attr)  
 else:  
 setattr(self.\_\_wrapped, attr, value)  
 return onInstance  
 return onDecorator  
   
def Private(\*attributes):  
 return accessControl(failIf=(lambda attr: attr in attributes))

Закрепление пройденного   
1159  
   
def Public(\*attributes):  
 return accessControl(failIf=(lambda attr: attr not in attributes))  
   
# Проверка: выделите следующий код в отдельный файл, чтобы декоратор можно   
# было использовать в других модулях  
   
@Private(‘age’) # Person = Private(‘age’)(Person)  
class Person: # Person = onInstance с информацией о состоянии  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
   
self.age = age # Доступ изнутри разрешен всегда  
   
X = Person(‘Bob’, 40)  
print(X.name) # Доступ снаружи контролируется  
X.name = ‘Sue’  
print(X.name)  
#print(X.age) # ОШИБКА, если сценарий не был запущен как “python -O”  
#X.age = 999 # то же самое  
#print(X.age) # то же самое  
   
@Public(‘name’)  
class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
   
X = Person(‘bob’, 40) # X – это onInstance  
print(X.name) # onInstance встраивает Person  
X.name = ‘Sue’  
print(X.name)  
#print(X.age) # ОШИБКА, если сценарий не был запущен как “python -O”  
#X.age = 999 # то же самое  
#print(X.age) # то же самое

Глава 39.  
   
Метаклассы  
В предыдущей главе мы исследовали декораторы и рассмотрели различные   
примеры их использования. В заключительной главе этой книги мы продол-  
жим наше знакомство со средствами создания инструментов и исследуем еще   
одну сложную тему: метаклассы.  
В некотором смысле метаклассы просто расширяют модель добавления про-  
граммного кода, предлагаемую декораторами. Как мы узнали в предыдущей   
главе, декораторы функций и классов позволяют перехватывать вызовы функ-  
ций и операции создания экземпляров и выполнять дополнительные действия.   
Точно так же и метаклассы позволяют перехватывать операции создания клас-  
сов – они предоставляют возможность добавлять дополнительный программ-  
ный код, который будет выполняться в конце инструкции class, хотя и иначе,   
чем декораторы. Кроме того, они реализуют обобщенный протокол управления   
объектами классов в программе.  
Подобно всем другим темам, обсуждаемым в этом разделе книги, тема мета-  
классов является дополнительной и может изучаться по мере необходимости.   
С практической точки зрения метаклассы позволяют получить более пол-  
ный контроль над работой множеств классов. Это очень мощный инструмент,   
к тому же разработанный, в общем, не для использования основной массой   
прикладных программистов (и, по правде сказать, не для слабонервных!).  
С другой стороны, метаклассы открывают дверь к различным шаблонам про-  
ектирования, которые очень сложно, если вообще возможно, реализовать дру-  
гими средствами. Они будут особенно интересны программистам, занимаю-  
щимся разработкой библиотек с гибкими прикладными интерфейсами или   
инструментов программирования для других программистов. Даже если вы не   
относитесь к этой категории программистов, тем не менее, изучение метаклас-  
сов поможет вам глубже понять, как устроена модель классов в языке Python.  
Как и в предыдущей главе, наша цель отчасти состоит в том, чтобы рассмо-  
треть более реалистичные примеры программного кода. Несмотря на то, что   
метаклассы являются частью ядра языка и не имеют прямого отношения   
к прикладному программированию, тем не менее, еще одна цель этой главы   
состоит в том, чтобы разжечь в вас интерес к исследованию крупных приклад-  
ных программ после того, как вы закончите читать эту книгу.

Нужны или не нужны метаклассы   
1161  
Нужны или не нужны метаклассы  
Метаклассы являются, пожалуй, самой сложной темой в этой книге, если не   
всего языка Python. Ниже приводится цитата из новостной группы comp.lang.  
python, принадлежащая давнему разработчику Python Тиму Петерсу (Tim Pe-  
Python Тиму Петерсу (Tim Pe-  
 Тиму Петерсу (Tim Pe-  
Tim Pe-  
 Pe-  
Pe-  
ters) (который также является автором известного девиза Python: «импорти-  
руй это»):  
[Метаклассы] – это гораздо более серьезная тема, которая не затрагива-  
ет 99% пользователей. Если вы задаетесь вопросом – нужны ли вам мета-  
классы, то, скорее всего, – не нужны (те, кому они действительно нужны,   
точно знают об этом и им не нужно объяснять – зачем они нужны).  
Другими словами, метаклассы в первую очередь предназначены для програм-  
мистов, занимающихся созданием библиотек и инструментов, которыми будут   
пользоваться другие. Во многих (если не в большинстве) случаях они являются   
не самым лучшим выбором для использования в прикладных программах. Это   
особенно верно, если вы разрабатываете программы, которые в будущем будут   
сопровождаться другими людьми. Использование чего-то «просто потому, что   
это круто» – не самое веское оправдание, если только вы не занимаетесь экс-  
периментами или изучением.  
Тем не менее метаклассы могут использоваться в самых разных целях, и поэто-  
му важно знать, когда они могут быть полезны. Например, метаклассы могут   
использоваться для расширения классов такими возможностями, как трас-  
сировка, сохранение в файлах, регистрация исключений и многими други-  
ми. Они могут также использоваться для конструирования отдельных частей   
классов во время выполнения, опираясь на файлы с настройками, позволяют   
единообразно применять декораторы функций ко всем методам класса, прове-  
рять на соответствие ожидаемым интерфейсам и так далее.  
В самых грандиозных воплощениях метаклассы могут использоваться даже   
для реализации альтернативных шаблонов проектирования, таких как аспект-  
но-ориентированное программирование, объектно-реляционные отображения   
(object/relational mappers, ORM) для баз данных и многого другого. Несмотря   
не то, что тех же результатов часто можно добиться другими способами (как   
мы увидим далее, области применения декораторов и метаклассов часто пере-  
секаются), тем не менее, метаклассы предоставляют формальную модель, при-  
способленную для решения этих задач. В этой книге недостаточно места, чтобы   
исследовать все возможные области применения метаклассов, но после изуче-  
ния основ в этой книге вам определенно имеет смысл поискать в Сети дополни-  
тельные варианты их использования.  
Пожалуй, самым подходящим обоснованием обращения к метаклассам в этой   
книге является то, что рассмотрение этой темы поможет вам снять покров тай-  
ны с механизма классов в языке Python. Даже если вы не будете использовать   
их в своей работе, беглое знакомство с метаклассами поможет вам глубже по-  
нять язык Python.  
Углубляемся в магию  
Основное внимание в этой книге уделялось простым приемам прикладного   
программирования, так как большинство программистов решают свои задачи   
за счет создания модулей, функций и классов. Они могут использовать классы   
и создавать экземпляры и иногда могут даже использовать методы перегрузки

1162   
Глава 39. Метаклассы   
операторов, но едва ли им требуется слишком глубоко вникать в особенности   
работы классов.  
Однако в этой книге мы также видели различные инструменты, которые по-  
зволяют разными способами управлять поведением интерпретатора и которые   
имеют отношение скорее к внутренней организации Python или к инструмен-  
Python или к инструмен-  
 или к инструмен-  
тальным средствам, чем к сфере прикладного программирования:  
Атрибуты механизма интроспекции  
Специальные атрибуты, такие как \_\_class\_\_ и \_\_dict\_\_, позволяют нам по-  
лучать информацию о внутреннем устройстве объектов Python и обрабаты-  
Python и обрабаты-  
 и обрабаты-  
вать их обобщенными способами – вывести перечень атрибутов объекта,   
имя класса и так далее.  
Методы перегрузки операторов  
Методы классов со специальными именами, такие как \_\_str\_\_ и \_\_add\_\_,   
позволяют перехватывать и определять поведение экземпляров классов   
при применении к ним встроенных операций, таких как вывод, операторы   
выражений и других. Они вызываются автоматически в ответ на попытку   
выполнить встроенную операцию и позволяют создавать классы, соответ-  
ствующие ожидаемым интерфейсам.  
Методы обработки обращений к атрибутам  
Специальная категория методов перегрузки операторов обеспечивает воз-  
можность перехватывать попытки доступа к атрибутам экземпляра: мето-  
ды \_\_getattr\_\_, \_\_setattr\_\_ и \_\_getattribute\_\_ позволяют классам-оберткам   
добавлять автоматически вызываемый программный код, который может   
проверять допустимость обращений к атрибутам и делегировать их выпол-  
нение встроенным объектам. С их помощью можно управлять доступом   
к любому количеству атрибутов – как к отдельным атрибутам, так и ко   
всем сразу, – значения которых могут вычисляться непосредственно в мо-  
мент доступа.  
Свойства классов  
Встроенная функция property позволяет ассоциировать программный код   
с определенным атрибутом, который будет автоматически вызываться при   
попытке получить значение атрибута, присвоить ему новое значение или   
удалить его. Хотя этот инструмент не настолько универсальный, как опи-  
санные в предыдущем абзаце, тем не менее, свойства позволяют организо-  
вать автоматический вызов программного кода при попытке обращения   
к отдельным атрибутам.  
Дескрипторы атрибутов классов  
В действительности, функция property – это простейший способ задать   
дескриптор атрибута, который автоматически вызывает функции, управ-  
ляющие доступом к атрибуту. Дескрипторы позволяют определить методы-  
обработчики \_\_get\_\_, \_\_set\_\_ и \_\_delete\_\_ в отдельном классе, которые ав-  
томатически вызываются при обращении к атрибуту, которому присвоен   
экземпляр этого класса. Они предоставляют способ добавить программный   
код, вызываемый автоматически при обращении к определенному атрибу-  
ту, и вызываются после того, как атрибут будет найден обычной процеду-  
рой поиска.

Нужны или не нужны метаклассы   
1163  
Декораторы функций и классов  
Как мы уже видели в главе 38, специальный синтаксис @callable декорато-  
ров позволяет добавлять логику, которая будет запускаться автоматически   
при вызове функции или при выполнении операции создания экземпляра.   
Эта обертывающая логика может выполнять трассировку или хронометраж   
вызовов, проверять аргументы, управлять всеми экземплярами класса, до-  
бавлять в экземпляры дополнительные особенности, такие как управление   
доступом к атрибутам, и многое другое. Синтаксис декораторов вставляет   
логику повторного присваивания имени, которая вызывается в конце ин-  
струкций определения функций и классов, – именам декорируемых функ-  
ций и классов присваиваются вызываемые объекты, перехватывающие по-  
следующие вызовы.  
Как упоминалось во введении к этой главе, метаклассы являются продолже-  
нием этой истории – они позволяют добавлять логику, которая автоматически   
вызывается при создании классов, в конце инструкции class. Эта логика не   
присваивает оригинальному имени класса вызываемый объект декоратора,   
а делегирует операцию создания самого класса специализированной логике.  
Другими словами, метаклассы – это всего лишь другой способ, позволяющий   
определить автоматически вызываемый программный код. Посредством ме-  
таклассов и других инструментов, перечисленных выше, Python предостав-  
Python предостав-  
 предостав-  
ляет способы вставлять логику обработки различных операций – вычисление   
выражений, обращение к атрибутам, вызовы функций, создание экземпляров   
классов, а теперь и создание объектов классов.  
В отличие от декораторов классов, которые обычно добавляют логику, вызы-  
ваемую на этапе создания экземпляров, метаклассы выполняются на этапе соз-  
дания классов – они представляют собой обработчики, которые используются   
для управления классами, а не их экземплярами.  
Например, метаклассы могут использоваться для автоматического декори-  
рования всех методов классов, регистрации всех классов для использования   
в библиотеках, автоматического добавления логики к классам, создания или   
расширения классов на основе упрощенных спецификаций в текстовых фай-  
лах и так далее. Благодаря тому, что метаклассы могут управлять процессом   
создания классов (и, как следствие, поведением, которое приобретают их эк-  
земпляры), область их применения чрезвычайно широка.  
Как мы уже видели, многие из этих дополнительных инструментов языка Py-  
Py-  
thon часто имеют пересекающиеся области применения. Например, управле-  
ние атрибутами часто может быть реализовано с помощью свойств, дескрипто-  
ров или методов управления атрибутами. Как мы увидим далее в этой главе,   
декораторы классов и метаклассы также часто оказываются взаимозаменяе-  
мыми. Декораторы классов чаще всего используются для управления экзем-  
плярами, тем не менее, они могут использоваться также и для управления   
классами. Аналогично, метаклассы предназначены для расширения конструк-  
ции классов, но они также часто используются для добавления программного   
кода, управляющего экземплярами. Поскольку выбор той или иной используе-  
мой техники иногда зависит исключительно от личных предпочтений, знание   
альтернатив может помочь вам выбрать правильный инструмент для решения   
той или иной задачи.

1164   
Глава 39. Метаклассы   
Недостатки «вспомогательных» функций  
Кроме того, подобно декораторам, о которых рассказывалось в предыдущей   
главе, с теоретической точки зрения, метаклассы являются необязательны-  
ми к применению инструментами. Обычно тех же результатов можно добить-  
ся, передавая объекты классов управляющим функциям (иногда их называют   
«вспомогательными»), практически так же, как можно достичь целей, пресле-  
дуемых декораторами, передавая управляющим функциям объекты функций   
и экземпляров. Однако точно так же, как и декораторы, метаклассы:  
 •  
Обеспечивают более формальный и очевидный способ управления.  
 •  
Помогают гарантировать, что прикладные программисты не забудут рас-  
ширить свои классы в соответствии с требованиями прикладного интер-  
фейса.  
 •  
Снижают избыточность программного кода и упрощают его сопровождение   
за счет переноса логики управления классами в одно место – в метакласс.  
Чтобы проиллюстрировать эти утверждения, предположим, что нам требует-  
ся автоматически добавить метод во множество классов. Конечно, мы могли   
бы сделать то же самое за счет наследования, если на этапе программирования   
классов известно, какой метод следует добавить. В этом случае мы можем про-  
сто реализовать метод в суперклассе и унаследовать его во всех классах:  
class Extras:  
 def extra(self, args): # Обычное наследование: слишком статично  
 ...  
   
class Client1(Extras): ... # Клиенты наследуют дополнительные методы  
class Client2(Extras): ...  
class Client3(Extras): ...  
   
X = Client1() # Создать экземпляр  
X.extra() # Вызвать дополнительный метод  
Однако иногда бывает невозможно заранее предсказать подобное расширение   
на этапе программирования классов. Представьте себе ситуацию, когда классы   
должны расширяться в ответ на действия пользователя во время выполнения   
программы или когда спецификации определяются в файлах с настройками.   
Конечно, мы могли бы реализовать все классы в воображаемом множестве,   
вручную проверяя необходимость добавления подобных методов, но это может   
существенно усложнить клиентские классы (в данном случае функция re-  
quired – это некоторая функция, выполняющая какие-то проверки):  
def extra(self, arg): ...  
   
class Client1: ... # Расширение клиентов: слишком разбросанно  
 if required():  
 Client1.extra = extra  
   
class Client2: ...  
 if required():  
 Client2.extra = extra  
   
class Client3: ...  
 if required():  
 Client3.extra = extra

Нужны или не нужны метаклассы   
1165  
X = Client1()  
X.extra()  
Мы можем также добавить методы в классы за пределами инструкций class,   
как показано ниже, благодаря тому, что методы классов – это обычные функ-  
ции, которые ассоциированы с классами и принимают экземпляры классов   
в первом аргументе self. Это вполне работоспособное решение, но оно переносит   
все бремя расширения на сами клиентские классы (и вообще-то предполагает,   
что мы не забудем сделать это!).  
С точки зрения простоты сопровождения, было бы гораздо лучше изолировать   
логику выбора в одном месте. Мы могли бы инкапсулировать часть этой рабо-  
ты, передавая классы управляющей функции – эта функция дополнит класс   
необходимыми расширениями и выполнит все необходимые проверки настро-  
ек во время выполнения:  
def extra(self, arg): ...  
   
def extras(Class): # Управляющая функция: слишком много ручной работы  
 if required():  
 Class.extra = extra  
   
class Client1: ...  
extras(Client1)  
   
class Client2: ...  
extras(Client2)  
   
class Client3: ...  
extras(Client3)  
   
X = Client1()  
X.extra()  
В этом примере классы передаются управляющей функции сразу после их   
создания. Подобное применение управляющих функций позволяет решить   
поставленную задачу, тем не менее, этот способ по-прежнему ложится тяже-  
лым грузом на плечи программистов, которые должны понять предъявляемые   
требования и придерживаться их. Было бы лучше иметь более простой способ   
принудительного расширения клиентских классов, используя который про-  
граммисты не забывали бы добавлять расширения и который не требовал бы   
выполнения операций с клиентскими классами. Другими словами, нам хоте-  
лось бы иметь возможность добавлять некоторый программный код, который   
автоматически вызывался бы в конце инструкции class для расширения   
класса.  
Это именно то, что предлагают метаклассы – объявляя метакласс, мы сообща-  
ем интерпретатору, что он должен передать создание объекта класса другому   
классу, указанному нами:  
def extra(self, arg): ...  
   
class Extras(type):  
 def \_\_init\_\_(Class, classname, superclasses, attributedict):  
 if required():  
 Class.extra = extra  
   
class Client1(metaclass=Extras): ... # Метакласс достаточно просто объявить

1166   
Глава 39. Метаклассы   
class Client2(metaclass=Extras): ... # Клиентский класс – экземпляр метакласса  
class Client3(metaclass=Extras): ...  
   
X = Client1() # X – экземпляр класса Client1  
X.extra()  
Поскольку интерпретатор автоматически вызывает метакласс в конце инструк-  
ции class, сразу после создания нового класса, он получает возможность рас-  
ширить, зарегистрировать или выполнить другие необходимые операции над   
классом. Кроме того, к клиентским классам предъявляется единственное тре-  
бование – они должны объявить метакласс. Каждый класс, в котором имеется   
такое объявление, автоматически получает все расширения, предусмотренные   
метаклассом, – и теперь, и в будущем, если метакласс изменится. Хотя это и не   
так заметно в таком маленьком примере, тем не менее, метаклассы лучше под-  
ходят для решения подобных задач, чем другие инструменты.  
Метаклассы против декораторов классов: раунд 1  
Интересно также отметить, что область применения декораторов классов, об-  
суждавшихся в предыдущей главе, иногда пересекается с областью примене-  
ния метаклассов. Несмотря на то, что обычно они используются для управле-  
ния экземплярами и расширения их возможностей, тем не менее, декораторы   
классов могут также использоваться для расширения самих классов независи-  
мо от создаваемых экземпляров.  
Например, предположим, что мы реализовали управляющую функцию так,   
что она возвращает расширенную версию класса, вместо того, чтобы просто   
модифицировать его экземпляры. Это обеспечило бы более высокую степень   
гибкости, так как в этом случае управляющая функция может вернуть объект   
любого типа, реализующий ожидаемый интерфейс класса:  
def extra(self, arg): ...  
   
def extras(Class):  
 if required():  
 Class.extra = extra  
 return Class  
   
class Client1: ...  
Client1 = extras(Client1)  
   
class Client2: ...  
Client2 = extras(Client2)  
   
class Client3: ...  
Client3 = extras(Client3)  
   
X = Client1()  
X.extra()  
Если вам показалось, что это напоминает декораторы классов, то вы не оши-  
блись. В предыдущей главе декораторы классов были представлены как ин-  
струмент расширения операции создания экземпляров. Благодаря тому, что   
принцип их действия основан на автоматическом присваивании результата   
функции оригинальному имени класса, нет никаких причин считать, что их

Нужны или не нужны метаклассы   
1167  
нельзя использовать для расширения классов еще до того, как будут созданы   
их экземпляры. То есть декораторы классов могут выполнять дополнительные   
операции не только при создании экземпляров, но и при создании самих клас-  
сов:  
def extra(self, arg): ...  
   
def extras(Class):  
 if required():  
 Class.extra = extra  
 return Class  
   
@extras  
class Client1: ... # Client1 = extras(Client1)  
   
@extras  
class Client2: ... # Присваивает объект имени класса независимо от экземпляров  
   
@extras  
class Client3: ...  
   
X = Client1() # Создать экземпляр расширенного класса  
X.extra() # X – экземпляр оригинального класса Client1  
Применение декораторов позволило автоматизировать предыдущий пример,   
в котором присваивание измененной версии класса его оригинальному име-  
ни происходило вручную. Как и в случае с метаклассами, здесь экземпляры   
создаются с помощью оригинального класса, а не объекта-обертки, потому что   
декоратор возвращает оригинальный класс. Фактически операция создания   
экземпляра в этом примере вообще не перехватывается.  
В данном конкретном случае – добавления методов в класс в процессе его соз-  
дания – выбор между метаклассами и декораторами достаточно произволен.   
Декораторы могут использоваться и для управления экземплярами, и для   
управления классами, причем вторая область их применения пересекается   
с областью применения метаклассов.  
Однако в действительности декораторы могут выполнять только одну функ-  
цию метаклассов. Как мы увидим далее, в этой своей роли декораторы явля-  
ются аналогом метода \_\_init\_\_ метаклассов, но у метаклассов имеются другие,   
дополнительные возможности настройки клиентских классов. Мы также узна-  
ем, что помимо инициализации класса метаклассы способны решать самые   
разные задачи, связанные с конструированием классов, реализовать которые   
с помощью декораторов может оказаться очень сложно.  
Кроме того, декораторы могут использоваться для управления и экземпляра-  
ми, и классами, но то же самое нельзя сказать о метаклассах – метаклассы   
предназначены для управления классами и применять их для управления эк-  
земплярами совсем непросто. Мы исследуем это отличие между декораторами   
и метаклассами ниже в этой главе.  
Примеры программного кода в этом разделе были в значительной степени аб-  
страктными, но мы скоро перейдем к более конкретным действующим при-  
мерам. Однако чтобы полностью понять, как действуют метаклассы, сначала   
необходимо получить более четкое представление об их модели.

1168   
Глава 39. Метаклассы   
Модель метаклассов  
Чтобы понять, как действуют метаклассы, вам необходимо поближе познако-  
миться с моделью типов в языке Python и с тем, что происходит в конце выпол-  
Python и с тем, что происходит в конце выпол-  
 и с тем, что происходит в конце выпол-  
нения инструкции class.  
Классы – экземпляры класса type  
До сих пор в этой книге мы решали свои задачи, создавая экземпляры встроен-  
ных типов, таких как списки и строки, а также экземпляры наших собствен-  
ных классов. Как мы уже знаем, экземпляры классов обладают некоторыми   
атрибутами с собственными данными, а кроме того, они наследуют атрибуты   
поведения от классов, на основе которых они были созданы. То же самое верно   
и для встроенных типов – экземпляры списков, например, обладают собствен-  
ными значениями и наследуют методы от типа list.  
Мы многого можем добиться с помощью таких объектов экземпляров, однако   
модель типов в языке Python оказывается немного богаче, чем я описал. В дей-  
Python оказывается немного богаче, чем я описал. В дей-  
 оказывается немного богаче, чем я описал. В дей-  
ствительности, в той модели, которую мы видели до сих пор, есть белые пятна:   
экземпляры создаются на основе классов, но на основе чего создаются сами   
классы? Оказывается, что классы также являются экземплярами некоторого   
класса:  
 •  
В Python 3.0 объекты пользовательских классов являются экземплярами   
объекта с именем type, который сам является классом.  
 •  
В Python 2.6 классы нового стиля наследуют класс object, который является   
подклассом класса type. Классические классы являются экземплярами type   
и не создаются из класса.  
Понятие типов мы исследовали в главе 9, а отношения между типами и класса-  
ми – в главе 31, тем не менее, мы еще раз пройдемся по основам, чтобы увидеть,   
как они применяются к метаклассам.  
Напомню, что встроенная функция type возвращает тип любого объекта (кото-  
рый сам по себе является объектом). Для встроенных типов, таких как списки,   
типом экземпляра является встроенный тип list, а типом типа list является   
сам тип type. Объект type, находящийся на вершине иерархии, создает более   
специализированные типы, а эти специализированные типы создают экзем-  
пляры. Вы можете наблюдать это в интерактивной оболочке. Например, в Py-  
Py-  
thon 3.0:  
C:\misc> c:\python30\python  
>>> type([]) # В 3.0 список – это экземпляр типа list  
<class ‘list’>  
>>> type(type([])) # Тип list – экземпляр класса type  
<class ‘type’>  
   
>>> type(list) # То же самое, но с использованием имен типов  
<class ‘type’>  
>>> type(type) # Тип объекта type – type: вершина иерархии  
<class ‘type’>  
Как мы узнали, когда в главе 31 изучали изменения в классах нового стиля,   
то же в значительной степени справедливо и для Python 2.6 (и более ранних   
версий), только типы в этой версии не являются классами – type является уни-

Модель метаклассов   
1169  
кальным встроенным объектом, который находится на вершине иерархии ти-  
пов и используется для конструирования других типов:  
C:\misc> c:\python26\python  
>>> type([]) # В 2.6 type – это немного иной объект  
<type ‘list’>  
>>> type(type([]))  
<type ‘type’>  
   
>>> type(list)  
<type ‘type’>  
>>> type(type)  
<type ‘type’>  
Оказывается, что отношение вида тип/экземпляр сохраняется и для классов:   
экземпляры создаются из классов, а классы создаются из объекта type. Однако   
в Python 3.0 понятие «тип» было объединено с понятием «класс». Фактически   
эти два понятия стали синонимами – классы  являются  типами,  а  типы  –   
классами. То есть:  
 •  
Типы определяются классами, а классы являются производными от type.  
 •  
Пользовательские классы являются экземплярами классов типов.  
 •  
Пользовательские классы являются типами, которые генерируют собствен-  
ные экземпляры.  
Как мы видели, эти положения подтверждаются программным кодом выше,   
который проверяет типы экземпляров: тип экземпляра – это класс, из кото-  
рого он был получен. Это же справедливо и для классов, что является ключом   
к теме этой главы. Поскольку классы обычно создаются из корневого класса   
type, большинству программистов не нужно задумываться об эквивалентности   
понятий тип/класс. Однако это открывает новые возможности для расшире-  
ния не только классов, но и их экземпляров.  
Например, классы в Python 3.0 (и классы нового стиля в Python 2.6) являются   
экземплярами класса type, а объекты экземпляров являются экземплярами   
своих классов. Фактически классы теперь имеют атрибут \_\_class\_\_, который   
ссылается на класс type, так же, как экземпляры имеют атрибут \_\_class\_\_, ссы-  
лающийся на классы, из которых они были созданы:  
C:\misc> c:\python30\python  
>>> class C: pass # Объект класса в 3.0 (класс нового стиля)  
...  
>>> X = C() # Объект экземпляра класса  
   
>>> type(X) # Экземпляр – это экземпляр класса  
<class ‘\_\_main\_\_.C’>  
>>> X.\_\_class\_\_ # Класс экземпляра  
<class ‘\_\_main\_\_.C’>  
   
>>> type(C) # Класс – экземпляр класса type  
<class ‘type’>  
>>> C.\_\_class\_\_ # Класс класса – type  
<class ‘type’>  
Обратите особое внимание на две последние строки в примере – классы явля-  
ются экземплярами класса type, так же, как обычные экземпляры являются

1170   
Глава 39. Метаклассы   
экземплярами классов. В версии 3.0 это одинаково справедливо как для встро-  
енных типов, так и для пользовательских классов. Фактически классы не яв-  
ляются каким-то отдельным понятием: они просто являются пользовательски-  
ми типами, а сам тип определяется классом.  
В Python 2.6 дело обстоит похожим образом в случае классов нового стиля, на-  
Python 2.6 дело обстоит похожим образом в случае классов нового стиля, на-  
 2.6 дело обстоит похожим образом в случае классов нового стиля, на-  
следующих класс object, потому что они обладают поведением классов в Py-  
Py-  
thon 3.0:  
C:\misc> c:\python26\python  
>>> class C(object): pass # Классы нового стиля в 2.6, также имеют класс  
...   
>>> X = C()  
   
>>> type(X)  
<class ‘\_\_main\_\_.C’>  
>>> type(C)  
<type ‘type’>  
   
>>> X.\_\_class\_\_  
<class ‘\_\_main\_\_.C’>  
>>> C.\_\_class\_\_  
<type ‘type’>  
Однако классические классы в версии 2.6 немного отличаются – они являются   
отражением модели классов в предыдущих версиях Python, поэтому у них от-  
Python, поэтому у них от-  
, поэтому у них от-  
сутствует атрибут \_\_class\_\_ и, подобно встроенным типам в 2.6, они являются   
экземплярами type, а не класса типа:  
C:\misc> c:\python26\python  
>>> class C: pass # Классические классы в 2.6 не имеют класса  
...   
>>> X = C()  
   
>>> type(X)  
<type ‘instance’>  
>>> type(C)  
<type ‘classobj’>  
   
>>> X.\_\_class\_\_  
<class \_\_main\_\_.C at 0x005F85A0>  
>>> C.\_\_class\_\_  
AttributeError: class C has no attribute ‘\_\_class\_\_’  
Метаклассы – подклассы класса type  
Итак, где нам может пригодиться знание, что в Python 3.0 классы являются   
экземплярами класса type? Оказывается, это обстоятельство позволяет нам   
создавать метаклассы. Поскольку теперь понятие типа совпадает с понятием   
класса, мы можем создать подкласс класса type с использованием обычных   
приемов объектно-ориентированного программирования и синтаксиса опреде-  
ления классов, чтобы адаптировать его. А так как классы в действительности   
являются экземплярами класса type, создание классов из адаптированных   
подклассов класса type позволит нам реализовать собственные типы классов.   
Естественно, все это в полной мере относится к любым классам в версии 3.0 и к   
классам нового стиля в Python 2.6:

Модель метаклассов   
1171  
 •  
type – это класс, из которого создаются пользовательские классы.  
 •  
Метаклассы – это подклассы класса type.  
 •  
Объекты классов – это экземпляры класса type или его подклассов.  
 •  
Объекты экземпляров создаются из классов.  
Другими словами, чтобы управлять созданием классов и расширять их воз-  
можности, нам достаточно указать, что пользовательский класс создается из   
пользовательского метакласса, а не из обычного класса type.  
Следует заметить, что отношение между экземпляром класса и типом – не то   
же, что наследование: пользовательские классы могут также иметь суперклас-  
сы, от которых они и их экземпляры наследуют атрибуты (наследуемые супер-  
классы перечисляются в круглых скобках в инструкции class и сохраняются   
в атрибуте \_\_bases\_\_ класса в виде кортежа). Отношение с типом, из которого   
создается класс и экземпляром которого он является, – это совсем другой вид   
отношений. Следующий раздел описывает процедуру, которой следует интер-  
претатор, реализуя отношение «экземпляр»–«некоторого типа».  
Протокол инструкции class  
Создание производных классов от класса type с целью расширения его возмож-  
ностей – это на самом деле лишь половина магии, происходящей за кулисами   
метаклассов. Нам еще необходим способ передать управление процедурой соз-  
дания класса метаклассу, вместо класса type. Чтобы полностью понять, как это   
делается, нам также необходимо знать, как действует инструкция class.  
Мы уже знаем, что когда интерпретатор встречает инструкцию class, он вы-  
полняет вложенный блок инструкции, чтобы создать атрибуты класса, – все   
операции присваивания, выполняющиеся на верхнем уровне внутри определе-  
ния класса, создают атрибуты объекта класса. Обычно это функции методов,   
создаваемые вложенными инструкциями def, но точно так же это могут быть   
любые другие атрибуты, которые будут служить атрибутами данных, общими   
для всех экземпляров.  
Говоря техническим языком, интерпретатор следует стандартному протоколу:   
в конце инструкции class после выполнения всех вложенных инструкций и со-  
хранения всех созданных имен в словаре пространства имен он вызывает объ-  
ект type, чтобы создать объект класса:  
class = type(classname, superclasses, attributedict)  
Объект type определяет метод \_\_call\_\_ перегрузки операторов, который вызы-  
вает два других метода, когда вызывается объект type:  
type.\_\_new\_\_(typeclass, classname, superclasses, attributedict)  
type.\_\_init\_\_(class, classname, superclasses, attributedict)  
Метод \_\_new\_\_ создает и возвращает новый объект класса, а затем этот вновь   
созданный объект инициализируется методом \_\_init\_\_. Как мы увидим чуть   
ниже, именно эти два метода обычно переопределяются в метаклассах, адап-  
тирующих класс type, для расширения классов.  
Например, для следующего определения класса:  
class Spam(Eggs): # Наследует класс Eggs  
 data = 1 # Атрибут данных класса

1172   
Глава 39. Метаклассы   
 def meth(self, arg): # Атрибут метода класса  
 pass  
интерпретатор выполнит вложенный блок программного кода, создаст два   
атрибута класса (data и meth) и затем вызовет объект type, чтобы создать объект   
класса:  
Spam = type(‘Spam’,(Eggs,), {‘data’: 1, ‘meth’:meth, ‘\_\_module\_\_’:’\_\_main\_\_’})  
Этот вызов выполняется в конце инструкции class, поэтому он является иде-  
альным местом, где можно было бы внести дополнения или как-то иначе обра-  
ботать класс. Вся хитрость состоит в том, чтобы заменить type его подклассом,   
который перехватит этот вызов. Как это сделать, рассказывается в следующем   
разделе.  
Объявление метаклассов  
Как мы только что видели, по умолчанию классы создаются из класса type.   
Чтобы заставить интерпретатор вместо него использовать наш метакласс, до-  
статочно просто добавить объявление метакласса, после чего он будет исполь-  
зоваться для создания класса. Порядок объявления метакласса зависит от ис-  
пользуемой версии Python. В Python 3.0 нужно указать желаемый метакласс   
в виде именованного аргумента в заголовке инструкции class:  
class Spam(metaclass=Meta): # В версии 3.0 и выше  
В заголовке перед объявлением метакласса также могут быть перечислены на-  
следуемые суперклассы. Так, в следующем примере новый класс Spam наследу-  
ет класс Eggs, но при этом является экземпляром и создается метаклассом Meta:  
class Spam(Eggs, metaclass=Meta): # Допускается указывать суперклассы  
Для достижения того же эффекта в Python 2.6 метакласс должен объявляться   
иначе – в виде атрибута класса, а не именованного аргумента. Наследование   
суперкласса object является обязательным условием, чтобы сделать этот класс   
классом нового стиля. Кроме того, такой способ объявления не действует в вер-  
сии 3.0, так как этот атрибут просто игнорируется:  
class spam(object): # Только в версии 2.6  
 \_\_metaclass\_\_ = Meta  
Кроме того, в версии 2.6 можно использовать глобальную переменную модуля   
\_\_metaclass\_\_, чтобы связать все классы в модуле с желаемым метаклассом. Эта   
переменная не поддерживается в Python 3.0, так как это была лишь временная   
мера, которая упрощала создание классов нового стиля, без объявления супер-  
класса object в каждом определении класса.  
При наличии объявления метакласса операция создания объекта класса, вы-  
полняемая в конце инструкции class, изменяется так, что вместо класса type   
она вызывает метакласс:  
class = Meta(classname, superclasses, attributedict)  
А так как метакласс является подклассом type, он наследует метод \_\_call\_\_ клас-  
са type, который делегирует вызовы методов создания и инициализации нового   
объекта класса метаклассу, если он определяет свои версии этих методов:

Программирование метаклассов   
1173  
Meta.\_\_new\_\_(Meta, classname, superclasses, attributedict)  
Meta.\_\_init\_\_(class, classname, superclasses, attributedict)  
Для примера ниже еще раз приводится фрагмент из предыдущего раздела, до-  
полненный спецификацией метакласса в формате версии 3.0:  
class Spam(Eggs, metaclass=Meta): # Наследует Eggs, экземпляр Meta  
 data = 1 # Атрибут данных класса  
 def meth(self, arg): # Атрибут метода класса  
 pass  
В конце этой инструкции class интерпретатор выполнит следующий вызов,   
чтобы создать объект класса:  
Spam = Meta(‘Spam’, (Eggs,), {‘data’:1, ‘meth’:meth, ‘\_\_module\_\_’:’\_\_main\_\_’})  
Если в метаклассе определены собственные версии методов \_\_new\_\_ или \_\_  
init\_\_, эти методы будут вызваны унаследованным от type методом \_\_call\_\_   
для создания и инициализации нового класса. В следующем разделе рассказы-  
вается, как создается этот последний фрагмент мозаики метаклассов.  
Программирование метаклассов  
Теперь мы знаем, как интерпретатор переадресует процедуру создания класса   
метаклассу, если таковой указан. Но как определить сам метакласс, расши-  
ряющий класс type?  
Оказывается, вы уже знаете большую часть ответа на этот вопрос – метаклассы   
определяются с помощью обычных инструкций class. Единственное их важное   
отличие от обычных классов состоит в том, что метаклассы автоматически вы-  
зываются интерпретатором в конце инструкции class, и они должны придер-  
живаться интерфейса, ожидаемого суперклассом type.  
Основы метаклассов  
Самый простой, пожалуй, метакласс, который только можно запрограммиро-  
вать, – это обычный подкласс класса type с методом \_\_new\_\_, который создает   
объект класса вызовом метода суперкласса type. Метод \_\_new\_\_ метакласса, как   
показано в следующем примере, вызывается методом \_\_call\_\_, унаследован-  
ным от type, – обычно в нем выполняются необходимые дополнительные опе-  
рации по настройке и затем вызывается метод \_\_new\_\_ суперкласса type, чтобы   
создать и вернуть новый объект класса:  
class Meta(type):  
 def \_\_new\_\_(meta, classname, supers, classdict):  
 # Вызывается унаследованным методом type.\_\_call\_\_  
 return type.\_\_new\_\_(meta, classname, supers, classdict)  
В действительности этот метакласс ничего не делает (с тем же успехом мы   
могли бы позволить создать класс с помощью класса по умолчанию type), но   
он демонстрирует способ, каким можно задействовать метакласс в процедуре   
создания класса для его расширения. Так как метакласс вызывается в конце   
инструкции class и благодаря тому, что метод \_\_call\_\_ объекта type вызыва-  
ет методы \_\_new\_\_ и \_\_init\_\_, реализации этих методов могут управлять всеми   
классами, создаваемыми с помощью метакласса.

1174   
Глава 39. Метаклассы   
Ниже снова приводится наш пример, где в метакласс были добавлены инструк-  
ции вывода и программный код, позволяющий проследить, как выполняется   
процедура создания класса:  
class MetaOne(type):  
 def \_\_new\_\_(meta, classname, supers, classdict):  
 print(‘In MetaOne.new:’, classname, supers, classdict, sep=’\n...’)  
 return type.\_\_new\_\_(meta, classname, supers, classdict)  
   
class Eggs:  
 pass  
   
print(‘making class’)  
class Spam(Eggs, metaclass=MetaOne): # Наследует Eggs, экземпляр Meta  
 data = 1 # Атрибут данных класса  
 def meth(self, arg): # Атрибут метода класса  
 pass  
   
print(‘making instance’)  
X = Spam()  
print(‘data:’, X.data)  
Здесь класс Spam наследует класс Eggs и является экземпляром класса Meta-  
One, но объект X является экземпляром класса Spam. Запустите этот пример под   
управлением Python 3.0 и обратите внимание, что метакласс вызывается в кон-  
це инструкции class, еще до того, как будет создан экземпляр класса, – мета-  
классы служат для создания классов, а классы – для создания экземпляров:  
making class  
In MetaOne.new:  
...Spam  
...(<class ‘\_\_main\_\_.Eggs’>,)  
...{‘\_\_module\_\_’: ‘\_\_main\_\_’, ‘data’: 1, ‘meth’: <function meth at 0x02AEBA08>}  
making instance  
data: 1  
Расширение операций конструирования   
и инициализации  
Метаклассы могут также переопределять метод \_\_init\_\_, вызываемый мето-  
дом \_\_call\_\_ объекта type: вообще говоря, метод \_\_new\_\_ создает и возвращает   
объект класса, а метод \_\_init\_\_ инициализирует уже созданный класс. Мета-  
классы могут переопределять оба метода и с их помощью управлять процессом   
создания класса:  
class MetaOne(type):  
 def \_\_new\_\_(meta, classname, supers, classdict):  
 print(‘In MetaOne.new: ‘, classname, supers, classdict, sep=’\n...’)  
 return type.\_\_new\_\_(meta, classname, supers, classdict)  
   
 def \_\_init\_\_(Class, classname, supers, classdict):  
 print(‘In MetaOne init:’, classname, supers, classdict, sep=’\n...’)  
 print(‘...init class object:’, list(Class.\_\_dict\_\_.keys()))  
   
class Eggs:  
 pass

Программирование метаклассов   
1175  
print(‘making class’)  
class Spam(Eggs, metaclass=MetaOne): # Наследует Eggs, экземпляр Meta  
 data = 1 # Атрибут данных класса  
 def meth(self, arg): # Атрибут метода класса  
 pass  
   
print(‘making instance’)  
X = Spam()  
print(‘data:’, X.data)  
В этом случае метод инициализации класса вызывается после вызова метода,   
конструирующего класс, но оба они вызываются в конце инструкции class,   
перед тем, как будет создан хотя бы один экземпляр:  
making class  
In MetaOne.new:  
...Spam  
...(<class ‘\_\_main\_\_.Eggs’>,)  
...{‘\_\_module\_\_’: ‘\_\_main\_\_’, ‘data’: 1, ‘meth’: <function meth at 0x02AAB810>}  
In MetaOne init:  
...Spam  
...(<class ‘\_\_main\_\_.Eggs’>,)  
...{‘\_\_module\_\_’: ‘\_\_main\_\_’, ‘data’: 1, ‘meth’: <function meth at 0x02AAB810>}  
...init class object: [‘\_\_module\_\_’, ‘data’, ‘meth’, ‘\_\_doc\_\_’]  
making instance  
data: 1  
Другие приемы программирования метаклассов  
Переопределение методов \_\_new\_\_ и \_\_init\_\_ суперкласса type – это наиболее ти-  
пичный способ добавить логику в процедуру создания объекта класса, но при   
этом существуют и другие способы.  
Использование простых фабричных функций  
Вообще говоря, метакласс в действительности не обязательно должен быть   
классом. Как мы уже знаем, чтобы создать класс, инструкция class произво-  
дит простой вызов в конце. Вследствие этого в качестве метакласса может ис-  
пользоваться любой вызываемый объект, который принимает передаваемые   
ему аргументы и возвращает объект, совместимый с ожидаемым классом. Фак-  
тически роль метакласса может играть объект простой фабричной функции:  
# Простая функция также может играть роль метакласса  
   
def MetaFunc(classname, supers, classdict):  
 print(‘In MetaFunc: ‘, classname, supers, classdict, sep=’\n...’)  
 return type(classname, supers, classdict)  
   
class Eggs:  
 pass  
   
print(‘making class’)  
class Spam(Eggs, metaclass=MetaFunc): # В конце вызовет простую функцию  
 data = 1 # Функция возвращает класс  
 def meth(self, args):  
 pass

1176   
Глава 39. Метаклассы   
print(‘making instance’)  
X = Spam()  
print(‘data:’, X.data)  
Если запустить этот пример, в конце инструкции class будет вызвана функ-  
ция, которая вернет новый объект ожидаемого класса. Функция перехватыва-  
ет вызов, который обычно предназначается методу \_\_call\_\_ объекта type:  
making class  
In MetaFunc:  
...Spam  
...(<class ‘\_\_main\_\_.Eggs’>,)  
...{‘\_\_module\_\_’: ‘\_\_main\_\_’, ‘data’: 1, ‘meth’: <function meth at 0x02B8B6A8>}  
making instance  
data: 1  
Перегрузка метода вызова процедуры   
создания класса в метаклассе  
При работе с метаклассами интерпретатор использует обычные механизмы   
ООП, поэтому метаклассы могут также перехватывать сам вызов процедуры   
создания, который выполняется в конце инструкции class, переопределив   
метод \_\_call\_\_. Однако при этом реализация метакласса становится немного   
сложнее:  
# Метод \_\_call\_\_ можно переопределить,   
# а метаклассы могут иметь свои метаклассы  
   
class SuperMeta(type):  
 def \_\_call\_\_(meta, classname, supers, classdict):  
 print(‘In SuperMeta.call: ‘, classname, supers, classdict,   
 sep=’\n...’)  
 return type.\_\_call\_\_(meta, classname, supers, classdict)  
   
class SubMeta(type, metaclass=SuperMeta):  
 def \_\_new\_\_(meta, classname, supers, classdict):  
 print(‘In SubMeta.new: ‘, classname, supers, classdict, sep=’\n...’)  
 return type.\_\_new\_\_(meta, classname, supers, classdict)  
   
 def \_\_init\_\_(Class, classname, supers, classdict):  
 print(‘In SubMeta init:’, classname, supers, classdict, sep=’\n...’)  
 print(‘...init class object:’, list(Class.\_\_dict\_\_.keys()))  
   
class Eggs:  
 pass  
   
print(‘making class’)  
class Spam(Eggs, metaclass=SubMeta):  
 data = 1  
 def meth(self, arg):  
 pass  
   
print(‘making instance’)  
X = Spam()  
print(‘data:’, X.data)  
Если запустить этот пример, интерпретатор по очереди вызовет все три пере-  
определенных метода. По сути – это то, что делает объект type:

Программирование метаклассов   
1177  
making class  
In SuperMeta.call:  
...Spam  
...(<class ‘\_\_main\_\_.Eggs’>,)  
...{‘\_\_module\_\_’: ‘\_\_main\_\_’, ‘data’: 1, ‘meth’: <function meth at 0x02B7BA98>}  
In SubMeta.new:  
...Spam  
...(<class ‘\_\_main\_\_.Eggs’>,)  
...{‘\_\_module\_\_’: ‘\_\_main\_\_’, ‘data’: 1, ‘meth’: <function meth at 0x02B7BA98>}  
In SubMeta init:  
...Spam  
...(<class ‘\_\_main\_\_.Eggs’>,)  
...{‘\_\_module\_\_’: ‘\_\_main\_\_’, ‘data’: 1, ‘meth’: <function meth at 0x02B7BA98>}  
...init class object: [‘\_\_module\_\_’, ‘data’, ‘meth’, ‘\_\_doc\_\_’]  
making instance  
data: 1  
Перегрузка метода вызова процедуры создания класса   
в обычных классах  
Предыдущий пример осложняется тем фактом, что для создания объектов   
классов используются метаклассы, которые не создают собственные экзем-  
пляры. Вследствие этого правила поиска имен для метаклассов несколько от-  
личаются от тех, к которым мы привыкли. Поиск метода \_\_call\_\_, например,   
выполняется в классе объекта. Для метаклассов это означает – в метаклассе   
метакласса.  
Чтобы задействовать обычный механизм поиска имен, начиная с экземпляра,   
мы можем использовать обычные классы и экземпляры. Следующий пример   
выведет те же результаты, что и предыдущая версия, но обратите внимание,   
что здесь методы \_\_new\_\_ и \_\_init\_\_ должны иметь другие имена, в противном   
случае они будут вызываться при создании экземпляра SubMeta, а не когда позд-  
нее он будет вызываться как метакласс:  
class SuperMeta:  
 def \_\_call\_\_(self, classname, supers, classdict):  
 print(‘In SuperMeta.call: ‘, classname, supers, classdict,   
 sep=’\n...’)  
 Class = self.\_\_New\_\_(classname, supers, classdict)  
 self.\_\_Init\_\_(Class, classname, supers, classdict)  
 return Class  
   
class SubMeta(SuperMeta):  
 def \_\_New\_\_(self, classname, supers, classdict):  
 print(‘In SubMeta.new: ‘, classname, supers, classdict, sep=’\n...’)  
 return type(classname, supers, classdict)  
   
 def \_\_Init\_\_(self, Class, classname, supers, classdict):  
 print(‘In SubMeta init:’, classname, supers, classdict, sep=’\n...’)  
 print(‘...init class object:’, list(Class.\_\_dict\_\_.keys()))  
   
class Eggs:  
 pass  
   
print(‘making class’)  
class Spam(Eggs, metaclass=SubMeta()): # Метакласс – экземпляр обычного класса  
 data = 1 # Вызывается в конце инструкции

1178   
Глава 39. Метаклассы   
 def meth(self, arg):  
 pass  
   
print(‘making instance’)  
X = Spam()  
print(‘data:’, X.data)  
Хотя эти альтернативные формы вполне работоспособны, однако в большин-  
 эти альтернативные формы вполне работоспособны, однако в большин-  
эти альтернативные формы вполне работоспособны, однако в большин-  
 альтернативные формы вполне работоспособны, однако в большин-  
альтернативные формы вполне работоспособны, однако в большин-  
 формы вполне работоспособны, однако в большин-  
формы вполне работоспособны, однако в большин-  
 вполне работоспособны, однако в большин-  
вполне работоспособны, однако в большин-  
 работоспособны, однако в большин-  
работоспособны, однако в большин-  
, однако в большин-  
однако в большин-  
 в большин-  
в большин-  
 большин-  
большин-  
стве случаев метаклассы выполняют свою работу, переопределяя методы \_\_  
new\_\_ и \_\_init\_\_ суперкласса type. На практике этого вполне достаточно, и такой   
подход гораздо проще других способов. Тем не менее ниже мы увидим, что про-  
стой метакласс в виде функции часто может действовать как декоратор класса,   
что позволяет метаклассам управлять не только классами, но и экземплярами.  
Экземпляры и наследование  
Из-за того, что объявления метаклассов напоминают объявления наследуемых   
суперклассов, при первом знакомстве это может вводить в заблуждение. Сле-  
дующие ключевые моменты помогут вам обобщить и прояснить модель мета-  
классов:  
 •  
Метаклассы наследуют класс type. Хотя они и играют свою особую роль,   
но определяются метаклассы с помощью инструкции class и поддержива-  
ют обычную модель ООП в языке Python. Например, будучи подклассами   
класса type, они могут переопределять методы объекта type, переопределяя   
и адаптируя их по мере необходимости. Обычно метаклассы переопреде-  
ляют методы \_\_new\_\_ и \_\_init\_\_ класса type, настраивая процедуру созда-  
ния и инициализации, но они также могут переопределять метод \_\_call\_\_,   
когда требуется перехватить вызов процедуры создания класса в конце ин-  
струкции class. Хотя это и необычно, но они также могут быть простыми   
функциями, которые возвращают произвольные объекты вместо подклас-  
сов класса type.  
 •  
Объявления метаклассов наследуются подклассами. Объявление meta-  
class=M в пользовательском классе наследуется подклассами этого класса,   
поэтому метакласс будет вызываться при конструировании любых классов,   
наследующих это объявление в цепочке суперклассов.  
 •  
Атрибуты метакласса не наследуются экземплярами классов. Объявле-  
ния метаклассов определяют отношения экземпляров, которые отличают-  
ся от отношений наследования. Так как классы являются экземплярами   
метаклассов, поведение, определяемое метаклассом, применяется к классу,   
но не к экземплярам этого класса. Экземпляры приобретают поведение от   
своих классов и суперклассов, но не от метаклассов. С технической точки   
зрения, поиск атрибутов экземпляров обычно выполняется только в слова-  
рях \_\_dict\_\_ экземпляров и во всех классах, наследуемых экземпляром, –   
метаклассы не включаются в цепочку поиска по дереву наследования.  
Чтобы продемонстрировать последние два пункта, рассмотрим следующий   
пример:  
class MetaOne(type):  
 def \_\_new\_\_(meta, classname, supers, classdict): # Переопределяет метод   
 print(‘In MetaOne.new:’, classname) # класса type  
 return type.\_\_new\_\_(meta, classname, supers, classdict)

Пример: добавление методов в классы   
1179  
 def toast(self):  
 print(‘toast’)  
   
class Super(metaclass=MetaOne): # Объявление метакласса наслед-ся подклассами  
 def spam(self): # MetaOne вызывается дважды   
 print(‘spam’) # при создании двух классов  
   
class C(Super): # Суперкласс: наследование – не экземпляр  
 def eggs(self): # Классы наследуют атрибуты суперклассов  
 print(‘eggs’) # Но не наследуют атрибуты метаклассов  
   
X = C()  
X.eggs() # Наследует от класса C  
X.spam() # Наследует от класса Super  
X.toast() # Не наследует от метакласса  
Если запустить этот пример, метакласс примет участие в конструировании   
обоих клиентских классов, но экземпляры унаследуют только атрибуты своих   
классов и не унаследуют атрибуты метакласса:  
In MetaOne.new: Super  
In MetaOne.new: C  
eggs  
spam  
AttributeError: ‘C’ object has no attribute ‘toast’  
Хотя детали имеют определенное значение, тем не менее, при работе с мета-  
классами важнее держать в уме общую картину. Метаклассы, подобные тем,   
что мы видели в этом примере, будут выполняться автоматически для всех   
классов, в которых они объявляются. В отличие от вспомогательных функций,   
которые мы видели выше, такие классы будут автоматически приобретать лю-  
бые расширения, определяемые метаклассом. Кроме того, подобные расшире-  
ния достаточно запрограммировать в одном месте – в метаклассе, что упроща-  
ет внесение изменений впоследствии, по мере изменения наших требований.   
Подобно многим другим инструментам языка Python метаклассы упрощают   
сопровождение программного кода, устраняя его избыточность. Однако чтобы   
полностью осознать всю мощь метаклассов, нам необходимо рассмотреть не-  
сколько более крупных и практичных примеров их использования.  
Пример: добавление методов в классы  
В этом и в следующем разделе мы рассмотрим примеры двух наиболее типич-  
ных случаев использования метаклассов: добавление методов в класс и авто-  
матическое декорирование всех методов. Это лишь две из множества областей   
применения метаклассов, каждую из которых невозможно рассмотреть в рам-  
ках этой главы из-за ограниченного пространства книги, – вам снова придется   
самостоятельно поискать в Сети дополнительные примеры применений. Однако   
приводимые примеры являются достаточно показательными, и их будет вполне   
достаточно, чтобы продемонстрировать основы применения метаклассов.  
Кроме того, оба примера дают возможность сравнить декораторы классов и ме-  
таклассы – наш первый пример сравнивает реализации расширения класса   
и обертывания его экземпляров на основе метакласса и декоратора, а во втором   
примере сначала применяется декоратор с метаклассом, а затем применяется

1180   
Глава 39. Метаклассы   
второй декоратор. Как вы увидите, эти два инструмента часто оказываются   
взаимозаменяемыми и даже взаимодополняющими.  
Расширение вручную  
Ранее в этой главе мы рассматривали шаблон программного кода, расширяю-  
щего классы добавлением дополнительных методов различными способами.   
Как мы видели, если к моменту написания классов уже известно, какие ме-  
тоды потребуется добавить, достаточно будет воспользоваться механизмом   
наследования. Того же эффекта зачастую можно добиться за счет внедрения   
объекта, используя прием композиции. Однако иногда, в более динамичных   
случаях, бывает необходимо использовать другие приемы – вспомогательных   
функций обычно бывает вполне достаточно, но метаклассы обеспечивают более   
очевидное решение и снижают затраты на сопровождение, когда в будущем по-  
требуется внести изменения.  
Давайте воплотим эти идеи в действующий программный код. Рассмотрим   
следующий пример расширения класса вручную – он добавляет два метода   
к двум классам после их создания:  
# Расширение вручную – добавление новых методов в классы  
   
class Client1:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 def spam(self):  
 return self.value \* 2  
   
class Client2:  
 value = ‘ni?’  
   
def eggsfunc(obj):  
 return obj.value \* 4  
   
def hamfunc(obj, value):  
 return value + ‘ham’  
   
Client1.eggs = eggsfunc  
Client1.ham = hamfunc  
   
Client2.eggs = eggsfunc  
Client2.ham = hamfunc  
   
X = Client1(‘Ni!’)  
print(X.spam())  
print(X.eggs())  
print(X.ham(‘bacon’))  
   
Y = Client2()  
print(Y.eggs())  
print(Y.ham(‘bacon’))  
Этот прием действует, потому что методы всегда могут быть добавлены к клас-  
су после его создания при условии, что присваиваемые функции принимают   
ссылку на экземпляр в первом аргументе self. Этот аргумент может использо-  
ваться для доступа к информации, хранящейся в экземпляре класса, несмотря   
на то, что сама функция может быть определена независимо от класса.

Пример: добавление методов в классы   
1181  
Если запустить этот пример, мы получим вывод от метода внутри первого   
класса, а также от двух методов, добавленных к классам после их создания:  
Ni!Ni!  
Ni!Ni!Ni!Ni!  
baconham  
ni?ni?ni?ni?  
baconham  
Этот метод с успехом может использоваться в отдельных случаях и может при-  
меняться для добавления методов к произвольным классам во время выполне-  
ния программы. Однако он имеет один существенный недостаток: нам придет-  
ся многократно выполнять добавление методов ко всем классам, нуждающим-  
ся в этих методах. В данном примере добавить два метода к двум классам было   
несложно, но в более сложных ситуациях такой подход может потребовать   
больше времени и к тому же он чреват ошибками. Если мы забудем выполнить   
расширение какого-либо класса или потребуется выполнить расширение как-  
то иначе, мы рискуем столкнуться с проблемами.  
Расширение с помощью метакласса  
Прием, основанный на расширении вручную, вполне дееспособен, но в круп-  
ных программах было бы лучше иметь возможность автоматически применять   
расширения сразу ко всему множеству классов. Благодаря этому мы смогли   
бы избежать ошибок при расширении классов. Кроме того, когда реализация   
расширения располагается в одном месте, это упрощает возможность внесения   
изменений в будущем – все классы будут получать изменения автоматически.  
Один из способов достижения этой цели заключается в использовании мета-  
классов. Если мы запрограммируем расширения в метаклассе, все классы,   
объявляющие этот метакласс, будут расширены единообразно и корректно   
и автоматически будут получать любые изменения, которые могут быть добав-  
лены в будущем. Этот прием демонстрирует следующий пример:  
# Расширение с помощью метакласса – лучше поддерживает изменения в будущем  
def eggsfunc(obj):  
 return obj.value \* 4  
   
def hamfunc(obj, value):  
 return value + ‘ham’  
   
class Extender(type):  
 def \_\_new\_\_(meta, classname, supers, classdict):  
 classdict[‘eggs’] = eggsfunc  
 classdict[‘ham’] = hamfunc  
 return type.\_\_new\_\_(meta, classname, supers, classdict)  
   
class Client1(metaclass=Extender):  
 def \_\_init\_\_(self, value):  
 self.value = value  
 def spam(self):  
 return self.value \* 2  
   
class Client2(metaclass=Extender):  
 value = ‘ni?’

1182   
Глава 39. Метаклассы   
X = Client1(‘Ni!’)  
print(X.spam())  
print(X.eggs())  
print(X.ham(‘bacon’))  
   
Y = Client2()  
print(Y.eggs())  
print(Y.ham(‘bacon’))  
На этот раз оба клиентских класса расширяются новыми методами благодаря   
тому, что они являются экземплярами метакласса, который выполняет расши-  
рение. Если запустить эту версию примера, он выведет те же результаты, что   
и прежде, – мы не изменили реализацию дополнительных методов, а просто   
реструктурировали программный код, организовав его более очевидным и на-  
глядным способом:  
Ni!Ni!  
Ni!Ni!Ni!Ni!  
baconham  
ni?ni?ni?ni?  
baconham  
Обратите внимание, что метакласс в этом примере по-прежнему играет доста-  
точно статичную роль: добавляет два известных метода к каждому классу, объ-  
являющему метакласс. Фактически если нам всегда требуется одно и то же –   
добавлять одни и те же два метода к множеству классов, мы точно так же могли   
бы определить их в одном общем суперклассе и наследовать его в подклассах.   
Однако на практике метаклассы способны обеспечить намного более динамич-  
ное поведение. Например, расширяемый класс мог бы формироваться на осно-  
ве некоторой произвольной логики во время выполнения:  
# Класс может формироваться, исходя из некоторых условий во время выполнения  
   
class MetaExtend(type):  
 def \_\_new\_\_(meta, classname, supers, classdict):  
 if sometest():  
 classdict[‘eggs’] = eggsfunc1  
 else:  
 classdict[‘eggs’] = eggsfunc2  
 if someothertest():  
 classdict[‘ham’] = hamfunc  
 else:  
 classdict[‘ham’] = lambda \*args: ‘Not supported’  
 return type.\_\_new\_\_(meta, classname, supers, classdict)  
Метаклассы против декораторов: раунд 2  
Если вы еще способны следить за развитием событий в этой главе, примите   
также к сведению, что с функциональной точки зрения, область применения   
декораторов классов, рассматривавшихся в предыдущей главе, часто пересека-  
ется с областью применения метаклассов, которые обсуждаются в этой главе.   
Это обусловлено тем, что:  
 •  
Декораторы классов присваивают оригинальным именам классов резуль-  
тат вызова функции в конце инструкции class.

Пример: добавление методов в классы   
1183  
 •  
Принцип действия метаклассов основан на переадресации процедуры соз-  
дания объекта класса в конце инструкции class.  
Несмотря на некоторые различия в моделях, на практике они обычно ис-  
пользуются для достижения одних и тех же целей, хотя и разными способа-  
ми. Фактически декораторы классов могут использоваться для управления   
экземплярами классов и самими классами. Однако если управление класса-  
ми с применением декораторов является вполне естественным решением, то   
управление экземплярами с помощью метаклассов может вызывать некото-  
рые затруднения. Метаклассы лучше приспособлены для управлениями объ-  
ектами классов.  
Расширение с помощью декоратора  
Так, метакласс в примере из предыдущего раздела, добавляющий два метода   
в класс на этапе его создания, можно было бы реализовать в виде декоратора   
класса – в подобных случаях декораторы можно считать аналогами метода \_\_  
init\_\_ метаклассов, так как ко времени вызова декоратора объект класса уже   
существует. Так же, как и при использовании метаклассов, в подобных слу-  
чаях декораторы сохраняют оригинальный тип класса, потому что они не до-  
бавляют объекты с обертывающей логикой. Следующий пример выводит те же   
результаты, что и предыдущий пример с метаклассом:  
# Расширение с помощью декоратора: реализует те же действия, что и метод   
# \_\_init\_\_ метакласса  
   
def eggsfunc(obj):  
 return obj.value \* 4  
   
def hamfunc(obj, value):  
 return value + ‘ham’  
   
def Extender(aClass):  
 aClass.eggs = eggsfunc # Управляет классом, а не экземпляром  
 aClass.ham = hamfunc # Аналог метода \_\_init\_\_ метакласса  
 return aClass  
   
@Extender  
class Client1: # Client1 = Extender(Client1)  
 def \_\_init\_\_(self, value): # Повторно присваивает оригинальному имени   
 self.value = value # класса значение функции-декоратора   
 def spam(self): # в конце инструкции class  
 return self.value \* 2  
   
@Extender  
class Client2:  
 value = ‘ni?’  
   
X = Client1(‘Ni!’) # X – экземпляр класса Client1  
print(X.spam())  
print(X.eggs())  
print(X.ham(‘bacon’))  
   
Y = Client2()  
print(Y.eggs())  
print(Y.ham(‘bacon’))

1184   
Глава 39. Метаклассы   
Другими словами, по крайней мере, в некоторых случаях декораторы спо-  
собны управлять классами так же легко, как и метаклассы. Однако обратное   
нельзя сказать о метаклассах – метаклассы могут использоваться для управ-  
ления экземплярами, но только с применением определенной доли магии. Эта   
возможность демонстрируется в следующем разделе.  
Управление экземплярами вместо классов  
Как мы только что видели, декораторы часто могут играть ту же роль управ-  
ления классами, что и метаклассы. Метаклассы, в свою очередь, часто могут   
играть ту же роль управления экземплярами, что и декораторы, но реализо-  
вать такое управление немного сложнее. То есть:  
 •  
Декораторы классов способны управлять и классами, и экземплярами.  
 •  
Метаклассы также способны управлять и классами, и экземплярами, но   
управление экземплярами реализуется сложнее.  
При этом в одних приложениях оказывается удобнее использовать один прием,   
а в других – другой. Например, рассмотрим следующий декоратор классов из   
примера в предыдущей главе. Он используется для вывода трассировочных со-  
общений, когда выполняется попытка получить значение атрибута с обычным   
именем:   
# Декоратор классов, используемый для трассировки попыток получить значения   
# атрибутов экземпляров извне  
   
def Tracer(aClass): # На этапе декорирования @  
 class Wrapper:  
 def \_\_init\_\_(self, \*args, \*\*kargs): # На этапе создания экземпляра  
 self.wrapped = aClass(\*args, \*\*kargs) # Использует имя в  
 def \_\_getattr\_\_(self, attrname): # объемлющей области  
 print(‘Trace:’, attrname) # Перехватывает обращения ко   
 # всем атр., кроме .wrapped  
 return getattr(self.wrapped, attrname) # Делегирует обращения   
 return Wrapper # обернутому объекту  
   
@Tracer  
class Person: # Person = Tracer(Person)  
 def \_\_init\_\_(self, name, hours, rate): # Wrapper запоминает Person  
 self.name = name  
 self.hours = hours  
 self.rate = rate # Доступ изнутри методов не трассируется  
 def pay(self):  
 return self.hours \* self.rate  
   
bob = Person(‘Bob’, 40, 50) # bob – в действительности экземпляр Wrapper  
print(bob.name) # экземпляр Person встраивается во Wrapper  
print(bob.pay()) # Вызовет \_\_getattr\_\_  
Если запустить этот пример, декоратор использует прием повторного присваи-  
вания значения оригинальному имени класса, чтобы обернуть объекты экзем-  
пляров объектом, который выводит трассировочные сообщения, как показано   
ниже:  
Trace: name  
Bob  
Trace: pay  
2000

Пример: добавление методов в классы   
1185  
Того же эффекта можно добиться с использованием метакласса, однако реа-  
лизация будет выглядеть не такой простой. Метаклассы предназначены для   
управления процедурой создания объектов классов, и их интерфейс приспосо-  
блен для решения этой задачи. Чтобы реализовать управление экземплярами   
с помощью метакласса, нам придется поколдовать. Следующий метакласс дает   
тот же эффект и выводит те же результаты, что и предыдущий декоратор:  
# Управление экземплярами подобно предыдущему примеру, но с помощью метакласса  
   
def Tracer(classname, supers, classdict): # На этапе создания класса  
 aClass = type(classname, supers, classdict) # Создать клиентский класс  
 class Wrapper:  
 def \_\_init\_\_(self, \*args, \*\*kargs): # На этапе создания экземпляра  
 self.wrapped = aClass(\*args, \*\*kargs)  
 def \_\_getattr\_\_(self, attrname):  
 print(‘Trace:’, attrname) # Перехватывает обращения ко  
 # всем атр., кроме .wrapped  
 return getattr(self.wrapped, attrname) # Делегирует обращения   
 return Wrapper # обернутому объекту  
   
class Person(metaclass=Tracer): # Создать класс Person с   
 # метаклассом Tracer  
 def \_\_init\_\_(self, name, hours, rate): # Wrapper запоминает Person  
 self.name = name  
 self.hours = hours  
 self.rate = rate # Доступ изнутри методов не трассируется  
 def pay(self):  
 return self.hours \* self.rate  
   
bob = Person(‘Bob’, 40, 50) # bob – в действительности экземпляр Wrapper  
print(bob.name) # экземпляр Person встраивается во Wrapper  
print(bob.pay()) # Вызовет \_\_getattr\_\_  
Эта реализация работает, как и ожидалось, но она опирается на использова-  
ние двух хитростей. Во-первых, в этом примере используется простая функция   
вместо класса, потому что подклассы type должны придерживаться протоко-  
ла создания объектов. Во-вторых, целевой класс должен создаваться вручную   
вызовом класса type – это необходимо, чтобы вернуть экземпляр обертки, тог-  
да как метаклассы создают и возвращают целевой класс. В действительности   
в этом примере мы использовали протокол метаклассов, чтобы имитировать   
декоратор, а не наоборот. Поскольку и метаклассы, и декораторы вызываются   
в конце инструкции class, во многих случаях они оказываются вариациями на   
одну и ту же тему. Эта версия метакласса воспроизводит те же результаты, что   
и декоратор:  
Trace: name  
Bob  
Trace: pay  
2000  
Изучите обе версии примера самостоятельно, чтобы оценить различия между   
ними. Вообще говоря, метаклассы лучше подходят для управления классами   
из-за особенностей их предназначения. Декораторы классов могут использо-  
ваться для управления и экземплярами, и классами, однако они не всегда яв-  
ляются лучшим выбором для замены метаклассов в сложных ситуациях, для   
описания которых недостаточно места в этой книге (если после прочтения этой

1186   
Глава 39. Метаклассы   
главы у вас появится желание узнать больше о декораторах и метаклассах, вос-  
пользуйтесь поиском в Сети или обратитесь к стандартным руководствам по   
языку Python). Следующий раздел завершает эту главу еще одним типичным   
примером использования метаклассов – автоматическое выполнение операций   
над методами классов.  
Пример: применение декораторов к методам  
Как мы видели в предыдущем разделе, благодаря тому, что метаклассы и деко-  
раторы вызываются в конце инструкции class, они часто оказываются взаимо-  
заменяемыми, несмотря на различия в синтаксисе. Часто выбор между ними   
во многом зависит от личных предпочтений. Кроме того, существует возмож-  
ность использовать их в различных комбинациях как взаимодополняющие ин-  
струменты. В этом разделе мы как раз исследуем пример такой комбинации –   
применение декоратора функции ко всем методам класса.  
Трассировка с декорированием вручную  
В предыдущей главе мы реализовали два декоратора функций, один из кото-  
рых выполняет трассировку и подсчет количества вызовов декорированной   
функции, а другой выполняет хронометраж вызовов. Тогда были представле-  
ны несколько версий этих декораторов – одни из этих версий могут применять-  
ся и к функциям, и к методам, а другие – нет. Для справки ниже приводят-  
ся окончательные версии декораторов, помещенные в файл модуля, чтобы их   
можно было использовать в разных программах:  
# Файл mytools.py: коллекция различных декораторов  
   
def tracer(func): # Вместо класса с методом \_\_call\_\_ используется функция  
 calls = 0 # Иначе “self” будет представлять экземпляр декоратора!  
 def onCall(\*args, \*\*kwargs):  
 nonlocal calls  
 calls += 1  
 print(‘call %s to %s’ % (calls, func.\_\_name\_\_))  
 return func(\*args, \*\*kwargs)  
 return onCall  
   
import time  
def timer(label=’’, trace=True): # Аргументы декоратора: сохраняются  
 def onDecorator(func): # На этапе декорирования сохраняется функция  
 def onCall(\*args, \*\*kargs): # При вызове: вызывается оригинал  
 start = time.clock() # Информация в области видимости +  
 result = func(\*args, \*\*kargs) # атрибуты функции  
 elapsed = time.clock() - start  
 onCall.alltime += elapsed  
 if trace:  
 format = ‘%s%s: %.5f, %.5f’  
 values = (label, func.\_\_name\_\_, elapsed, onCall.alltime)  
 print(format % values)  
 return result  
 onCall.alltime = 0  
 return onCall  
 return onDecorator

Пример: применение декораторов к методам   
1187  
Как мы узнали в предыдущей главе, чтобы воспользоваться этими декорато-  
рами вручную, достаточно импортировать их из модуля и поместить с исполь-  
зованием синтаксиса декораторов @ перед каждым методом, вызовы которого   
предполагается отслеживать или хронометрировать:  
from mytools import tracer  
   
class Person:  
 @tracer  
 def \_\_init\_\_(self, name, pay):  
 self.name = name  
 self.pay = pay  
   
 @tracer  
 def giveRaise(self, percent): # giveRaise = tracer(giverRaise)  
 self.pay \*= (1.0 + percent) # onCall запоминает giveRaise  
   
 @tracer  
 def lastName(self): # lastName = tracer(lastName)  
 return self.name.split()[-1]  
   
bob = Person(‘Bob Smith’, 50000)  
sue = Person(‘Sue Jones’, 100000)  
print(bob.name, sue.name)  
sue.giveRaise(.10) # Вызовет onCall(sue, .10)  
print(sue.pay)  
print(bob.lastName(), sue.lastName()) # Вызовет onCall(bob), запоминает   
 # lastName  
Если запустить этот пример, мы получим следующие результаты – вызовы   
декорированных методов перехватываются логикой декоратора и затем деле-  
гируются оригинальным методам, потому что имена оригинальных методов   
были связаны с декоратором:  
call 1 to \_\_init\_\_  
call 2 to \_\_init\_\_  
Bob Smith Sue Jones  
call 1 to giveRaise  
110000.0  
call 1 to lastName  
call 2 to lastName  
Smith Jones  
Трассировка с использованием метаклассов   
и декораторов  
Пример декорирования вручную, представленный в предыдущем разделе, дей-  
ствует, как ожидалось, но от нас потребовалось добавить декоратор перед каж-  
дым методом, для которого требуется выполнить трассировку, а позднее нам   
придется вручную удалять эти декораторы, когда надобность в трассировке от-  
падет. Если у нас возникнет необходимость выполнить трассировку всех мето-  
дов класса, в крупных программах это может оказаться достаточно утомитель-  
ным. Было бы лучше, если бы у нас имелась возможность каким-либо образом   
автоматически применить декоратор tracer сразу ко всем методам класса.

1188   
Глава 39. Метаклассы   
Именно такую возможность дают нам метаклассы – благодаря тому, что они   
вызываются уже после того, как класс будет сконструирован, метаклассы ока-  
зываются естественным инструментом добавления декораторов ко всем мето-  
дам класса. Сканируя словарь с именами атрибутов класса и проверяя их на   
принадлежность к функциям, мы можем автоматически вызывать декораторы   
для методов и присваивать результаты оригинальным именам. Этот прием по-  
зволяет автоматически присвоить декораторы именам методов, но при этом мы   
можем использовать его более глобально:  
# Метакласс, применяющий декоратор tracer ко всем методам клиентского класса  
   
from types import FunctionType  
from mytools import tracer  
   
class MetaTrace(type):  
 def \_\_new\_\_(meta, classname, supers, classdict):  
 for attr, attrval in classdict.items():  
 if type(attrval) is FunctionType: # Метод?  
 classdict[attr] = tracer(attrval) # Декорировать  
 return type.\_\_new\_\_(meta, classname, supers, classdict)# Создать класс  
   
class Person(metaclass=MetaTrace):  
 def \_\_init\_\_(self, name, pay):  
 self.name = name  
 self.pay = pay  
 def giveRaise(self, percent):  
 self.pay \*= (1.0 + percent)  
 def lastName(self):  
 return self.name.split()[-1]  
   
bob = Person(‘Bob Smith’, 50000)  
sue = Person(‘Sue Jones’, 100000)  
print(bob.name, sue.name)  
sue.giveRaise(.10)  
print(sue.pay)  
print(bob.lastName(), sue.lastName())  
Если запустить этот пример, он выведет те же результаты, что и прежде, – вы-  
зовы методов сначала будут перехвачены декоратором tracer, а затем будут вы-  
званы оригинальные методы:   
call 1 to \_\_init\_\_  
call 2 to \_\_init\_\_  
Bob Smith Sue Jones  
call 1 to giveRaise  
110000.0  
call 1 to lastName  
call 2 to lastName  
Smith Jones  
Результаты, которые вы видите здесь, являются результатом совместной рабо-  
ты декоратора и метакласса – метакласс автоматически применяет декоратор   
функций к каждому методу на этапе создания класса, а декоратор функций   
автоматически перехватывает вызовы методов, чтобы вывести трассировочные   
сообщения. Данная комбинация «просто работает» благодаря универсально-  
сти обоих инструментов.

Пример: применение декораторов к методам   
1189  
Применение произвольного декоратора к методам  
В предыдущем примере метакласс работает с единственным, конкретным де-  
коратором функций – tracer. Однако совсем несложно обобщить его, чтобы   
обеспечить возможность применения любого декоратора ко всем методам клас-  
са. Все, что для этого требуется сделать, – это добавить объемлющую область   
видимости, чтобы сохранить требуемый декоратор, практически так же, как   
мы делали это в декораторах в предыдущей главе. Ниже приводится пример   
реализации такого обобщения и затем пример применения декоратора tracer:  
# Фабрика метаклассов: применяет любой декоратор ко всем методам класса  
   
from types import FunctionType  
from mytools import tracer, timer  
   
def decorateAll(decorator):  
 class MetaDecorate(type):  
 def \_\_new\_\_(meta, classname, supers, classdict):  
 for attr, attrval in classdict.items():  
 if type(attrval) is FunctionType:  
 classdict[attr] = decorator(attrval)  
 return type.\_\_new\_\_(meta, classname, supers, classdict)  
 return MetaDecorate  
   
class Person(metaclass=decorateAll(tracer)):# Применить произвольный декоратор  
 def \_\_init\_\_(self, name, pay):  
 self.name = name  
 self.pay = pay  
 def giveRaise(self, percent):  
 self.pay \*= (1.0 + percent)  
 def lastName(self):  
 return self.name.split()[-1]  
   
bob = Person(‘Bob Smith’, 50000)  
sue = Person(‘Sue Jones’, 100000)  
print(bob.name, sue.name)  
sue.giveRaise(.10)  
print(sue.pay)  
print(bob.lastName(), sue.lastName())  
Если запустить этот пример, он снова выведет те же результаты, что и преды-  
дущие примеры, – мы по-прежнему декорируем все методы клиентского клас-  
са с помощью декоратора функций tracer, но делаем это более обобщенным спо-  
собом:  
call 1 to \_\_init\_\_  
call 2 to \_\_init\_\_  
Bob Smith Sue Jones  
call 1 to giveRaise  
110000.0  
call 1 to lastName  
call 2 to lastName  
Smith Jones  
Теперь, чтобы применить другой декоратор, нам достаточно будет просто за-  
менить имя декоратора в заголовке инструкции class. Например, чтобы вос-

1190   
Глава 39. Метаклассы   
пользоваться декоратором функций timer, показанным ранее, мы могли бы при   
определении нашего класса использовать любую из двух последних строк за-  
головков, приведенных ниже, – первая из них применяет декоратор timer со   
значениями аргументов по умолчанию, а вторая задает текст в аргументе label:  
class Person(metaclass=decorateAll(tracer)): # Применяет tracer  
   
class Person(metaclass=decorateAll(timer())): # Применяет timer, со значениями   
 # аргументов по умолчанию  
class Person(metaclass=decorateAll(timer(label=’\*\*’))): # Декоратор с   
 # аргументами  
Обратите внимание, что в данном случае отсутствует возможность определять   
значения аргументов декоратора отдельно для каждого метода, но имеется воз-  
можность передавать аргументы, которые будут применяться ко всем мето-  
дам, как показано выше. Чтобы опробовать такую возможность, используйте   
последнюю строку с объявлением метакласса, применяющего декоратор timer,   
и добавьте следующие строки в конец сценария:  
# Если используется timer: общее время работы каждого метода  
   
print(‘-’\*40)  
print(‘%.5f’ % Person.\_\_init\_\_.alltime)  
print(‘%.5f’ % Person.giveRaise.alltime)  
print(‘%.5f’ % Person.lastName.alltime)  
Ниже приводятся результаты работы измененного сценария – теперь мета-  
класс обертывает методы декоратором timer, поэтому мы можем сказать, как   
долго работал каждый из них:  
\*\*\_\_init\_\_: 0.00001, 0.00001  
\*\*\_\_init\_\_: 0.00001, 0.00002  
Bob Smith Sue Jones  
\*\*giveRaise: 0.00001, 0.00001  
110000.0  
\*\*lastName: 0.00001, 0.00001  
\*\*lastName: 0.00001, 0.00002  
Smith Jones  
----------------------------------------  
0.00002  
0.00001  
0.00002  
Метаклассы против декораторов: раунд 3  
Здесь область применения декораторов также пересекается с областью приме-  
нения метаклассов. В следующей версии мы заменили метакласс из предыду-  
щего примера декоратором класса. Здесь определяется и используется декора-  
тор класса, который применяет декоратор функций ко всем методам класса.   
Хотя предыдущее предложение больше напоминает заклинание, чем техниче-  
ское описание, тем не менее, эта версия действует вполне естественным обра-  
зом – декораторы в языке Python поддерживают возможность произвольного   
вложения и комбинирования:

Пример: применение декораторов к методам   
1191  
# Фабрика декораторов классов: применяет любой декоратор ко всем методам класса  
   
from types import FunctionType  
from mytools import tracer, timer  
   
def decorateAll(decorator):  
 def DecoDecorate(aClass):  
 for attr, attrval in aClass.\_\_dict\_\_.items():  
 if type(attrval) is FunctionType:  
 setattr(aClass, attr, decorator(attrval)) # Не \_\_dict\_\_  
 return aClass  
 return DecoDecorate  
   
@decorateAll(tracer) # Используется декоратор класса  
class Person: # Применяет декоратор func к методам  
 def \_\_init\_\_(self, name, pay): # Person = decorateAll(..)(Person)  
 self.name = name # Person = DecoDecorate(Person)  
 self.pay = pay  
 def giveRaise(self, percent):  
 self.pay \*= (1.0 + percent)  
 def lastName(self):  
 return self.name.split()[-1]  
   
bob = Person(‘Bob Smith’, 50000)  
sue = Person(‘Sue Jones’, 100000)  
print(bob.name, sue.name)  
sue.giveRaise(.10)  
print(sue.pay)  
print(bob.lastName(), sue.lastName())  
Если запустить этот пример в таком виде, декоратор класса применит декора-  
тор функций tracer ко всем методам и будет выводить трассировочные сообще-  
ния при вызове любых методов (этот пример выводит те же результаты, что   
и предыдущая версия, основанная на метаклассе):  
call 1 to \_\_init\_\_  
call 2 to \_\_init\_\_  
Bob Smith Sue Jones  
call 1 to giveRaise  
110000.0  
call 1 to lastName  
call 2 to lastName  
Smith Jones  
Обратите внимание, что декоратор класса возвращает оригинальный, расши-  
ренный класс, а не обертку для его экземпляров (что характерно для приема   
обертывания). Как и в версии с метаклассом, мы сохраняем тип оригинально-  
го класса� экземпляр класса Person – это экземпляр класса Person, а не некото-  
рого класса-обертки. Фактически этот декоратор класса выполняется только   
на этапе создания класса – он вообще не участвует в операциях создания эк-  
земпляров.  
Это отличие может иметь значение в программах, требующих, чтобы опера-  
ции проверки типов экземпляров возвращали оригинальный класс, а не оберт-  
ку. Расширяя сам класс, а не его экземпляры, декораторы классов сохраняют

1192   
Глава 39. Метаклассы   
оригинальный тип класса. При этом методы класса уже не являются ориги-  
нальными функциями, потому что теперь они привязаны к декораторам, но на   
практике это не очень важно. Впрочем, то же самое происходит и в версии на   
основе метакласса.  
Обратите также внимание, что подобно версии с метаклассом в этом примере   
не поддерживается возможность определять аргументы декоратора функций   
отдельно для каждого метода, но имеется возможность передавать аргументы,   
которые будут применяться ко всем методам. Чтобы использовать этот же при-  
ем для применения декоратора timer, например, достаточно использовать лю-  
бую из двух последних строк, следующих ниже, непосредственно перед опреде-  
лением класса, – первая из них применяет декоратор со значениями аргумен-  
тов по умолчанию, а вторая явно определяет значение одного из аргументов:  
@decorateAll(tracer) # Декорирует все методы декоратором tracer  
   
@decorateAll(timer()) # Декорирует все методы декоратором timer со значениями   
 # аргументов по умолчанию defaults  
   
@decorateAll(timer(label=’@@’)) # То же самое, но определяет   
 # аргумент декоратора  
Как и прежде, воспользуемся последним объявлением декоратора и добавим   
следующие строки в конец сценария, чтобы протестировать наш пример с дру-  
гим декоратором:  
# Если используется timer: общее время работы каждого метода  
   
print(‘-’\*40)  
print(‘%.5f’ % Person.\_\_init\_\_.alltime)  
print(‘%.5f’ % Person.giveRaise.alltime)  
print(‘%.5f’ % Person.lastName.alltime)  
Сценарий выведет те же результаты – мы получили результаты хронометража   
отдельных вызовов и всех вызовов в сумме для каждого метода, но при этом мы   
передали декоратору timer другое значение в аргументе label:  
@@\_\_init\_\_: 0.00001, 0.00001  
@@\_\_init\_\_: 0.00001, 0.00002  
Bob Smith Sue Jones  
@@giveRaise: 0.00001, 0.00001  
110000.0  
@@lastName: 0.00001, 0.00001  
@@lastName: 0.00001, 0.00002  
Smith Jones  
----------------------------------------  
0.00002  
0.00001  
0.00002  
Как видите, метаклассы и декораторы классов часто оказываются не только   
взаимозаменяемыми, но и взаимодополняющими инструментами. И те и дру-  
гие предоставляют дополнительные и мощные возможности управления объ-  
ектами классов и экземпляров, так как оба инструмента в конечном итоге по-  
зволяют добавлять программный код, который автоматически вызывается   
в процессе создания класса. Хотя в некоторых, более сложных случаях может

Пример: применение декораторов к методам   
1193  
оказаться предпочтительнее использовать какой-то определенный инстру-  
мент, тем не менее, в большинстве ситуаций выбор того или иного инструмента   
или их комбинаций во многом зависит от личных предпочтений.  
«Необязательные» особенности языка  
В начале этой главы я привел цитату о том, что метаклассы не представ-  
ляют интереса для 99% программистов на языке Python, чтобы подчер-  
кнуть их относительную сложность. Однако это утверждение недоста-  
точно точное, и не только в количественной оценке.   
Автор цитаты – мой друг с самых первых лет знакомства с языком   
Python, и я совсем не хотел кого-нибудь задеть. Кроме того, я часто де-  
лал подобные заявления об особенностях языка, малопонятных мне са-  
мому, – даже в этой самой книге.  
Тем не менее проблема состоит в том, что в действительности оценка   
особенностей, вынесенная в заголовок, относится только к тем, кто ра-  
ботает в одиночку и пользуется только тем программным кодом, кото-  
рый написан ими самими. Как только подобная «необязательная» осо-  
бенность языка начинает использоваться кем-либо в организации, она   
перестает быть необязательной – она фактически становится обязатель-  
ной для всех в этой организации. То же относится и к стороннему про-  
граммному обеспечению, используемому в ваших программах, – если   
автор этого программного обеспечения использует дополнительные воз-  
можности языка, они перестают быть необязательными для вас, потому   
что вам придется изучить эту особенность, чтобы иметь возможность   
использовать или изменять программный код.  
Такое положение вещей касается всех дополнительных инструментов,   
перечисленных в начале этой главы, – декораторов, свойств, дескрип-  
торов, метаклассов и так далее. Если кто-то начинает использовать их   
в программах, с которыми вам придется работать, они автоматически   
становятся обязательной частью ваших знаний. То есть никакая «нео-  
бязательная» особенность в действительности не является необяза-  
тельной. Большинству из нас не приходится выбирать.  
Именно поэтому многие ветераны Python (включая и меня) иногда жа-  
луются на то, что с течением времени язык Python все разрастается   
и усложняется. Новые особенности, добавляемые ветеранами, поднима-  
ют интеллектуальную планку для начинающих. Основные идеи языка   
Python, такие как динамическая типизация и встроенные типы, оста-  
лись по сути теми же, при этом изучение дополнительных особенностей   
может оказаться обязательным для любых программистов, пользую-  
щихся языком Python. Поэтому я решил осветить эти темы в данной   
книге, хотя о них почти не говорилось в предыдущих изданиях. Невоз-  
можно пропустить обсуждение расширенных особенностей, если они   
используются в программном коде, который вы должны уметь читать   
и понимать.

1194   
Глава 39. Метаклассы   
С другой стороны, многие читатели могут изучать расширенные темы   
по мере необходимости. И, честно признаться, прикладные программи-  
сты обычно тратят больше времени на изучение библиотек и расшире-  
ний, а не на расширенные, и иногда таинственные особенности языка.   
Например, книга «Программирование на Python»1, продолжающая эту   
книгу, главным образом обсуждает вопросы применения прикладных   
библиотек для решения таких задач, как разработка графического ин-  
терфейса, работа с базами данных и с Сетью, а не эзотерические инстру-  
менты языка.   
Обратная сторона такого роста языка состоит в том, что он стал более   
мощным. При аккуратном использовании такие инструменты, как де-  
кораторы и метаклассы, оказываются не только «крутыми» фишками,   
но и позволяют творчески мыслящим программистам создавать более   
гибкие и удобные прикладные интерфейсы для использования другими   
программистами. Как мы уже видели, они позволяют решать проблемы   
инкапсуляции и сопровождения программного кода.  
Является ли изучение этих расширенных особенностей языка необхо-  
димым для вас, – решать вам. К сожалению, нередко решение прини-  
мается исходя из уровня подготовки, – более опытные программисты   
применяют более сложные инструменты, забывая о других, менее опыт-  
ных программистах. Однако, к счастью, так бывает не всегда – хоро-  
шие программисты прекрасно понимают, что простота – это искусство,   
а дополнительные инструменты должны использоваться, только когда   
они действительно необходимы. Это справедливо для любых языков   
программирования, но для языка Python в особенности, потому что он   
часто предстает перед новыми или начинающими программистами как   
инструмент с расширенными возможностями.  
Если вы все еще не согласны, имейте в виду, что существует множество   
пользователей Python, которые с трудом ориентируются даже в основах   
ООП и классов. Это правда – я встречал тысячи таких пользователей.   
Программы на языке Python, требующие, чтобы их пользователи владе-  
ли всеми тонкостями метаклассов, декораторов и тому подобного, веро-  
ятно не должны рассчитывать на очень широкий рынок сбыта.  
В заключение  
В этой главе мы изучили метаклассы и исследовали примеры их применения   
на практике. Метаклассы позволяют вторгаться в процесс создания классов   
с целью управления и расширения пользовательских классов. Поскольку ме-  
таклассы автоматизируют эту процедуру, они могут оказаться более удачным   
выбором для разработчиков библиотек, чем вспомогательные функции или ис-  
1   
Лутц М. «Программирование на Python», 2-е изд. – Пер. с англ. – СПб.: Символ-  
Плюс, 2002. Четвертое издание этой книги выйдет в 2011 году.

Закрепление пройденного   
1195  
пользование программного кода вручную. Благодаря тому, что они инкапсули-  
руют такой программный код, они упрощают его сопровождение больше, чем   
другие подходы.  
Попутно мы также увидели, что области применения декораторов классов   
и метаклассов часто пересекаются: поскольку и те и другие вызываются в кон-  
це инструкции class, они иногда могут оказаться взаимозаменяемыми. Деко-  
раторы классов могут использоваться для управления объектами классов и эк-  
земпляров� метаклассы тоже могут играть обе эти роли, однако основная их   
цель – классы.  
Поскольку в этой главе рассматривались расширенные темы, вашему внима-  
нию будет представлено лишь несколько контрольных вопросов, которые по-  
могут закрепить основы (если вы зашли настолько далеко в главе о метаклас-  
сах, вы уже заслуживаете дополнительного поощрения!). Так как это послед-  
няя часть книги, в ней не будет предложено упражнений для самостоятельного   
решения. Обязательно прочитайте приложения, следующие далее, где описы-  
вается процесс установки и даются решения к упражнениям в предыдущих   
частях книги.  
Завершив работу с контрольными вопросами, вы достигнете официального   
окончания этой книги. Теперь, когда вы узнали язык Python изнутри и сна-  
Python изнутри и сна-  
 изнутри и сна-  
ружи, вашим следующим шагом в его освоении должно стать исследование   
библиотек, приемов и инструментов, пригодных для использования в предмет-  
ной области, в которой вы работаете. Язык Python получил настолько широ-  
Python получил настолько широ-  
 получил настолько широ-  
кое распространение, что вы без труда найдете достаточное количество ресур-  
сов, которые могут пригодиться практически в любых приложениях, которые   
только можно представить, – от приложений с графическим интерфейсом,   
веб-приложений и приложений баз данных до численного программирования,   
робототехники и системного администрирования.  
С этого места Python становится еще более увлекательным, но кроме того, на   
этом месте заканчивается эта книга и начинаются другие. Чтобы узнать, куда   
следует обратить свое внимание после прочтения этой книги, просмотрите спи-  
сок рекомендованной литературы, который приводится в предисловии. Удач-  
ного вам путешествия. И, конечно же, «Всегда ищите светлую сторону жизни!».  
Закрепление пройденного  
Контрольные вопросы  
1. Что такое метакласс?  
2. Как объявляется метакласс класса?  
3. Как пересекаются области применения декораторов классов и метаклассов   
с точки зрения управления классами?  
4. Как пересекаются области применения декораторов классов и метаклассов   
с точки зрения управления экземплярами?  
5. Какое место занимают декораторы и метаклассы в вашем арсенале? (И, по-  
жалуйста, выразите свой ответ в стиле популярной пародии Монти Пайтона.)

1196   
Глава 39. Метаклассы   
Ответы  
1. Метакласс – это класс, используемый для создания классов. Обычные клас-  
сы по умолчанию являются экземплярами класса type. Метаклассы – это   
обычно подклассы класса type, которые переопределяют методы создания   
классов с целью изменить процедуру создания класса, которая выполняет-  
ся в конце инструкции class. Обычно метаклассы переопределяют методы   
\_\_new\_\_ и \_\_init\_\_, чтобы внедриться в процесс создания класса. Метаклас-  
сы могут быть реализованы и другими способами – как простые функции,   
например. Но все они должны создавать и возвращать объект нового класса.  
2. В версии Python 3.0 и выше используется именованный аргумент в заголов-  
Python 3.0 и выше используется именованный аргумент в заголов-  
 3.0 и выше используется именованный аргумент в заголов-  
ке инструкции class: class C(metaclass=M). В версии Python 2.X используется   
атрибут класса: \_\_metaclass\_\_ = M. В 3.0 строка заголовка инструкции class   
может также включать объявления суперклассов (или базовых классов)   
перед именованным аргументом metaclass.  
3. Поскольку оба они автоматически вызываются в конце инструкции class,   
декораторы классов и метаклассы могут использоваться для управления   
классами. Декораторы присваивают оригинальному имени класса вызы-  
ваемый объект, а метаклассы перехватывают процедуру создания класса   
с помощью вызываемого объекта, при этом оба они могут играть сходные   
роли. Декораторы, управляющие классами, просто расширяют и возвра-  
щают оригинальный объект класса. Метаклассы расширяют класс после   
его создания.  
4. Поскольку оба они автоматически вызываются в конце инструкции class,   
декораторы классов и метаклассы могут использоваться для управления   
экземплярами, добавляя объект-обертку, который будет перехватывать   
операции создания экземпляров. Декораторы могут присваивать ориги-  
нальному имени класса вызываемый объект, который будет запускаться   
при попытке создать экземпляр и который сохраняет оригинальный объект   
класса. Метаклассы могут делать то же самое, но при этом они должны так-  
же создавать объект класса, поэтому использовать их в этой роли несколько   
сложнее.  
5. Наше главное оружие – декораторы... декораторы и метаклассы... мета-  
классы и декораторы.... У нас есть два оружия – метаклассы и декораторы...   
и безжалостная эффективность.... У нас есть три оружия – метаклассы,   
декораторы и безжалостная эффективность... и почти фанатичная предан-  
ность Гвидо.... У нас есть четыре... нет.... Среди нашего оружия.... В нашем   
арсенале... имеются такие элементы, как метаклассы, декораторы.... Я нач-  
ну сначала....

Часть IX.  
Приложения

Приложение A.  
   
Установка и настройка  
В этом приложении описываются подробности установки и настройки в по-  
мощь тем, кто впервые делает это.  
Установка интерпретатора Python  
Для запуска сценариев на языке Python необходимо иметь интерпретатор   
Python, поэтому первым шагом к использованию этого языка является уста-  
новка Python. Если он еще не установлен у вас на компьютере, вам необхо-  
димо получить, установить последнюю версию Python на свой компьютер и,   
возможно, настроить его. Для каждого компьютера эту процедуру необходи-  
мо выполнить всего один раз, а если вы запускаете фиксированные двоичные   
файлы (описываемые в главе 2), вам вообще не потребуется устанавливать ин-  
терпретатор.  
Возможно, Python уже установлен?   
Прежде чем приступать к установке, проверьте, возможно, на вашем компью-  
тере уже установлена свежая версия Python. Если вы работаете в операци-  
Python. Если вы работаете в операци-  
. Если вы работаете в операци-  
онной системе Linux, Mac OS X или UNIX, вполне вероятно, что Python уже   
установлен на вашем компьютере, правда, это может быть версия, на один-два   
выпуска старше самой новой. Вот как это можно проверить:   
 •  
В операционной системе �indows поищите пункт Python в меню кнопки   
Пуск (Start), расположенной в левом нижнем углу экрана.  
 •  
В Mac OS X откройте окно терминала (Aplications (Приложения) → Utilities (Ути-  
литы) → Terminal (Терминал)) и введите в командной строке команду python.   
 •  
В Linux и UNIX введите команду python в командной оболочке (в окне тер-  
минала) и посмотрите, что произойдет. Кроме того, можно попробовать по-  
искать подкаталог python на его обычном месте – в каталогах /usr/bin, /usr/  
local/bin и других.  
Если интерпретатор установлен, убедитесь, что это достаточно свежая версия.   
Несмотря на то, что для работы с большей частью этой книги подойдет лю-  
бая, достаточно свежая версия, тем не менее, это издание посвящено версиям

1200   
Приложение A. Установка и настройка   
Python 2.6 и 3.0, поэтому вам может потребоваться установить одну из этих   
версий, чтобы иметь возможность запускать примеры из книги.  
Коль скоро речь зашла о версиях, я рекомендую начинать с версии Python 3.0   
или выше, если вы только приступаете к изучению языка Python и вам не тре-  
Python и вам не тре-  
 и вам не тре-  
буется сопровождать программы, написанные для Python 2.X, – в противном   
случае вы, скорее всего, будете использовать версию Python 2.6. Некоторые по-  
Python 2.6. Некоторые по-  
 2.6. Некоторые по-  
пулярные системы, написанные на языке Python, все еще используют старые   
версии (версия 2.5 по-прежнему распространена достаточно широко), поэтому,   
если вам приходится работать с существующими системами, устанавливайте   
версию, которая наилучшим образом отвечает вашим потребностям, – в сле-  
дующем разделе описывается, где можно получить различные версии интер-  
претатора.  
Где получить Python  
Если вы не нашли Python у себя на компьютере, вам придется установить его.   
Могу вас обрадовать, Python является программным обеспечением, распро-  
Python является программным обеспечением, распро-  
 является программным обеспечением, распро-  
страняемым с открытыми исходным текстами, и его можно свободно загрузить   
из Сети, к тому же на большинстве платформ Python устанавливается доста-  
Python устанавливается доста-  
 устанавливается доста-  
точно просто.  
Самую свежую и лучшую версию Python всегда можно получить на официаль-  
Python всегда можно получить на официаль-  
 всегда можно получить на официаль-  
ном веб-сайте проекта http://www.python.org. Воспользуйтесь ссылкой Download   
(Загрузить) на этой странице и выберите версию для своей платформы. Здесь   
вы найдете уже собранные выполняемые файлы дистрибутивов для �indows   
(которые достаточно просто запустить, чтобы выполнить установку), файлы   
с установочными образами для Mac OS X, дистрибутивы с полными исходными   
текстами (которые требуется скомпилировать, чтобы установить интерпрета-  
тор) и другие.   
Интерпретатор Python в наши дни уже стал стандартной частью операционной   
системы Linux, поэтому вы также сможете найти пакеты в формате RPM (уста-  
новка произво дится с помощью утилиты rpm). На веб-сайте проекта Python   
можно также найти ссылки на страницы, где можно загрузить версии Python   
для других поддерживаемых платформ. Поиск в системе Google – еще один от-  
личный способ отыскать пакеты с Python. В числе прочих платформ, для ко-  
торых можно найти собранные версии интерпретатора Python, можно назвать   
iPods, Palm, сотовые телефоны Nokia, PlayStation и PSP, Solaris, AS/400 и �in-  
, Palm, сотовые телефоны Nokia, PlayStation и PSP, Solaris, AS/400 и �in-  
Palm, сотовые телефоны Nokia, PlayStation и PSP, Solaris, AS/400 и �in-  
, сотовые телефоны Nokia, PlayStation и PSP, Solaris, AS/400 и �in-  
Nokia, PlayStation и PSP, Solaris, AS/400 и �in-  
, PlayStation и PSP, Solaris, AS/400 и �in-  
PlayStation и PSP, Solaris, AS/400 и �in-  
 и PSP, Solaris, AS/400 и �in-  
PSP, Solaris, AS/400 и �in-  
, Solaris, AS/400 и �in-  
Solaris, AS/400 и �in-  
, AS/400 и �in-  
AS/400 и �in-  
/400 и �in-  
�in-  
dows Mobile  
Если вы скучаете по окружению UNIX, работая в ОС �indows, вас наверняка   
заинтересует оболочка Cygwin и версия Python для нее (http://www.cygwin.com).   
Cygwin – это библиотека и комплект инструментов, распространяемые на осно-  
 – это библиотека и комплект инструментов, распространяемые на осно-  
ве лицензии GPL, которые обеспечивают полную имитацию среды UNIX в ОС   
�indows. В состав этой оболочки входит уже готовая к использованию версия   
Python, которая использует все инструменты, предоставляемые ОС UNIX.  
Интерпретатор Python можно также обнаружить на компакт-дисках с дистри-  
Python можно также обнаружить на компакт-дисках с дистри-  
 можно также обнаружить на компакт-дисках с дистри-  
бутивами Linux, в составе некоторых программных продуктов и как прило-  
Linux, в составе некоторых программных продуктов и как прило-  
, в составе некоторых программных продуктов и как прило-  
жение к некоторым книгам по языку Python. Обычно они немного отстают от   
текущей версии, но это отставание, как правило, не очень велико.  
Кроме того, вы можете найти интерпретатор Python в составе некоторых ком-  
мерческих пакетов. Например, компания ActiveState распространяет Python

Установка интерпретатора Python   
1201  
в составе своего пакета ActivePython, в состав которого входят стандартный   
интерпретатор Python с расширениями для разработки программ в операци-  
Python с расширениями для разработки программ в операци-  
 с расширениями для разработки программ в операци-  
онной системе �indows, такими как Py�in32, интегрированная среда раз-  
�indows, такими как Py�in32, интегрированная среда раз-  
, такими как Py�in32, интегрированная среда раз-  
работки Python�in (описывается в главе 3) и другие часто используемые рас-  
Python�in (описывается в главе 3) и другие часто используемые рас-  
 (описывается в главе 3) и другие часто используемые рас-  
ширения. Кроме того, Python входит в состав Enthought Python Distribution –   
пакета программ для научных вычислений. Помимо этого существует версия   
Portable Python, которая настроена так, что позволяет запускать интерпрета-  
тор непосредственно с переносного устройства. Дополнительную информацию   
ищите в Сети.  
Наконец, если вам интересны альтернативные реализации Python, поищите   
в Сети Jython (версия Python для среды Java) и IronPython (версия Python для   
среды C#/.NET)� обе версии были описаны в главе 2. Описание процедуры уста-  
C#/.NET)� обе версии были описаны в главе 2. Описание процедуры уста-  
#/.NET)� обе версии были описаны в главе 2. Описание процедуры уста-  
NET)� обе версии были описаны в главе 2. Описание процедуры уста-  
)� обе версии были описаны в главе 2. Описание процедуры уста-  
новки этих систем выходит далеко за рамки этой книги.  
Установка  
Загрузив дистрибутив Python, его необходимо установить. Порядок установки   
сильно зависит от платформы, и поэтому ниже приводятся некоторые рекомен-  
дации по установке в некоторых основных платформах:  
Windows  
Для операционной системы �imdows дистрибутив Python поставляется   
в виде инсталляционного файла в формате MSI – просто щелкните дважды   
на ярлыке этого файла и отвечайте на вопросы нажатием кнопок Yes (Да)   
или Next (Далее), чтобы выполнить установку с параметрами по умолчанию.   
Установка по умолчанию включает в себя комплект документации, под-  
держку библиотеки построения графических интерфейсов tkinter (Tkinter   
в Python 2.6), базы данных shelve, а также среду разработки IDLE с графи-  
shelve, а также среду разработки IDLE с графи-  
, а также среду разработки IDLE с графи-  
IDLE с графи-  
 с графи-  
ческим интерфейсом. Обычно Python 3.0 и 2.6 устанавливается в каталог   
C:\Python30 и C:\Python26, хотя во время установки можно указать другой   
каталог.  
После установки в подменю Все программы (All Programs), в меню кнопки Пуск   
(Start), появляется дополнительное меню Python, в котором имеется пять   
пунктов, обеспечивающих быстрый доступ к наиболее типичным задачам:   
запуск IDLE, чтение документации, запуск интерактивного сеанса, чтение   
стандартных руководств по языку Python в веб-броузере и удаление. Боль-  
Python в веб-броузере и удаление. Боль-  
 в веб-броузере и удаление. Боль-  
шинство из этих действий связаны с концепциями, которые детально рас-  
сматривались в этой книге.  
После установки интерпретатор Python автоматически регистрирует себя   
в качестве программы, предназначенной для открытия файлов Python   
щелчком мыши (этот прием запуска программ описывается в главе 3). Су-  
ществует возможность собрать Python из исходных текстов в �indows, но   
обычно этого не делается.  
Одно замечание для пользователей �indows Vista: особенности системы   
безопасности текущей версии �indows Vista изменили некоторые из пра-  
�indows Vista изменили некоторые из пра-  
 Vista изменили некоторые из пра-  
Vista изменили некоторые из пра-  
 изменили некоторые из пра-  
вил использования инсталляционных файлов MSI. Обращайтесь за помо-  
MSI. Обращайтесь за помо-  
. Обращайтесь за помо-  
щью к тексту врезки «Установка Python из установочного файла формата   
MSI в �indows Vista» в этом приложении, если инсталляционный файл   
Python не запускается или если установка не выполняется в правильный   
каталог на вашей машине.

1202   
Приложение A. Установка и настройка   
Linux  
Для операционной системы Linux интерпретатор Python доступен в виде   
одного или нескольких файлов RPM, которые распаковыва ются обычным   
способом (за подробностями обращайтесь к странице справочного руко-  
водства по RPM). В зависимости от того, какие па кеты RPM вы загрузили,   
в одном может находиться сам интерпре татор Python, а в других – дополни-  
тельная поддержка tkinter и сре да разработки IDLE. Так как Linux являет-  
ся UNIX-подобной опера ционной системой, к нему применимы рекоменда-  
ции, которые да ются в следующем абзаце.  
UNIX  
В операционных системах UNIX Python обычно компилируется из дистри-  
бутива с исходными текстами на языке C. Обычно для этого требуется рас-  
паковать файл и запустить команды config и make – Py thon настроит про-  
цедуру сборки автоматически, в соответствии с системой, где выполняется   
сборка. Однако обязательно ознакомь тесь с содержимым файла README,   
где приводятся дополнитель ные замечания по процессу сборки. Поскольку   
Python является программным продуктом, распространяемым с открыты-  
ми исход ными текстами, его исходный программный код может свободно   
использоваться и распространяться.  
Процедура установки в других платформах может существенно отли чаться.   
Так, установка «Pippy» версии Python для PalmOS, напри мер, требует выпол-  
нения операции синхронизации вашего PDA, а Py thon для PDA Sharp Zaurus,   
работающего под управлением Linux, по ставляется в виде одного или более   
файлов .ipk, которые достаточно просто запустить, чтобы выполнить установ-  
ку. Дополнительные про цедуры установки для дистрибутивов в виде исполня-  
емых файлов и па кетов с исходными текстами прекрасно документированы,   
поэтому мы пропустим дальнейшие подробности.  
Установка Python из установочного файла   
формата MSI в Windows Vista  
Когда я писал эти строки, дистрибутив Python для �indows распростра-  
нялся в виде инсталляционного файла .msi. Файлы этого формата пре-  
красно работают в �indows XP (достаточно просто выполнить двойной   
щелчок мышью на этом файле, чтобы запустить его), однако в текущей   
версии �indows Vista все может оказаться не так просто. В частности,   
запуск инсталлятора щелчком мыши на моей машине привел к тому,   
что Python был установлен в корневой каталог диска C:  вместо вы-  
бранного по умолчанию каталога C:\PythonXX. Интерпретатор вполне   
работоспособен и в корневом каталоге, но это неправильное место для   
установки.  
Эта проблема обусловлена особенностями поведения системы безопас-  
ности в �indows Vista. Дело в том, что в действительности файлы MSI   
не являются настоящими исполняемыми файлами, поэтому они непра-  
вильно наследуют права администратора, даже если запускаются ад-  
министратором. Файлы MSI запускаются программой-инсталлятором   
MSI, ассоциация с которой определена в реестре �indows.

Настройка Python   
1203  
Трудно сказать, является ли это проблемой дистрибутива Python или   
она характерна для версии �indows Vista. Например, на своем послед-  
нем ноутбуке мне удалось установить Python 2.6 и 3.0 без каких-либо   
проблем. Однако чтобы установить Python 2.5.2 на свой карманный   
компьютер OQO, работающий под управлением �indows Vista, мне при-  
шлось воспользоваться командной строкой, чтобы выполнить установ-  
ку с правами администратора.   
Если установка Python выполняется в неправильный каталог, вос-  
пользуйтесь следующим решением: откройте меню кнопки Пуск (Start),   
выберите пункт Все программы (All Programs), выберите пункт Стандарт-  
ные (Accessories), щелкните правой кнопкой на пункте Командная строка   
(Command Prompt), в контекстном меню выберите пункт Запуск с правами   
администратора (Run as administrator) и в диалоге управления доступом   
выберите кнопку Продолжить (Continue). Теперь в окне Командная строка   
(Command Prompt) запустите команду cd, чтобы перейти в каталог, где на-  
ходится файл MSI с дистрибутивом Python (например, cd C:�ser\down-  
loads), и затем запустите программу-инсталлятор MSI, выполнив коман-  
ду вида: msiexec /i python-2.5.2.msi. Для завершения установки следуйте   
обычным указаниям мастера установки с графическим интерфейсом.  
Естественно, такое поведение может измениться через какое-то время.   
Эта процедура может оказаться необязательной в будущих версиях   
Vista и, возможно, появятся другие обходные пути (такие как отклю-  
чение системы безопасности Vista, если вы решитесь на это). Возмож-  
но так же, что сам установочный файл Python будет распространяться   
в другом формате, ликвидирующем эту проблему, например в виде на-  
стоящего исполняемого файла. Прежде чем использовать какие-либо   
обходные пути, попробуйте сначала просто запустить установку щелч-  
ком мыши, возможно, она уже работает должным образом.  
Настройка Python  
После установки Python вам может потребоваться выполнить некоторые на-  
Python вам может потребоваться выполнить некоторые на-  
 вам может потребоваться выполнить некоторые на-  
стройки, влияющие на то, как Python будет выполнять ваш программный код.   
(Если вы только начинаете знакомство с языком, возможно, вам лучше вообще   
пропустить этот раздел – для создания программ начального уровня обычно не   
требуется изменять какие-либо параметры настройки.)  
Вообще говоря, некоторые особенности поведения интерпретатора могут быть   
настроены с помощью переменных окружения и параметров командной стро-  
ки. В этом разделе мы коротко рассмотрим переменные окружения Python. Па-  
Python. Па-  
. Па-  
раметры командной строки, которые указываются при запуске программ на   
языке Python из командной строки системы, используются достаточно редко   
и часто играют узкоспециализированные роли – за дополнительной информа-  
цией по этому вопросу обращайтесь к другим источникам документации.  
Переменные окружения Python  
Переменные окружения, иногда известные как переменные команд ной обо-  
лочки или переменные DOS, определяются за пределами ин терпретатора Python

1204   
Приложение A. Установка и настройка   
и потому могут использоваться для настройки его поведения каждый раз, ког-  
да он запускается на данном компьюте ре. Интерпретатор Python пользуется   
множеством переменных окру жения, но лишь немногие из них используются   
достаточно часто� они приведены ниже. В табл. A.1. приводятся основные пере-  
менные окру жения, используемые для настройки интерпретатора Python.  
Таблица A.1. Переменные окружения, имеющие важное значение  
Переменная  
Назначение  
PATH (или path)  
Путь поиска файлов, используемый системой   
(используется при поиске исполняемого файла python)  
PYTHONPATH  
Путь поиска модулей Python (используется операцией   
импортирования)  
PYTHONSTARTUP  
Путь к интерактивному файлу запуска Python  
TCL\_LIBRARY,   
TK\_LIBRARY  
Переменные окружения для tkinter   
(расширение для создания графического интерфейса)  
Эти переменные используются вполне однозначно, и тем не менее, приведу не-  
сколько рекомендаций:  
PATH  
Переменная PATH определяет список каталогов, где операционная система   
будет пытаться отыскать исполняемые файлы программ. Обычно этот спи-  
сок должен включать каталог, где находится интерпретатор Python (файл   
python в операционной системе UNIX или файл python.exe в �indows).  
Вам вообще не придется настраивать эту переменную, если вы работаете   
в каталоге, где находится интерпретатор Python, или вводите в командной   
строке полный путь к выполняемому файлу интерпретатора. В �indows,   
например, настройки в переменной PATH не имеют значения, если перед за-  
PATH не имеют значения, если перед за-  
 не имеют значения, если перед за-  
пуском какого-либо программного кода выполнить команду cd C:\Python30   
(чтобы перейти в каталог, где находится интерпретатор Python), или вместо   
команды python вы всегда выполняете команду C:\Python30\python (в команде   
присутствует полный путь к исполняемому файлу). Кроме того, переменная   
окружения PATH в основном используется для запуска команд из командной   
строки – эта переменная не имеет значения при запуске программ щелчком   
мыши на ярлыке или из интегрированной среды разработки.  
PYTHONPATH  
Переменная окружения PYTHONPATH играет похожую роль: интерпретатор   
Python использует переменную PYTHONPATH во время поиска файлов модулей,   
когда они импортируются программами. Эта переменная содержит спи-  
сок каталогов в формате, зависящим от типа используемой платформы –   
в UNIX каталоги в списке отделяются двоеточием, а в �indows – точкой   
с запятой. Обычно этот список должен включать только каталоги с вашими   
исходными текстами. Содержимое этой переменной окружения включа-  
ется в список sys.path вместе с каталогом, в котором находится сценарий,   
с любыми каталогами, перечисленными в файлах .pth, и с каталогами стан-  
дартной библиотеки.  
Вам не потребуется настраивать эту переменную, если не импортиро вать   
модули, находящиеся в других каталогах, потому что интерпре татор всегда

Настройка Python   
1205  
автоматически пытается отыскать модули в домашнем каталоге програм-  
мы. Настраивать эту переменную придется, только если какой-либо модуль   
должен импортировать другой модуль, рас положенный в другом каталоге.   
Смотрите также далее обсуждение файлов .pth, которые являются альтер-  
нативой переменной PYTHONPATH. Дополнительную информацию о перемен-  
ной PYTHONPATH вы найдете в главе 21.  
PYTHONSTARTUP  
Если в переменной PYTHONSTARTUP указано полное имя файла с программным   
кодом на языке Python, интерпретатор будет запускать этот файл автома-  
Python, интерпретатор будет запускать этот файл автома-  
, интерпретатор будет запускать этот файл автома-  
тически всякий раз, когда запускается интерактивный сеанс работы с ин-  
терпретатором, как если бы инструкции из этого файла вводились вручную   
в интерактивной командной оболочке. Этот способ используется редко, но   
его удобно применять, когда необходимо обеспечить загрузку некоторых   
утилит для работы в интерактивной оболочке, т. к. позволяет сэкономить   
время на импортировании вручную.  
Настройки tkinter  
Если вы предполагаете использовать набор инструментальных средств по-  
строения графического интерфейса tkinter, вам может понадобиться запи-  
сать в две последние переменные из табл. A.1 имена катало гов библиотек Tcl   
и Tk (похоже на PYTHONPATH). Однако в �indows (где поддержка tkinter уста-  
навливается вместе с интерпретатором Python) это не требуется и обычно не   
требуется, если Tcl и Tk уста новлены в стандартные каталоги.  
Обратите внимание: эти настройки окружения являются внешними по отно-  
шению к интерпретатору Python, по этому совершенно неважно, когда будет   
выполнена их настройка: она может быть выполнена как до, так и после уста-  
новки Python, главное, что это должно быть сделано перед запуском интерпре-  
татора.  
Установка поддержки tkinter (и IDLE) в Linux  
Среда разработки IDLE, описанная в главе 2, представляет собой про-  
грамму на языке Python, которая использует библиотеку tkinter (Tkinter   
в Python 2.6) для создания графического интерфейса. Библиотека tkint-  
er – это набор инструментальных средств для построения графическо-  
го интерфейса. Она является стандартным компонентом дистрибутива   
Python для �indows и некоторых других платформ. Однако в некото-  
рых дистрибутивах Linux эта библиотека не входит в стандартный ком-  
плект устанавливаемых компонентов. Чтобы добавить в интерпретатор   
Python поддержку графического интерфейса в операционной системе   
Linux, попробуйте запустить команду yum tkinter, которая автоматиче-  
ски установит все необходимые библиотеки. Эта команда должна рабо-  
тать в дистрибутивах Linux (и в некоторых других системах), где имеет-  
Linux (и в некоторых других системах), где имеет-  
 (и в некоторых других системах), где имеет-  
ся программа установки пакетов yum.  
Как установить параметры конфигурации  
Способ установки переменных окружения, имеющих отношение к Python,   
и устанавливаемые значения зависят от типа компьютера, с которым вы ра-

1206   
Приложение A. Установка и настройка   
ботаете. Не забывайте, что вам не обязательно выполнять все эти настройки,   
особенно если вы работаете в среде IDLE (описанной в главе 3).  
Но предположим для иллюстрации, что у вас имеется несколько весь ма полез-  
ных модулей в каталогах utilities и package1 где-то в компью тере, и вам необ-  
ходимо иметь возможность импортировать их из фай лов модулей, расположен-  
ных не в домашнем каталоге. То есть, чтобы загрузить файл с именем spam.  
py из каталога utilities, вам необходимо обеспечить возможность выполнить   
инструкцию:  
import spam  
из другого файла, расположенного в каком-то другом каталоге. Для этого сле-  
дует одним из возможных способов настроить путь поиска мо дулей, чтобы   
включить в него каталог, содержащий файл spam.py. Ни же приводится не-  
сколько советов, как это можно сделать.  
Переменные окружения в UNIX/Linux  
В системе UNIX способ установки значения переменной окружения зависит от   
используемой командной оболочки. При использовании ко мандной оболочки   
csh для установки пути поиска модулей можно до бавить строку, как показано   
ниже, в свой файл .cshrc или .login:  
setenv PYTHONPATH /usr/home/pycode/utilities:/usr/lib/pycode/package1  
Она сообщает интерпретатору Python о том, что поиск импортируемых моду-  
лей должен выполняться в двух каталогах. Однако если вы ис пользуете ко-  
мандную оболочку ksh, настройки можно выполнить в фай ле .kshrc, и на этот   
раз строка будет иметь следующий вид:  
export PYTHONPATH=”/usr/home/pycode/utilities:/usr/lib/pycode/package1”  
Другие командные оболочки могут использовать другой (но достаточно похо-  
жий) синтаксис.  
Переменные DOS (Windows)  
Если вы используете MS-DOS или старую версию �indows, вам может потребо-  
ваться добавить определение переменных окружения в свой файл C:\autoexec.  
bat и перезагрузить компьютер, чтобы изменения вступили в силу. Команда   
настройки для таких компьютеров имеет синтаксис, уникальный для DOS:  
set PYTHONPATH=c:\pycode�tilities;d:\pycode\package1  
Вы можете ввести эту команду в окне сеанса DOS, но тогда настройки будут   
иметь эффект только в этом окне. Настройки в файле .bat сохра няются посто-  
янно и являются глобальными для всех программ.  
Переменные окружения в Windows  
В наиболее свежих версиях �indows, включая XP и Vista, имеется возмож-  
ность устанавли вать значение переменной окружения PYTHONPATH и других пере-  
менных с помощью графического интерфейса и тем самым избежать редакти-  
рования файлов и перезагрузки компьютера. В �indows XP выберите ярлык   
Система (System) в меню Панель управления (Control Panel), перейдите на вкладку   
Дополнительно (Advanced) и щелкните на кнопке Переменные среды (Environment   
Variables), чтобы отредактировать или добавить новые переменные (PYTHONPATH –

Настройка Python   
1207  
это обычно пользовательская переменная). Используйте те же имена перемен-  
ных, значения и синтаксис, которые были показаны выше, в команде настрой-  
ки для DOS. Процедура настройки выполняется похожим образом и в �indows   
Vista, но при этом вам, вероятно, придется проверить выполнение операций.  
Вам не потребуется перезагружать компьютер, но необходимо будет перезапу-  
стить интерпретатор Python, если к моменту внесения изме нений он уже был   
запущен – он воспринимает настройки пути только во время запуска. Если вы   
работаете в окне программы Командная строка (Command Prompt), вам наверняка   
придется перезапустить ее, чтобы настройки вступили в силу.  
Реестр Windows  
Если вы опытный пользователь �indows, вы можете также настроить путь   
с помощью редактора реестра �indows. Выберите пункт меню Пуск (Start) →   
Выполнить… (Run…) и введите команду regedit. Если этот инструмент редакти-  
рования установлен у вас на компьютере, вы сможе те с его помощью отыскать   
записи, имеющие отношение к Python, и вы полнить необходимые изменения.   
Это достаточно сложная процедура, при выполнении которой легко ошибить-  
ся, поэтому если вы не знакомы с реестром, я рекомендую использовать другие   
возможности (в действительности такой подход сродни нейрохирургической   
операции на вашем компьютере, поэтому будьте осторожны!).  
Файлы путей  
Наконец, если для настройки пути поиска модулей вы решили исполь зовать   
файл .pth, а не переменную окружения PYTHONPATH, в операцион ной системе   
�indows можно создать текстовый файл со следующим содержимым (файл C:\  
Python30\mypath.pth):  
c:\pycode�tilities  
d:\pycode\package1  
Его содержимое будет отличаться для разных платформ, а каталог его разме-  
щения может отличаться как в зависимости от платформы, так и в зависимо-  
сти от версии Python. Интерпретатор отыскивает эти фай лы автоматически во   
время запуска.  
Имена каталогов в файлах пути могут быть абсолютными или относи тельными   
по отношению к каталогу, где находится файл пути. Допус кается использовать   
несколько файлов .pth (все каталоги, перечислен ные в них, будут добавлены   
в путь поиска), а сами файлы .pth могут размещаться в любых каталогах, кото-  
рые проверяются автоматиче ски, в зависимости от используемой платформы   
и версии Python. На пример, Python N.M пытается отыскать такие файлы в ка-  
талогах C:\PythonNM и C:\PythonNM\Lib\site-packages в операционной систе-  
ме �indows и в каталогах /usr/local/lib/python.M/site-packages и /usr/ local/lib/  
site-python в Unix и Linux. Подробнее об использовании файлов .pth и настрой-  
ке переменной sys.path рассказывается в главе 21.  
Поскольку эти настройки часто бывают необязательными и эта книга не опи-  
сывает командные оболочки операционных систем, я оставляю освещение под-  
робностей другим источникам информации. За более подробными сведениями   
обращайтесь к страницам справочного руко водства своей командной оболочки   
или к другой документации. Если вы испытываете затруднения в определении   
того, какие настройки вам следует выполнить, обратитесь за помощью к свое-  
му системному администратору или другому опытному товарищу.

1208   
Приложение A. Установка и настройка   
Параметры командной строки интерпретатора  
При запуске Python из системной командной строки (она же командная обо-  
Python из системной командной строки (она же командная обо-  
 из системной командной строки (она же командная обо-  
лочка) вы можете передавать ему различные параметры, управляющие рабо-  
той интерпретатора. В отличие от переменных окружения, параметры команд-  
ной строки могут изменяться при каждом новом запуске сценария. Полная   
форма команды запуска интерпретатора версии 3.0 выглядит, как показано   
ниже (команда запуска для версии 2.6 выглядит почти также, за исключением   
некоторых отличий в параметрах):  
python [-bBdEhiOsSuvVWx?] [-c command | -m module-name | script | - ] [args]  
В большинстве случаев для запуска исходного файла программы с аргумен-  
тами, используемыми самой программой, в команде указываются только па-  
раметры script и args. В качестве иллюстрации рассмотрим следующий файл   
сценария main.py, который выводит список аргументов командной строки, до-  
ступных сценарию в виде списка sys.argv:  
# Файл main.py  
import sys  
print(sys.argv)  
В следующей команде обе ее части, python и main.py, могут содержать полный   
путь к файлу, а три аргумента (a b –c) предназначены для сценария – они до-  
ступны в виде списка sys.argv. Первый элемент списка sys.argv всегда содержит   
имя файла сценария, если оно известно:  
c:\Python30> python main.py a b –c # Типичный способ: запуск файла сценария  
[‘main.py’, ‘a’, ‘b’, ‘-c’]  
Другие параметры командной строки позволяют указывать выполняемый   
программный код непосредственно в командной строке (-c), принимать его из   
потока стандартного ввода (– (дефис) означает, что программный код должен   
читаться из канала или из входного файла) и так далее:  
c:\Python30> python -c “print(2 \*\* 100)” # Выполняемый код в командной строке  
1267650600228229401496703205376  
   
c:\Python30> python -c “import main” # Импортировать файл, чтобы выполнить его  
[‘-c’]  
   
c:\Python30> python - < main.py a b –c # Выполняемый код принимается   
 [‘-’, ‘a’, ‘b’, ‘-c’] # со стандартного ввода  
   
c:\Python30> python - a b -c < main.py # Имеет тот же эффект,   
[‘-’, ‘a’, ‘b’, ‘-c’] # что и предыдущая команда  
При получении параметра –m интерпретатор пытается отыскать указанный мо-  
дуль в пути поиска модулей (sys.path) и выполнить его как обычный сценарий   
(как модуль \_\_main\_\_). Расширение .py в данном случае следует отбросить, по-  
скольку здесь подразумевается имя модуля, а не файла:  
c:\Python30> python -m main a b –c # Отыскать/запустить модуль как сценарий  
[‘c:\\Python30\\main.py’, ‘a’, ‘b’, ‘-c’]  
Кроме того, параметр –m поддерживает запуск модулей в пакетах с использова-  
нием синтаксиса относительных путей, а также модулей в архивах .zip. Этот   
ключ обычно используется для запуска модулей отладчика pdb и профилиров-

Дополнительная информация   
1209  
щика profile при отладке сценариев из командной строки, а не в интерактив-  
ной оболочке. Однако похоже, что поведение этих модулей в таком режиме из-  
менилось в версии 3.0 (модуль profile, по всей видимости, затронуло удаление   
функции execfile в Python 3.0, а pdb выполняет ненужную трассировку множе-  
ства операций ввода/вывода в новом модуле io):  
c:\Python30> python -m pdb main.py a b -c # Отладить сценарий  
--Return--  
> c:\python30\lib\io.py(762)closed()->False  
-> return self.raw.closed  
(Pdb) c  
   
c:\Python30> C:\python26\python -m pdb main.py a b -c # В 2.6 ситуация лучше?  
> c:\python30\main.py(1)<module>()  
-> import sys  
(Pdb) c  
   
c:\Python30> python -m profile main.py a b -c # Профилировать сценарий  
   
c:\Python30> python -m cProfile main.py a b -c # Профилировщик с пониженным  
 # потреблением ресурсов  
Сразу вслед за командой «python» и перед ссылкой на программный код, кото-  
python» и перед ссылкой на программный код, кото-  
» и перед ссылкой на программный код, кото-  
рый требуется запустить, имеется возможность указать дополнительные аргу-  
менты, управляющие поведением самого интерпретатора. Эти аргументы ис-  
пользуются самим интерпретатором, и они не предназначены для управления   
поведением сценария. Например, параметр -O запускает интерпретатор в опти-  
мизированном режиме, -u – отключает механизм буферизации стандартных   
потоков ввода/вывода, а параметр –i переводит интерпретатор в интерактив-  
ный режим после выполнения сценария:  
c:\Python30> python –u main.py a b -c # Отключает буферизацию потоков вывода  
Python 2.6 поддерживает ряд дополнительных параметров, обеспечивающих   
совместимость с версией 3.0 (-3, -Q) и позволяющих определять случаи непо-  
следовательного использования символов табуляции для оформления отсту-  
пов, которые всегда проверяются в версии 3.0 (-t� смотрите главу 12). За до-  
полнительной информацией о доступных параметрах командной строки обра-  
щайтесь к руководствам или справочникам по языку Python. Или, еще лучше,   
спросите у самого интерпретатора – запустите такую команду:  
c:\Python30> python -?  
чтобы вывести справочную информацию, в которой описываются все поддер-  
живаемые параметры командной строки. Если вам приходится иметь дело со   
сложными командами запуска сценариев, обязательно познакомьтесь с моду-  
лями getopt и optparse из стандартной библиотеки, которые реализуют обработ-  
ку более сложных параметров командной строки.  
Дополнительная информация  
Комплект стандартных руководств на сегодняшний день включает ценные   
рекомендации по использованию языка Python на различных платформах.   
Комплект стандартных руководств доступен в операционной системе �indows   
в меню кнопки Пуск (Start), после установки Python (пункт Python Manuals (Руко-

1210   
Приложение A. Установка и настройка   
водства Python)) и в электронном виде на сайте http://www.python.org. Загляните   
в руководство, озаглавленное «Using Python», где вы найдете множество под-  
Using Python», где вы найдете множество под-  
 Python», где вы найдете множество под-  
Python», где вы найдете множество под-  
», где вы найдете множество под-  
сказок и рекомендаций, а также описание настроек и особенностей командной   
строки для различных платформ.  
Как обычно, Сеть – ваш лучший друг, особенно когда дело касается области,   
которая развивается значительно быстрее, чем успевают выходить новые из-  
дания книг, подобных этой. Учитывая широкую распространенность Python,   
велика вероятность, что ответы на вопросы по использованию языка, которые   
у вас могут возникнуть, вы сможете найти в Сети.

Приложение B.  
   
Решения упражнений  
Часть I. Введение  
Упражнения находятся в главе 3, в разделе «Упражнения к первой части».  
1. Взаимодействие. Предположим, что настройка Python выполнена пра-  
вильно, тогда сеанс взаимодействия с интерактивной оболочкой должен   
выглядеть примерно так, как показано ниже (вы можете запустить инте-  
рактивную оболочку любым способом, по своему усмотрению – в IDLE, из   
системной командной строки и так далее):  
% python  
...строки с информацией об авторских правах...  
>>> “Hello World!”  
‘Hello World!’  
>>> # Для выхода используйте Ctrl-D или Ctrl-Z, или закройте окно  
2. Программы. Содержимое файла (то есть, модуля) module1.py и сеанс взаи-  
модействия с командной оболочкой операционной системы должны выгля-  
деть, как показано ниже:  
print ‘Hello module world!’  
   
% python module1.py  
Hello module world!  
Вы можете запустить этот файл любым другим способом – щелчком мыши   
на ярлыке файла, с помощью пункта меню Run (Запустить) → Run Module (за-  
пустить модуль) и так далее.  
3. Модули. Следующий пример интерактивного сеанса иллюстрирует запуск   
модуля при его импортировании:  
% python  
>>> import module1  
Hello module world!  
>>>

1212   
Приложение B. Решения упражнений   
Не забывайте, чтобы запустить модуль еще раз в течение этого же сеанса   
работы в интерактивной оболочке, необходимо перезагрузить его. Просьба   
переместить файл в другой каталог и снова импортировать его содержит   
одну хитрость: если интерпретатор Python создал в первоначальном ката-  
логе файл module1.pyc, при импорте он будет использовать этот файл, даже   
если файл с исходным программным кодом (.py) будет перемещен в другой   
каталог, расположенный за пределами пути поиска модулей. Если интер-  
претатор имел доступ к файлу с исходным программным кодом, он автома-  
тически создаст файл .pyc, содержащий байт-код скомпилированной вер-  
сии модуля. Подробнее о модулях рассказывается в главе 3.  
4. Сценарии. Предположим, что ваша платформа поддерживает интерпрета-  
цию комбинации символов #!, тогда решение этого упражнения могло бы   
выглядеть, как показано ниже (на вашем компьютере в строке #! может по-  
требоваться указать другой путь):  
#!/usr/local/bin/python (или #!/usr/bin/env python)  
print(‘Hello module world!’)  
   
% chmod +x module1.py  
   
% module1.py  
Hello module world!  
5. Ошибки. Ниже показано, какие сообщения об ошибках будут получены по   
завершении этого упражнения. В действительности вы вызываете исклю-  
чения – по умолчанию при возникновении исключений интерпретатор за-  
вершает работу программы и выводит сообщение об ошибке вместе содер-  
жимым стека вызовов на экран. Информация из стека показывает, в каком   
месте программы произошло исключение. В седьмой части книги вы узнае-  
те, как перехватывать исключения с помощью инструкции try и как обра-  
батывать их. Вы также узнаете, что в состав Python входит полноценный   
отладчик исходного программного кода, обладающий возможностями ана-  
лиза ошибок после возникновения исключения. А пока обратите внимание,   
что в случае появления ошибок Python выводит вполне информативные со-  
общения (вместо того, чтобы просто аварийно завершать программу вообще   
без каких-либо сообщений):  
% python  
>>> 2 \*\* 500  
3273390607896141870013189696827599152216642046043064789483291368096133796404674554  
883270092325904157150886684127560071009217256545885393053328527589376  
>>>  
>>> 1 / 0  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
ZeroDivisionError: int division or modulo by zero  
>>>  
>>> spam  
Traceback (most recent call last):  
 File “<stdin>”, line 1, in <module>  
NameError: name ‘spam’ is not defined  
6. Прерывание программы. Если вы введете следующие инструкции:  
L = [1, 2]  
L.append(L)

Часть I. Введение   
1213  
тем самым вы создадите циклическую структуру данных. В версиях ниже,   
чем Python 1.5.1, интерпретатор еще не обладает возможностью определять   
случаи зацикливания в объектах, и он будет выводить бесконечный поток   
[1, 2, [1, 2, [1, 2, [1, 2, и так далее, пока вы не нажмете комбинацию кла-  
виш, выполняющих прерывание программы (которая, с технической точ-  
ки зрения, возбуждает исключение прерывания с клавиатуры и приводит   
к выводу сообщения по умолчанию). Начиная с версии Python 1.5.1, интер-  
претатор уже умеет определять случаи зацикливания и вместо бесконечной   
последовательности выведет [[...]].  
Причина зацикливания пока для вас неочевидна, чтобы ее понять, необхо-  
димы знания, которые вы получите во второй части книги. Однако в двух   
словах замечу, что операции присваивания в языке Python всегда создают   
ссылки на объекты, а не их копии. Вы можете представлять себе объекты   
как участки памяти, а ссылки на них – как неявное следование по указа-  
 – как неявное следование по указа-  
– как неявное следование по указа-  
телям. После выполнения первой инструкции, из тех, что представлены   
выше, имя L превратится в именованную ссылку на объект списка, состо-  
ящий из двух элементов, – указатель на область памяти. Списки в языке   
Python в действительности являются массивами ссылок на объекты, обла-  
 в действительности являются массивами ссылок на объекты, обла-  
дающими методом append, который изменяет сам массив, добавляя в конец   
ссылку на другой объект. В данном случае вызов метода append добавляет   
в конец списка L ссылку на сам список L, что приводит к зацикливанию,   
как показано на рис. B.1: указатель в конце списка, который указывает на   
начало списка.  
Помимо особого способа вывода, о котором вы узнаете в главе 6, цикличе-  
ские объекты должны также по-особому обрабатываться сборщиком мусо-  
ра, в противном случае занимаемое ими пространство памяти не сможет   
быть освобождено даже после того, как они перестанут использоваться.   
В некоторых программах, где выполняется обход произвольных объектов,   
вам, возможно, самим придется определять такие зацикливания и внима-  
тельно изучать действия программы, чтобы избежать зацикливания. Хоти-  
те верьте, хотите нет, но циклические структуры данных иногда могут быть   
полезны, несмотря на особенности их вывода.  
Имена  
Объекты  
L  
1  
2  
Рис. B.1. Циклический объект, созданный за счет добавления списка   
к самому себе. По умолчанию интерпретатор добавляет ссылку на ориги-  
нальный список, а не его копию

1214   
Приложение B. Решения упражнений   
Часть II. Типы и операции  
Упражнения находятся в главе 9, в разделе «Упражнения ко второй части».  
1. Основы. Ниже приводятся результаты, которые вы должны получить, с не-  
которыми комментариями к ним. Обратите внимание, что здесь иногда ис-  
пользуется символ ;, чтобы поместить несколько инструкций в одну строку   
(символ ; – это разделитель инструкций), и символ запятой, используемый   
для создания кортежей, которые выводятся в круглых скобках. Кроме того,   
имейте в виду, что результат деления с помощью оператора / отличается   
в версиях Python 2.6 и 3.0 (подробности приводятся в главе 5), а функция   
list(), обертывающая вызовы методов словаря, необходима, чтобы вывести   
результаты в версии 3.0, но не в 2.6 (подробности приводятся в главе 8):  
# Числа  
   
>>> 2 \*\* 16 # 2 в степени 16  
65536  
>>> 2 / 5, 2 / 5.0 # / усекает до целого числа в версии 2.6, но не в 3.0  
(0.40000000000000002, 0.40000000000000002)  
   
# Строки  
   
>>> “spam” + “eggs” # Конкатенация  
‘spameggs’  
>>> S = “ham”  
>>> “eggs “ + S  
‘eggs ham’  
>>> S \* 5 # Повторение  
‘hamhamhamhamham’  
>>> S[:0] # Пустой срез с первого элемента - [0:0]  
‘’  
   
>>> “green %s and %s” % (“eggs”, S) # Форматирование  
‘green eggs and ham’  
>>> ‘green {0} and {1}’.format(‘eggs’, S)  
‘green eggs and ham’  
   
# Кортежи  
   
>>> (‘x’,)[0] # Индексирование кортежа, состоящего из одного элемента  
‘x’  
>>> (‘x’, ‘y’)[1] # Индексирование кортежа, состоящего из двух элементов  
‘y’  
   
# Списки  
   
>>> L = [1,2,3] + [4,5,6] # Операции над списками  
>>> L, L[:], L[:0], L[-2], L[-2:]  
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [], 5, [5, 6])  
>>> ([1,2,3]+[4,5,6])[2:4]  
[3, 4]  
>>> [L[2], L[3]] # Извлечение по смещениям с сохранением в списке  
[3, 4]  
>>> L.reverse(); L # Метод: обратное упорядочивание элементов в списке  
[6, 5, 4, 3, 2, 1]  
>>> L.sort(); L # Метод: сортировка элементов в списке

Часть II. Типы и операции   
1215  
[1, 2, 3, 4, 5, 6]  
>>> L.index(4) # Метод: смещение первого вхождения 4 (поиск)  
3  
   
# Словари  
   
>>> {‘a’:1, ‘b’:2}[‘b’] # Извлечение элемента словаря по ключу  
2  
>>> D = {‘x’:1, ‘y’:2, ‘z’:3}  
>>> D[‘w’] = 0 # Создаст новый элемент словаря  
>>> D[‘x’] + D[‘w’]  
1  
>>> D[(1,2,3)] = 4 # Использование кортежа в качестве ключа (неизменяемый)  
   
>>> D  
{‘w’: 0, ‘z’: 3, ‘y’: 2, (1, 2, 3): 4, ‘x’: 1}  
   
>>> list(D.keys()), list(D.values()), (1,2,3) in D # Методы, проверка ключа  
([‘w’, ‘z’, ‘y’, (1, 2, 3), ‘x’], [0, 3, 2, 4, 1], True)  
   
# Пустые объекты  
   
>>> [[]], [“”,[],(),{},None] # Множество пустых объектов  
([[]], [‘’, [], (), {}, None])  
2. Индексирование  и  извлечение  среза. Если попытаться получить доступ   
к элементу, индекс которого выходит за пределы списка (например, L[4]),   
будет возбуждено исключение – интерпретатор всегда проверяет выход за   
пределы последовательностей.  
С другой стороны, операция извлечения среза с использованием индексов,   
выходящих за пределы последовательности (например, L[-1000:100]), будет   
выполнена без ошибок, потому что интерпретатор Python всегда корректи-  
Python всегда корректи-  
 всегда корректи-  
рует границы срезов, выходящие за пределы, так, чтобы они соответство-  
вали границам последовательности (пределы устанавливаются равными   
нулю и длине последовательности, если это необходимо).  
Извлечение последовательности в обратном порядке, когда нижняя грани-  
ца больше верхней (например, L[3:1]), в действительности работать не будет.   
Вы получите пустой срез ([ ]), потому что интерпретатор скорректирует   
границы среза так, чтобы нижняя граница оказалась меньше или равна   
верхней границе (например, выражение L[3:1] будет преобразовано в выра-  
жение L[3:3], пустая позиция со смещением 3). Срезы в языке Python всегда   
извлекаются слева направо, даже при использовании отрицательных ин-  
дексов (они сначала будут преобразованы в положительные индексы за счет   
добавления длины последовательности). Обратите внимание, что в Python   
2.3 появились срезы с тремя пределами: выражение L[3:1:-1] на самом деле   
извлекает срез справа налево:  
>>> L = [1, 2, 3, 4]  
>>> L[4]  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
IndexError: list index out of range  
(IndexError: выход индекса за пределы списка)  
>>> L[-1000:100]  
[1, 2, 3, 4]

1216   
Приложение B. Решения упражнений   
>>> L[3:1]  
[]  
>>> L  
[1, 2, 3, 4]  
>>> L[3:1] = [‘?’]  
>>> L  
[1, 2, 3, ‘?’, 4]  
3. Индексирование, извлечение среза и инструкция del. Сеанс взаимодействия   
с интерпретатором должен выглядеть примерно так, как показано ниже.   
Обратите внимание, что присваивание пустого списка по указанному сме-  
щению приведет к сохранению объекта пустого списка в этой позиции, но   
присваивание пустого списка срезу приведет к удалению этого среза. Опе-  
рация присваивания срезу ожидает получить другую последовательность,   
в противном случае будет получено сообщение о неверном типе операнда –   
она вставляет элементы, находящиеся внутри этой последовательности,   
а не саму последовательность:  
>>> L = [1,2,3,4]  
>>> L[2] = []  
>>> L  
[1, 2, [], 4]  
>>> L[2:3] = []  
>>> L  
[1, 2, 4]  
>>> del L[0]  
>>> L  
[2, 4]  
>>> del L[1:]  
>>> L  
[2]  
>>> L[1:2] = 1  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
TypeError: illegal argument type for built-in operation  
4. Кортежи. Значения X и Y поменяются местами. Когда слева и справа от опе-  
ратора присваивания (=) появляются кортежи, интерпретатор выполнит   
присваивание объектов справа объектам слева в соответствии с их позиция-  
ми. Это объяснение, пожалуй, самое простое для понимания, хотя целевые   
объекты слева и не являются настоящим кортежем, несмотря на то, что они   
именно так и выглядят, – это просто набор независимых имен, которым вы-  
полняется присваивание. Элементы справа – это кортеж, который распако-  
вывается в процессе выполнения операции присваивания (кортеж играет   
роль временного хранилища присваиваемых данных, необходимого для до-  
стижения эффекта обмена):  
>>> X = ‘spam’  
>>> Y = ‘eggs’  
>>> X, Y = Y, X  
>>> X  
‘eggs’  
>>> Y  
‘spam’  
5. Ключи словарей. В качестве ключа словаря может использоваться любой   
неизменяемый объект, включая целые числа, кортежи, стро ки и т. д. Это

Часть II. Типы и операции   
1217  
действительно словарь, несмотря на то, что некоторые из его ключей выгля-  
дят как целочисленные смещения. Допускает ся с одним и тем же словарем   
использовать ключи разных типов:  
>>> D = {}  
>>> D[1] = ‘a’  
>>> D[2] = ‘b’  
>>> D[(1, 2, 3)] = ‘c’  
>>> D  
{1: ‘a’, 2: ‘b’, (1, 2, 3): ‘c’}  
6. Индексирование словарей. Извлечение элемента по несуществую щему клю-  
чу (D[‘d’]) приводит к появлению ошибки. Присваивание по несуществую-  
щему ключу (D[‘d’]=’spam’) создает новый элемент словаря. С другой сторо-  
ны, выход индекса за границы списка тоже вызывает появление ошибки�   
ошибкой считается и присваивание по индексу, выходящему за границы   
списка. Имена переменных на поминают ключи словаря – прежде чем об-  
ратиться к переменным, им необходимо присвоить значение. Имена пере-  
менных, если пона добится, можно обрабатывать как ключи словарей (они   
становятся видимы в пространстве имен модуля или в пределах словарей):  
>>> D = {‘a’:1, ‘b’:2, ‘c’:3}  
>>> D[‘a’]  
1  
>>> D[‘d’]  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
KeyError: d  
>>> D[‘d’] = 4  
>>> D  
{‘b’: 2, ‘d’: 4, ‘a’: 1, ‘c’: 3}  
>>>  
>>> L = [0, 1]  
>>> L[2]  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
IndexError: list index out of range  
>>> L[2] = 3  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
IndexError: list assignment index out of range  
7. Общие операции. Ответы на вопросы:  
 •  
Оператор + не работает с объектами разных типов (например, строка +   
список, список + кортеж).  
 •  
Оператор + не работает со словарями, так как они не являются последо-  
вательностями.  
 •  
Метод append может применяться только к спискам, но не к строкам,   
а метод keys может применяться только к словарям. Метод append пред-  
полагает, что целевой объект является изменяемым, поскольку изменя-  
ется сам объект – строки относятся к неизменяемым типам.  
 •  
Операции извлечения среза и конкатенации всегда возвращают новый   
объект того же типа, что и объекты операнды.

1218   
Приложение B. Решения упражнений   
>>> “x” + 1  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
TypeError: illegal argument type for built-in operation  
>>>  
>>> {} + {}  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
TypeError: bad operand type(s) for +  
>>>  
>>> [].append(9)  
>>> “”.append(‘s’)  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
AttributeError: attribute-less object  
>>>  
>>> list({}.keys()) # Функция list необходима в 3.0  
[]  
>>> [].keys()  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
AttributeError: keys  
>>>  
>>> [][:]  
[]  
>>> “”[:]  
‘’  
8. Индексирование строк. Так как строки представляют собой кол лекции од-  
носимвольных строк, каждый раз, когда извлекается элемент строки, воз-  
вращается строка, из которой также допускает ся извлекать элементы по   
индексам. Выражение S[0][0][0][0][0] всего лишь извлекает первый символ   
снова и снова. Этот прием не применим к спискам (списки могут хранить   
объекты произволь ных типов), если список не содержит строки:  
>>> S = “spam”  
>>> S[0][0][0][0][0]  
‘s’  
>>> L = [‘s’, ‘p’]  
>>> L[0][0][0]  
‘s’  
9. Неизменяемые  типы. Правильными являются оба следующих ре шения.   
Присваивание по индексу недопустимо, потому что строки являются неиз-  
меняемыми объектами:  
>>> S = “spam”  
>>> S = S[0] + ‘l’ + S[2:]  
>>> S  
‘slam’  
>>> S = S[0] + ‘l’ + S[2] + S[3]  
>>> S  
‘slam’  
(Смотрите также в главе 36 описание типа bytearray, появившегося   
в Python 3.0, – это изменяемая последовательность малых целых чисел, ко-  
торая обрабатывается практически так же, как и строка.)

Часть III. Инструкции и синтаксис   
1219  
10. Вложенные структуры. Ниже приводится пример такой структуры:  
>>> me = {‘name’:(‘John’, ‘Q’, ‘Doe’), ‘age’:’?’, ‘job’:’engineer’}  
>>> me[‘job’]  
‘engineer’  
>>> me[‘name’][2]  
‘Doe’  
11. Файлы. Ниже приводится пример одного из способов создания файла и чте-  
ния информации из него в языке Python (ls – это команда UNIX, в �indows   
вместо нее следует использовать команду dir):  
# Файл: maker.py  
file = open(‘myfile.txt’, ‘w’)  
file.write(‘Hello file world!\n’) # Или: open().write()  
file.close() # закрывать файл не всегда обязательно  
   
# Файл: reader.py  
   
file = open(‘myfile.txt’) # ‘r’ - режим открытия по умолчанию  
print(file.read()) # Или print(open().read())  
   
% python maker.py  
% python reader.py  
Hello file world!  
   
% ls -l myfile.txt  
-rwxrwxrwa 1 0 0 19 Apr 13 16:33 myfile.txt  
Часть III. Инструкции и синтаксис  
Упражнения находятся в главе 15, в разделе «Упражнения к третьей части».  
1. Основы циклов. При выполнении этого упражнения вы получите примерно   
такой программный код:  
>>> S = ‘spam’  
>>> for c in S:  
... print ord(c)  
...  
115  
112  
97  
109  
   
>>> x = 0  
>>> for c in S: x += ord(c) # Или: x = x + ord(c)  
...  
>>> x  
433  
   
>>> x = []  
>>> for c in S: x.append(ord(c))  
...  
>>> x  
[115, 112, 97, 109]  
   
>>> list(map(ord, S)) # Функция list необходима в 3.0, но не в 2.6  
[115, 112, 97, 109]

1220   
Приложение B. Решения упражнений   
2. Символы обратного слеша. Пример выведет символ подачи звуко вого сиг-  
нала (\a) 50 раз – здесь предполагается, что ваш компьютер в состоянии вос-  
произвести его, поэтому при запуске этого фрагмен та не под управлением   
IDLE вы можете получить серию звуковых сигналов (или один длинный   
сигнал, если ваш компьютер обладает достаточно высоким быстродействи-  
ем). Эй! Я предупреждал вас!  
3. Сортировка  словарей. Ниже приводится один из возможных вари антов   
решения этого упражнения (вернитесь к главе 8 или 14, если что-то по-  
казалось вам непонятным). Не забывайте, что вам необходимо разделить   
вызовы методов keys и sort, как это сделано здесь, потому что метод sort   
возвращает объект None. Начиная с версии Python 2.2, появилась возмож-  
ность организовать итерации по ключам словаря без вызова метода keys (на-  
пример, for key in D:), но ключи не будут отсортированы этим программным   
кодом. В самых свежих версиях Python того же эффекта можно добиться   
с помощью встроенной функции sorted:  
>>> D = {‘a’:1, ‘b’:2, ‘c’:3, ‘d’:4, ‘e’:5, ‘f’:6, ‘g’:7}  
>>> D  
{‘f’: 6, ‘c’: 3, ‘a’: 1, ‘g’: 7, ‘e’: 5, ‘d’: 4, ‘b’: 2}  
>>>  
>>> keys = list(D.keys()) # Функция list необходима в 3.0, но не в 2.6  
>>> keys.sort()  
>>> for key in keys:  
... print(key, ‘=>’, D[key])  
...  
a => 1  
b => 2  
c => 3  
d => 4  
e => 5  
f => 6  
g => 7  
   
>>> for key in sorted(D): # Так лучше, в наиболее свежих версиях Pythons  
... print(key, ‘=>’, D[key])  
4. Программирование альтернативной логики. Ниже приводятся не сколько   
вариантов решения этого упражнения. Ваши решения мо гут несколько от-  
личаться – цель этого упражнения состоит в том, чтобы дать вам возмож-  
ность поэкспериментировать с созданием альтернативных ветвей исполне-  
ния программы, поэтому приветст вуются любые правильные решения:  
# a  
   
L = [1, 2, 4, 8, 16, 32, 64]  
X = 5  
   
i = 0  
while i < len(L):  
 if 2 \*\* X == L[i]:  
 print(‘at index’, i)  
 break  
 i += 1  
else:  
 print(X, ‘not found’)

Часть IV. Функции   
1221  
# b  
   
L = [1, 2, 4, 8, 16, 32, 64]  
X = 5  
   
for p in L:  
 if (2 \*\* X) == p:  
 print((2 \*\* X), ‘was found at’, L.index(p))  
 break  
else:  
 print(X, ‘not found’)  
   
# c  
   
L = [1, 2, 4, 8, 16, 32, 64]  
X = 5  
   
if (2 \*\* X) in L:  
 print((2 \*\* X), ‘was found at’, L.index(2 \*\* X))  
else:  
 print(X, ‘not found’)  
   
# d  
   
X = 5  
L = []  
for i in range(7): L.append(2 \*\* i)  
print(L)  
   
if (2 \*\* X) in L:  
 print((2 \*\* X), ‘was found at’, L.index(2 \*\* X))  
else:  
 print(X, ‘not found’)  
   
# f  
   
X = 5  
L = list(map(lambda x: 2\*\*x, range(7))) # или [2\*\*x for x in range(7)]  
print(L) # list() – чтобы вывести все результаты в 3.0  
   
if (2 \*\* X) in L:  
 print((2 \*\* X), ‘was found at’, L.index(2 \*\* X))  
else:  
 print(X, ‘not found’)  
Часть IV. Функции  
Упражнения находятся в главе 20, в разделе «Упражнения к четвертой части».  
1. Основы. В этом упражнении нет ничего сложного, но обратите внимание на   
то, что инструкция print (а, следовательно, и ваша функция) с технической   
точки зрения, является полиморфической операцией, которая правильно   
интерпретирует типы всех объектов:  
% python  
>>> def func(x): print(x)  
...  
>>> func(“spam”)  
spam

1222   
Приложение B. Решения упражнений   
>>> func(42)  
42  
>>> func([1, 2, 3])  
[1, 2, 3]  
>>> func({‘food’: ‘spam’})  
{‘food’: ‘spam’}  
2. Аргументы. Ниже приводится пример решения. Не забывайте, что для   
вывода результата необходимо использовать функцию print, потому что   
выполнение программного кода в модуле – это совсем не то же самое, что   
выполнение программного кода, который вводится в интерактивной обо-  
лочке, – интерпретатор не выводит автомати чески результаты выражений,   
вычисляемых в файлах модулей:  
def adder(x, y):  
 return x + y  
   
print(adder(2, 3))  
print(adder(‘spam’, ‘eggs’))  
print(adder([‘a’, ‘b’], [‘c’, ‘d’]))  
   
% python mod.py  
5  
spameggs  
[‘a’, ‘b’, ‘c’, ‘d’]  
3. Переменное  число  аргументов. В следующем файле adder.py пока заны   
два варианта реализации функции adder. Самое сложное здесь заключа-  
ется в инициализации значения суммы пустым значением в зависимости   
от типа передаваемых аргументов. Первое решение выясняет, является ли   
первый аргумент целым числом, и извлека ет пустой срез первого аргумен-  
та (предполагается, что он является последовательностью), если он не яв-  
ляется целым числом. Во вто ром решении в качестве начального значения   
используется первый аргумент, а затем выполняется сканирование аргу-  
ментов, начиная со второго, почти как в одном из вариантов функции min,   
показан ной в главе 18.  
Второе решение является более предпочтительным. Но в обоих ва риантах   
предполагается, что все аргументы принадлежат к одному и тому же типу,   
а, кроме того, что функция не бу дет работать со словарями (как мы видели   
во второй части книги, оператор + не работает с операндами разных типов   
и со словарями). Вы можете добавить проверку типа аргументов и преду-  
смотреть специальный программный код для обработки словарей, за что   
вам причитаются дополнительные баллы:  
def adder1(\*args):  
 print(‘adder1’, end=’ ‘)  
 if type(args[0]) == type(0): # Целое число?  
 sum = 0 # Инициализировать нулем  
 else: # иначе - последовательность:  
 sum = args[0][:0] # Использовать пустой срез первого аргумента  
 for arg in args:  
 sum = sum + arg  
 return sum  
   
def adder2(\*args):  
 print(‘adder2’, end=’ ‘)

Часть IV. Функции   
1223  
 sum = args[0] # Инициализировать значением первого аргумента  
 for next in args[1:]:  
 sum += next # Прибавить аргументы 2..N  
 return sum  
   
for func in (adder1, adder2):  
 print(func(2, 3, 4))  
 print(func(‘spam’, ‘eggs’, ‘toast’))  
 print(func([‘a’, ‘b’], [‘c’, ‘d’], [‘e’, ‘f’]))  
   
% python adders.py  
adder1 9  
adder1 spameggstoast  
adder1 [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]  
adder2 9  
adder2 spameggstoast  
adder2 [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]  
4. Именованные аргументы. Ниже приводится мое решение первой и второй   
частей этого упражнения (файл mod.py). Для обхода именованных аргумен-  
тов в заголовке функции следует исполь зовать форму списка аргументов   
\*\*args и цикл в теле функции (на пример, for x in args.keys(): и использовать   
args[x]) или использо вать args.values(), чтобы реализовать сложение значе-  
ний позиционных аргументов в списке \*args:  
def adder(good=1, bad=2, ugly=3):  
 return good + bad + ugly  
   
print(adder())  
print(adder(5))  
print(adder(5, 6))  
print(adder(5, 6, 7))  
print(adder(ugly=7, good=6, bad=5))  
   
% python mod.py  
6  
10  
14  
18  
18  
   
# Вторая часть решения  
   
def adder1(\*args): # Сумма значений произвольного количества   
 tot = args[0] # позиционных аргументов  
 for arg in args[1:]:  
 tot += arg  
 return tot  
   
def adder2(\*\*args): # Сумма значений произвольного количества именованных арг.  
 argskeys = list(args.keys()) # функция list необходима в 3.0!  
 tot = args[argskeys[0]]  
 for key in argskeys[1:]:  
 tot += args[key]  
 return tot  
   
def adder3(\*\*args): # То же самое, но преобразует в список значений  
 args = list(args.values()) # list необходима для индексирования в 3.0!  
 tot = args[0]

1224   
Приложение B. Решения упражнений   
 for arg in args[1:]:  
 tot += arg  
 return tot  
   
def adder4(\*\*args): # То же самое, но использует версию с позиционными арг.  
 return adder1(\*args.values())  
   
print(adder1(1, 2, 3), adder1(‘aa’, ‘bb’, ‘cc’))  
print(adder2(a=1, b=2, c=3), adder2(a=’aa’, b=’bb’, c=’cc’))  
print(adder3(a=1, b=2, c=3), adder3(a=’aa’, b=’bb’, c=’cc’))  
print(adder4(a=1, b=2, c=3), adder4(a=’aa’, b=’bb’, c=’cc’))  
5. (и 6) Ниже приводится мое решение упражнений 5 и 6 (файл dicts.py). Од-  
нако это решение пригодно только в качестве упражне ния, потому что   
в Python 1.5 у словарей появились методы D.copy() и D1.update(D2), которые   
реализуют операции копирования и сложе ния (слияния) словарей. (За до-  
полнительной информацией обращай тесь к справочному руководству по   
библиотеке Python или к книге «Python Pocket Reference», выпущенной из-  
дательством O’Reilly.) Операция извлечения среза X[:] не применима к сло-  
варям, потому что они не являются последовательностями (подробности   
в главе 8). Кроме того, не забывайте, что в случае присваивания (e = d) про-  
сто создается вторая ссылка на один и тот же объект словаря – измене ния   
в словаре d будут приводить к изменениям в словаре e:  
def copyDict(old):  
 new = {}  
 for key in old.keys():  
 new[key] = old[key]  
 return new  
   
def addDict(d1, d2):  
 new = {}  
 for key in d1.keys():  
 new[key] = d1[key]  
 for key in d2.keys():  
 new[key] = d2[key]  
 return new  
   
% python  
>>> from dicts import \*  
>>> d = {1: 1, 2: 2}  
>>> e = copyDict(d)  
>>> d[2] = ‘?’  
>>> d  
{1: 1, 2: ‘?’}  
>>> e  
{1: 1, 2: 2}  
   
>>> x = {1: 1}  
>>> y = {2: 2}  
>>> z = addDict(x, y)  
>>> z  
{1: 1, 2: 2}  
6. Смотрите решение № 5.  
7. Дополнительные примеры на сопоставление аргументов. Ниже приводит-  
ся пример сеанса работы с интерактивной оболочкой, который вы должны

Часть IV. Функции   
1225  
получить в ходе выполнения этого упражнения, с некоторыми коммента-  
риями, поясняющими порядок сопоставления аргументов:  
def f1(a, b): print(a, b) # Обычные аргументы  
   
def f2(a, \*b): print(a, b) # Сопоставление выполняется по позиции   
   
def f3(a, \*\*b): print(a, b) # Переменное число именованных аргументов  
   
def f4(a, \*b, \*\*c): print(a, b, c) # Смешанный режим  
   
def f5(a, b=2, c=3): print(a, b, c)# Аргументы со значениями по умолчанию  
   
def f6(a, b=2, \*c): print(a, b, c) # Аргументы по умолчанию и переменное число   
 # позиционных аргументов  
   
% python  
>>> f1(1, 2) # Сопоставление по позиции (важен порядок)  
1 2  
>>> f1(b=2, a=1) # Сопоставление по имени (порядок не важен)  
1 2  
   
>>> f2(1, 2, 3) # Дополнительные позиционные аргументы   
1 (2, 3) # объединяются в кортеж  
   
>>> f3(1, x=2, y=3) # Дополнительные именованные аргументы   
1 {‘x’: 2, ‘y’: 3} # объединяются в словарь  
   
>>> f4(1, 2, 3, x=2, y=3) # Дополнительные аргументы обоих типов  
1 (2, 3) {‘x’: 2, ‘y’: 3}  
   
>>> f5(1) # Два последних аргумента получают   
1 2 3 # значения по умолчанию  
>>> f5(1, 4) # Используется только одно значение   
1 4 3 # по умолчанию  
   
>>> f6(1) # Один аргумент: соответствует имени “a”  
1 2 ()  
>>> f6(1, 3, 4) # Дополнительный позиционный аргумент  
1 3 (4,) # собирается в кортеж  
8. Снова простые числа. Ниже приводится пример нахождения про стых чи-  
сел, оформленный в виде функции в модуле (файл primes.py), которую мож-  
но вызывать несколько раз. Я добавил проверку if, чтобы отсечь отрицатель-  
ные числа, 0 и 1. Кроме того, я заменил оператор / на //, чтобы это решение   
правильно работало в Python 3.0, где было внесено изменение в действие опе-  
ратора /, о чем рассказывалось в главе 5, и чтобы добавить поддержку веще-  
ственных чисел (раскомментируйте инструкцию from и замените оператор //   
на /, чтобы увидеть разницу между версиями Python 2.6 и 3.0):  
#from \_\_future\_\_ import division  
   
def prime(y):  
 if y <= 1: # У некоторых может быть y > 1  
 print(y, ‘not prime’)  
 else:  
 x = y // 2 # В версии 3.0 оператор / терпит неудачу  
 while x > 1:  
 if y % x == 0: # нет остатка?  
 print(y, ‘has factor’, x)

1226   
Приложение B. Решения упражнений   
 break # Обойти ветку else  
 x -= 1  
 else:  
 print(y, ‘is prime’)  
   
prime(13); prime(13.0)  
prime(15); prime(15.0)  
prime(3); prime(2)  
prime(1); prime(-3)  
Далее приводится пример работы модуля. Оператор // позволяет работать   
с вещественными числами тоже, хотя этого быть не должно.  
% python primes.py  
13 is prime  
13.0 is prime  
15 has factor 5  
15.0 has factor 5.0  
3 is prime  
2 is prime  
1 not prime  
-3 not prime  
Эта функция все еще не слишком пригодна к многократному ис-  
пользованию – вместо вывода результата она могла бы возвращать значе-  
ния, но для экспериментов вполне достаточно и такой реали зации. Кроме   
того, она не строго соответствует математическому определению простых   
чисел, которыми могут быть только целые числа, и не обладает достаточ-  
ной эффективностью. Эти улучшения я оставляю для самостоятельной реа-  
лизации читателям, склонным к математике. (Подсказка: цикл for через   
range(y, 1, -1) будет вы полняться немного быстрее, чем цикл while, но самым   
узким местом здесь является сам алгоритм.) Для измерения производи-  
тельности альтернативных реализаций воспользуйтесь встроенным моду-  
лем time и отрывком кода, подобным следующей универсаль ной функции   
timer (за дополнительной информацией обращайтесь к справочному руко-  
водству по библиотеке):  
def timer(reps, func, \*args):  
 import time  
 start = time.clock()  
 for i in range(reps):  
 apply(func, args)  
 return time.clock() - start  
9. Генераторы списков. Ниже приводится вариант программного кода, кото-  
рый должен получиться у вас, – у меня есть свои предпочтения, но я о них   
умолчу:  
>>> values = [2, 4, 9, 16, 25]  
>>> import math  
   
>>> res = []  
>>> for x in values: res.append(math.sqrt(x))  
...  
   
>>> res  
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

Часть IV. Функции   
1227  
>>> list(map(math.sqrt, values))  
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]  
   
>>> [math.sqrt(x) for x in values]  
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]  
10. Хронометраж. Ниже приводится моя версия программного кода, который   
выполняет хронометраж трех способов извлечения квадратного корня,   
а также результаты, полученные в Python 2.6 и 3.0. Последним выводится   
результат каждой функции, чтобы убедиться, что все три решают одну и ту   
же задачу:  
# Файл mytimer.py (2.6 and 3.0)  
...то же, что и в главе 20...  
   
# Файл timesqrt.py  
import sys, mytimer  
reps = 10000  
repslist = range(reps) # Вызов функции range вынесен за пределы цикла в 2.6  
   
from math import sqrt # Не math.sqrt: чтобы не тратить время   
def mathMod(): # на доступ к атрибуту  
 for i in repslist:  
 res = sqrt(i)  
 return res  
   
def powCall():  
 for i in repslist:  
 res = pow(i, .5)  
 return res  
   
def powExpr():  
 for i in repslist:  
 res = i \*\* .5  
 return res  
   
print(sys.version)  
for tester in (mytimer.timer, mytimer.best):  
 print(‘<%s>’ % tester.\_\_name\_\_)  
 for test in (mathMod, powCall, powExpr):  
 elapsed, result = tester(test)  
 print (‘-’\*35)  
 print (‘%s: %.5f => %s’ %  
 (test.\_\_name\_\_, elapsed, result))  
Ниже приводятся результаты тестирования в Python 3.0 и 2.6. В обоих слу-  
Python 3.0 и 2.6. В обоих слу-  
 3.0 и 2.6. В обоих слу-  
чаях модуль math оказывается быстрее оператора \*\*, который, в свою оче-  
редь, быстрее функции pow. Однако вам следует протестировать свой про-  
граммный код на своей машине и со своей версией Python. Кроме того, об-  
Python. Кроме того, об-  
. Кроме того, об-  
ратите внимание, что Python 3.0 по результатам тестирования оказывается   
почти в два раза медленнее, чем Python 2.6, – версия 3.1 или более поздние   
могут иметь более высокую производительность (выполните эти тесты, что-  
бы лично убедиться – так ли это):  
c:\misc> c:\python30\python timesqrt.py  
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]  
<timer>  
-----------------------------------

1228   
Приложение B. Решения упражнений   
mathMod: 5.33906 => 99.994999875  
-----------------------------------  
powCall: 7.29689 => 99.994999875  
-----------------------------------  
powExpr: 5.95770 => 99.994999875  
<best>  
-----------------------------------  
mathMod: 0.00497 => 99.994999875  
-----------------------------------  
powCall: 0.00671 => 99.994999875  
-----------------------------------  
powExpr: 0.00540 => 99.994999875  
c:\misc> c:\python26\python timesqrt.py  
2.6.1 (r261:67517, Dec 4 2008, 16:51:00) [MSC v.1500 32 bit (Intel)]  
<timer>  
-----------------------------------  
mathMod: 2.61226 => 99.994999875  
-----------------------------------  
powCall: 4.33705 => 99.994999875  
-----------------------------------  
powExpr: 3.12502 => 99.994999875  
<best>  
-----------------------------------  
mathMod: 0.00236 => 99.994999875  
-----------------------------------  
powCall: 0.00402 => 99.994999875  
-----------------------------------  
powExpr: 0.00287 => 99.994999875  
Чтобы выполнить измерение производительности генераторов  словарей   
и эквивалентных им циклов for в Python 3.0 в интерактивном режиме, вы-  
Python 3.0 в интерактивном режиме, вы-  
 3.0 в интерактивном режиме, вы-  
полните операции, как показано ниже. По результатам тестирования эти   
два способа показывают почти одинаковую производительность, по край-  
ней мере в Python 3.0, – в отличие от генераторов списков, цикл for оказы-  
вается немножко быстрее, чем генератор словарей (хотя, надо заметить, это   
различие является совсем незначительным – в конечном итоге цикл позво-  
ляет сэкономить всего полсекунды при создании 50 словарей, по 1 000 000   
элементов в каждом). Повторюсь еще раз: прежде чем принимать эти ре-  
зультаты в качестве меры оценки, вы должны сами выполнить тестирова-  
ние на своем компьютере и со своей версией Python:  
c:\misc> c:\python30\python  
>>>  
>>> def dictcomp(I):  
... return {i: i for i in range(I)}  
...  
>>> def dictloop(I):  
... new = {}  
... for i in range(I): new[i] = i  
... return new  
...  
>>> dictcomp(10)  
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}  
>>> dictloop(10)  
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}

Часть V. Модули   
1229  
>>>  
>>> from mytimer import best, timer  
>>> best(dictcomp, 10000)[0] # словарь из 10 000 элементов  
0.0013519874732672577  
>>> best(dictloop, 10000)[0]  
0.001132965223233029  
>>>  
>>> best(dictcomp, 100000)[0] # 100 000 элементов: в 10 раз медленнее  
0.01816089754424155  
>>> best(dictloop, 100000)[0]  
0.01643484018219965  
>>>  
>>> best(dictcomp, 1000000)[0] # 1 000 000 элементов:   
0.18685105229855026 # еще в 10 раз медленнее  
>>> best(dictloop, 1000000)[0] # Время создания одного словаря  
0.1769041177020938  
>>>  
>>> timer(dictcomp, 1000000, \_reps=50)[0] # 1 000 000 элементов  
10.692516087938543  
>>> timer(dictloop, 1000000, \_reps=50)[0] # Время создания 50 словарей  
10.197276050447755  
Часть V. Модули  
Упражнения находятся в главе 24, в разделе «Упражнения к пятой части».  
1. Основы  импортирования. Решение этого упражнения выглядит го раздо   
проще, чем можно было бы подумать. Когда вы закончите, у вас должен по-  
лучиться файл (mymod.py) и сеанс взаимодействия с интерактивной оболоч-  
кой, как показано ниже. Не забывайте, что интерпретатор Python может   
читать содержимое файла целиком в список строк, а получить длину каж-  
дой строки в списке можно с помощью встроенной функции len:  
def countLines(name):  
 file = open(name)  
 return len(file.readlines())  
   
def countChars(name):  
 return len(open(name).read())  
   
def test(name): # Или передать объект файла  
 return countLines(name), countChars(name) # Или вернуть словарь  
   
% python  
>>> import mymod  
>>> mymod.test(‘mymod.py’)  
(10, 291)  
Обратите внимание, что эти функции читают файл в память цели ком и по-  
тому не в состоянии работать с патологически большими файлами, которые   
не смогут уместиться в памяти компьютера. Что бы обеспечить более вы-  
сокую устойчивость, можно организовать построчное чтение содержимого   
файла с помощью итератора и на капливать длины строк в ходе итераций:  
def countLines(name):  
 tot = 0  
 for line in open(name): tot += 1

1230   
Приложение B. Решения упражнений   
 return tot  
   
def countChars(name):  
 tot = 0  
 for line in open(name): tot += len(line)  
 return tot  
В операционной системе UNIX полученный результат можно про верить   
с помощью команды wc. В �indows можно щелкнуть на фай ле правой кноп-  
кой мыши и посмотреть его свойства. Обратите вни мание, что результат,   
возвращаемый сценарием, может несколько отличаться от того, что сооб-  
щает �indows, – с целью обеспечения переносимости интерпретатор Python   
преобразует пары символов \r\n, отмечающих конец каждой строки, в один   
символ \n, в резуль тате чего теряется по одному байту (символу) на каждую   
строку. Чтобы ре зультат сценария в точности соответствовал тому, что сооб-  
щает �indows, файл необходимо открыть в режиме двоичного доступа (‘rb’)   
или прибавлять к общему результату число строк.  
Чтобы реализовать часть упражнения «для честолюбивых» (пере давать   
функциям объект файла, чтобы открывать его приходилось всего один раз),   
вам, скорее всего, придется использовать метод seek объекта файла. Мы не   
рассматривали этот метод в книге, но он ра ботает точно так же, как функ-  
ция fseek в языке C (которая, собст венно, и вызывается внутренней реали-  
зацией метода): метод seek переустанавливает текущую позицию в файле   
в указанное смеще ние. После вызова метода seek последующие операции   
ввода/выво да будут выполняться относительно новой позиции. Чтобы пере-  
меститься в начало файла, не закрывая и не открывая его повторно, можно   
вызвать метод file.seek(0)� все вызовы метода read выполня ют чтение из те-  
кущей позиции в файле, поэтому, чтобы начать по вторное чтение, необхо-  
димо переместить текущую позицию в нача ло файла. Ниже показано, как   
может выглядеть такая реализация:  
def countLines(file):  
 file.seek(0) # Переместиться в начало файла  
 return len(file.readlines())  
   
def countChars(file):  
 file.seek(0) # То же самое(переместиться в начало)  
 return len(file.read())  
   
def test(name):  
 file = open(name) # Передать объект файла  
 return countLines(file), countChars(file) # Открыть файл один раз  
   
>>> import mymod2  
>>> mymod2.test(“mymod2.py”)  
(11, 392)  
2. from/from \*. Ниже приводится решение в части использования инструкции   
вида from \*. Чтобы выполнить вторую часть упражнения, замените \* именем   
countChars:  
% python  
>>> from mymod import \*  
>>> countChars(“mymod.py”)  
291

Часть V. Модули   
1231  
3. \_\_main\_\_. Если вы не допустили ошибок в модуле, он сможет работать в лю-  
бом из режимов (в режиме самостоятельной программы или в режиме им-  
портируемого модуля):  
def countLines(name):  
 file = open(name)  
 return len(file.readlines())  
   
def countChars(name):  
 return len(open(name).read())  
   
def test(name): # Или передать объект файла  
 return countLines(name), countChars(name) # Или вернуть словарь  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 print(test(‘mymod.py’))  
   
% python mymod.py  
(13, 346)  
Это именно тот случай, когда я предпочел бы реализовать передачу имени   
исследуемого файла через параметры командной строки или ввод его поль-  
зователем по запросу, а не определять его жестко в тексте сценария (смотри-  
те описание sys.argv в главе 24 и описание функции input в главе 10):  
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 print(test(input(‘Enter file name:’))  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 import sys  
 print(test(sys.argv[1]))  
4. Вложенное  импортирование. Ниже представлено мое решение (файл   
myclient.py):  
from mymod import countLines, countChars  
print(countLines(‘mymod.py’), countChars(‘mymod.py’))  
   
% python myclient.py  
13 346  
Что касается остальной части этого упражнения, функции модуля mymod до-  
ступны (то есть были импортированы) на верхнем уровне мо дуля myclient,   
потому что инструкция from просто присваивает их именам в импортирую-  
щем модуле (выглядит так, как если бы опре деления функций def находи-  
лись в модуле myclient). Например, в сле дующем файле приводится альтер-  
нативный вариант реализации:  
import myclient  
myclient.countLines(...)  
   
from myclient import countChars  
countChars(...)  
Если бы в модуле myclient вместо инструкции from использовалась инструк-  
ция import, вам пришлось бы использовать полные имена функций из моду-  
ля mymod:  
import myclient  
myclient.mymod.countLines(...)

1232   
Приложение B. Решения упражнений   
from myclient import mymod  
mymod.countChars(...)  
Вообще говоря, можно определить модуль-коллектор, который будет им-  
портировать все имена из других модулей, чтобы сделать их дос тупными   
в виде единого модуля. Если воспользоваться следующим программным   
кодом, можно покончить с тремя разными копиями имени somename (mod1.  
somename, collector.somename и \_\_main\_\_.somename) – все три имени изначально   
будут ссылаться на один и тот же объект це лого числа, а в интерактивной   
оболочке будет существовать только одно имя somename:  
# Файл: mod1.py  
somename = 42  
   
# Файл: collector.py  
from mod1 import \* # Собирает множество имен в один модуль  
from mod2 import \* # from выполняет присваивание именам в этом модуле  
from mod3 import \*  
   
>>> from collector import somename  
5. Импорт  пакетов. Для этого упражнения я поместил файл mymod.py из   
упражнения 3 в каталог пакета. Далее описываются мои действия по на-  
стройке каталога и необходимого файла \_\_init\_\_.py в консоли �indows – вы   
должны учитывать особенности своей платформы (например, использовать   
команды mv и vi вместо move и edit). Эти действия можно выполнять в любом   
произвольном каталоге (про сто так вышло, что я запускал команды, нахо-  
дясь в каталоге, куда был установлен Python), частично эти действия можно   
выполнить в проводнике файловой системы с графическим интерфейсом.  
По окончании я получил подкаталог mypkg, содержащий файлы \_\_init\_\_.  
py и mymod.py. В каталоге mypkg обязательно должен нахо диться файл \_\_  
init\_\_.py, но его наличие в родительском каталоге не обязательно. Каталог   
mypkg находится в домашнем каталоге, в пути поиска модулей. Обратите   
внимание, что инструкция print в ини циализационном файле выполняется   
только при первой операции импорта:  
C:\python30> mkdir mypkg  
C:\Python30> move mymod.py mypkg\mymod.py  
C:\Python30> edit mypkg\\_\_init\_\_.py  
...добавление инструкции print...  
C:\Python30> python  
>>> import mypkg.mymod  
initializing mypkg  
>>> mypkg.mymod.countLines(‘mypkg\mymod.py’)  
13  
>>> from mypkg.mymod import countChars  
>>> countChars(‘mypkg\mymod.py’)  
346  
6. Повторная загрузка. В этом упражнении вам просто предлагается поэкспе-  
риментировать с возможностью повторной загрузки модуля changer.py из   
примера в книге, поэтому мне нечего показать здесь.  
7. Циклический импорт. Суть в том, что когда первым импортируется модуль   
recur2, ситуация рекурсивного импорта возникает в инструкции import   
в модуле recur1, а не в инструкции from в модуле recur2.

Часть VI. Классы и ООП   
1233  
Если говорить более подробно, все происходит следующим образом: когда   
модуль recur2 импортируется первым, в результате рекурсив ного импорта   
модуля recur2 из модуля recur1 модуль recur2 извлека ется целиком. К мо-  
менту, когда модуль recur2 импортируется моду лем recur1, его простран-  
ство имен еще не заполнено, но так как им портирование выполняется ин-  
струкцией import, а не инструкцией from, никаких проблем не возникает:   
интерпретатор отыскивает и возвращает уже созданный объект модуля   
recur2 и продолжает выполнять оставшуюся часть модуля recur1 без сбо-  
ев. Когда импорт в модуле recur2 возобновляется, вторая инструкция from   
обнаружи вает в модуле recur1 имя Y (модуль был выполнен полностью), по-  
этому в такой ситуации не возникает ошибок. Запуск файла как от дельного   
сценария – это не то же самое, что импортирование его в виде модуля: это   
равноценно интерактивному выполнению первой инструкции import или   
from в сценарии. Например, запуск модуля recur1 как сценария равносилен   
предварительному импортирова нию модуля recur2 в интерактивной обо-  
лочке, поскольку recur2 – это первый модуль, импортируемый в recur1.  
Часть VI. Классы и ООП  
Упражнения находятся в главе 31, в разделе «Упражнения к шестой части».  
1. Наследование. Ниже приводится решение этого упражнения (файл adder.  
py) вместе с несколькими примерами действий в интерактив ной оболочке.   
Метод перегрузки оператора \_\_add\_\_ присутствует только в суперклассе   
и вызывает конкретные методы add, опреде ляемые подклассами:  
class Adder:  
 def add(self, x, y):  
 print(‘not implemented!’)  
 def \_\_init\_\_(self, start=[]):  
 self.data = start  
 def \_\_add\_\_(self, other): # Или в подклассах?  
 return self.add(self.data, other) # Или возвращать тип?  
   
class ListAdder(Adder):  
 def add(self, x, y):  
 return x + y  
   
class DictAdder(Adder):  
 def add(self, x, y):  
 new = {}  
 for k in x.keys(): new[k] = x[k]  
 for k in y.keys(): new[k] = y[k]  
 return new  
   
% python  
>>> from adder import \*  
>>> x = Adder()  
>>> x.add(1, 2)  
not implemented!  
>>> x = ListAdder()  
>>> x.add([1], [2])  
[1, 2]  
>>> x = DictAdder()  
>>> x.add({1:1}, {2:2})

1234   
Приложение B. Решения упражнений   
{1: 1, 2: 2}  
>>> x = Adder([1])  
>>> x + [2]  
not implemented!  
>>>  
>>> x = ListAdder([1])  
>>> x + [2]  
[1, 2]  
>>> [2] + x  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
TypeError: \_\_add\_\_ nor \_\_radd\_\_ defined for these operands  
Обратите внимание, что в последнем тесте возникла ошибка при по пытке   
использовать экземпляр класса справа от оператора +. Если вы захотите   
исправить эту ошибку, реализуйте метод \_\_radd\_\_, как описывается в гла-  
ве 29, в разделе «Перегрузка операторов».  
Если вы начнете сохранять текущее значение в экземпляре, вы мо жете   
также переписать метод add, чтобы он принимал единствен ный аргумент,   
в духе других примеров из шестой части:  
class Adder:  
 def \_\_init\_\_(self, start=[]):  
 self.data = start  
 def \_\_add\_\_(self, other): # Передается единственный аргумент  
 return self.add(other) # Операнд слева хранится в self  
 def add(self, y):  
 print(‘not implemented!’)  
   
class ListAdder(Adder):  
 def add(self, y):  
 return self.data + y  
   
class DictAdder(Adder):  
 def add(self, y):  
 pass # Измените, чтобы использовать self.data вместо x  
   
x = ListAdder([1, 2 ,3])  
y = x + [4, 5, 6]  
print(y) # Выведет [1, 2, 3, 4, 5, 6]  
Поскольку значения присоединяются к самим объектам, а не пере даются   
в виде аргументов, эта версия определенно является более объектно-  
ориентированной. И как только вы доберетесь до этого мо мента, вы навер-  
няка обнаружите, что вообще можно избавиться от методов add и просто   
определить методы \_\_add\_\_ в двух подклассах.  
2. Перегрузка операторов. В программном коде решения (файл myl ist.py) ис-  
пользуется несколько методов перегрузки операторов, о ко торых в книге   
рассказывается не слишком много, однако в них нет ничего сложного. Опе-  
рация копирования начального значения в конструкторе имеет большое   
значение, потому что оно может быть представлено изменяемым объектом –   
едва ли есть разумная причи на, чтобы изменять или обладать ссылкой на   
объект, который, вполне возможно, используется где-то за пределами клас-  
са. Метод \_\_getattr\_\_ перехватывает попытки обращения к обернутому спи-  
ску. Подсказки, которые помогут упростить реализацию, вы найде те в раз-  
деле «Расширение типов наследованием» в главе 31:

Часть VI. Классы и ООП   
1235  
class MyList:  
 def \_\_init\_\_(self, start):  
 #self.wrapped = start[:] # Скопировать start: без побочных эффектов  
 self.wrapped = [] # Убедиться, что это список  
 for x in start: self.wrapped.append(x)  
 def \_\_add\_\_(self, other):  
 return MyList(self.wrapped + other)  
 def \_\_mul\_\_(self, time):  
 return MyList(self.wrapped \* time)  
 def \_\_getitem\_\_(self, offset):  
 return self.wrapped[offset]  
 def \_\_len\_\_(self):  
 return len(self.wrapped)  
 def \_\_getslice\_\_(self, low, high):  
 return MyList(self.wrapped[low:high])  
 def append(self, node):  
 self.wrapped.append(node)  
 def \_\_getattr\_\_(self, name): # Другие члены: sort/reverse/и так далее  
 return getattr(self.wrapped, name)  
 def \_\_repr\_\_(self):  
 return repr(self.wrapped)  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 x = MyList(‘spam’)  
 print(x)  
 print(x[2])  
 print(x[1:])  
 print(x + [‘eggs’])  
 print(x \* 3)  
 x.append(‘a’)  
 x.sort()  
 for c in x: print(c, end=’ ‘)  
   
% python mylist.py  
[‘s’, ‘p’, ‘a’, ‘m’]  
a  
[‘p’, ‘a’, ‘m’]  
[‘s’, ‘p’, ‘a’, ‘m’, ‘eggs’]  
[‘s’, ‘p’, ‘a’, ‘m’, ‘s’, ‘p’, ‘a’, ‘m’, ‘s’, ‘p’, ‘a’, ‘m’]  
a a m p s  
Обратите внимание, насколько важно копировать начальное значе ние, вы-  
полняя добавление в конец вместо извлечения среза, потому что в против-  
ном случае может получиться объект, не являющийся списком и потому   
не обладающий методами списка, такими как ap pend (например, операция   
извлечения среза строки возвращает дру гую строку, но не список). Если бы   
начальное значение было объек том типа MyList, можно было бы выполнить   
копирование с помощью операции извлечения среза, потому что этот класс   
перегружает эту операцию и обеспечивает ожидаемый интерфейс списков.   
Однако для других объектов, таких как строки, следует избегать использо-  
вания операции извлечения среза. Кроме того, следует заметить, что на   
сегодняшний день тип множества стал встроенным типом в языке Python,   
поэтому цель данного упражнения в значительной степени заключается   
в том, чтобы просто дать возможность попрак тиковаться (подробнее о мно-  
жествах рассказывается в главе 5).

1236   
Приложение B. Решения упражнений   
3. Подклассы. Мое решение (файл mysub.py) приводится ниже. Ваше решение   
должно быть похожим:  
from mylist import MyList  
   
class MyListSub(MyList):  
 calls = 0 # Используется всеми экземплярами  
   
 def \_\_init\_\_(self, start):  
 self.adds = 0 # Свой для каждого экземпляра  
 MyList.\_\_init\_\_(self, start)  
   
 def \_\_add\_\_(self, other):  
 MyListSub.calls += 1 # Счетчик, единый для класса  
 self.adds += 1 # Подсчет экземпляров  
 return MyList.\_\_add\_\_(self, other)  
   
 def stats(self):  
 return self.calls, self.adds # Все операции сложения, операции   
 # сложения для данного экземпляра  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 x = MyListSub(‘spam’)  
 y = MyListSub(‘foo’)  
 print(x[2])  
 print(x[1:])  
 print(x + [‘eggs’])  
 print(x + [‘toast’])  
 print(y + [‘bar’])  
 print(x.stats())  
   
% python mysub.py  
a  
[‘p’, ‘a’, ‘m’]  
[‘s’, ‘p’, ‘a’, ‘m’, ‘eggs’]  
[‘s’, ‘p’, ‘a’, ‘m’, ‘toast’]  
[‘f’, ‘o’, ‘o’, ‘bar’]  
(3, 2)  
4. Методы метакласса. Я решил это упражнение, как показано ниже. Обра-  
тите внимание, что в Python 2.6 операторы будут извлекать атрибуты с по-  
мощью метода \_\_getattr\_\_ – он должен возвращать значение, чтобы операто-  
ры могли работать. Внимание: как отмечалось в главе 30, метод \_\_getattr\_\_   
не вызывается встроенными операциями в Python 3.0, поэтому следующее   
решение не будет работать, как ожидается, в 3.0, – в классе, подобном это-  
му, необходимо явно переопределить методы \_\_X\_\_ перегрузки операторов.   
Подробнее об этом рассказывается в главах 30, 37 и 38.  
>>> class Meta:  
... def \_\_getattr\_\_(self, name):  
... print(‘get’, name)  
... def \_\_setattr\_\_(self, name, value):  
... print(‘set’, name, value)  
...  
>>> x = Meta()  
>>> x.append  
get append

Часть VI. Классы и ООП   
1237  
>>> x.spam = “pork”  
set spam pork  
>>>  
>>> x + 2  
get \_\_coerce\_\_  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
TypeError: call of non-function  
>>>  
>>> x[1]  
get \_\_getitem\_\_  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
TypeError: call of non-function  
   
>>> x[1:5]  
get \_\_len\_\_  
Traceback (innermost last):  
 File “<stdin>”, line 1, in ?  
TypeError: call of non-function  
5. Объекты  множеств. Ниже приводится сеанс взаимодействия, который   
должен у вас получиться. Комментарии описывают, какие методы вызы-  
ваются.  
% python  
>>> from setwrapper import Set  
>>> x = Set([1, 2, 3, 4]) # Вызывается \_\_init\_\_  
>>> y = Set([3, 4, 5])  
   
>>> x & y # \_\_and\_\_, intersect, затем \_\_repr\_\_  
Set:[3, 4]  
>>> x | y # \_\_or\_\_, union, затем \_\_repr\_\_  
Set:[1, 2, 3, 4, 5]  
   
>>> z = Set(“hello”) # \_\_init\_\_ удаляет повторяющиеся символы  
>>> z[0], z[-1] # \_\_getitem\_\_  
(‘h’, ‘o’)  
   
>>> for c in z: print(c, end=’ ‘) # \_\_getitem\_\_  
...  
h e l o  
>>> len(z), z # \_\_len\_\_, \_\_repr\_\_  
(4, Set:[‘h’, ‘e’, ‘l’, ‘o’])  
   
>>> z & “mello”, z | “mello”  
(Set:[‘e’, ‘l’, ‘o’], Set:[‘h’, ‘e’, ‘l’, ‘o’, ‘m’])  
Мой расширенный подкласс, позволяющий обрабатывать сразу несколько   
операндов, приводится ниже (файл multiset.py). В нем потребовалось изме-  
нить всего два метода из оригинального набора. Строка документирования   
в классе поясняет принцип его действия:  
from setwrapper import Set  
   
class MultiSet(Set):  
 “””  
 Наследует все атрибуты класса Set, но расширяет методы intersect

1238   
Приложение B. Решения упражнений   
 и union, добавляя возможность обработки нескольких операндов;   
 обратите внимание, что “self” – по-прежнему первый аргумент  
 (теперь сохраняется в списке аргументов \*args); кроме того,  
 обратите внимание, что теперь унаследованные операторы & и |   
 вызывают новые методы с двумя аргументами, но для одновременной   
 обработки более чем 2 операндов требуется вызов метода,   
 а не выражения:  
 “””  
   
 def intersect(self, \*others):  
 res = []  
 for x in self: # Сканировать первую последов-ть  
 for other in others: # Для всех остальных аргументов  
 if x not in other: break # Элемент присутствует во всех?  
 else: # Нет: прервать цикл  
 res.append(x) # Да: добавить элемент в конец  
 return Set(res)  
   
 def union(\*args): # Self - args[0]  
 res = []  
 for seq in args: # Для всех аргументов  
 for x in seq: # Для всех узлов  
 if not x in res:  
 res.append(x) # Добавить новый элемент в результат  
 return Set(res)  
Ваш сеанс взаимодействия с интерактивной оболочкой должен вы глядеть   
примерно так, как показано ниже. Обратите внимание, что пересечение   
двух множеств можно находить как с помощью опера тора &, так и с помо-  
щью метода intersect, но для случая трех и более множеств обязательно   
нужно вызывать метод intersect – оператор & является двухместным. Кро-  
ме того, обратите внимание, что можно было класс MultiSet назвать просто   
Set, чтобы сделать это изменение более прозрачным, если бы в файле mul-  
tiset в качестве имени насле дуемого класса мы использовали полное имя   
setwrapper.Set:  
>>> from multiset import \*  
>>> x = MultiSet([1,2,3,4])  
>>> y = MultiSet([3,4,5])  
>>> z = MultiSet([0,1,2])  
   
>>> x & y, x | y # Два операнда  
(Set:[3, 4], Set:[1, 2, 3, 4, 5])  
   
>>> x.intersect(y, z) # Три операнда  
Set:[]  
>>> x.union(y, z)  
Set:[1, 2, 3, 4, 5, 0]  
>>> x.intersect([1,2,3], [2,3,4], [1,2,3]) # Четыре операнда  
Set:[2, 3]  
>>> x.union(range(10)) # Два операнда также допустимы  
Set:[1, 2, 3, 4, 0, 5, 6, 7, 8, 9]  
6. Связи  в дереве  классов. Ниже приводятся измененный класс ListInstance   
и результаты повторного запуска теста. Сделайте то же самое, использовав

Часть VI. Классы и ООП   
1239  
функцию dir, а также задействуйте эту функцию в реализации вывода объ-  
ектов классов в виде древовидной структуры:  
class ListInstance:  
 def \_\_str\_\_(self):  
 return (“<Instance of %s(%s), address %s:\n%s>” % (  
 self.\_\_class\_\_.\_\_name\_\_, # Имя моего класса  
 self.supers(), # Мои суперклассы  
 id(self), # Мой адрес  
 self.attrnames()) ) # список name=value  
 def attrnames(self):  
 ...не изменился...  
 def supers(self):  
 names = []  
 for super in self.\_\_class\_\_.\_\_bases\_\_: # Вверх на один уровень  
 names.append(super.\_\_name\_\_) # имя, не repr(super)  
 return ‘, ‘.join(names)  
   
C:\python\examples> python testmixin.py  
<Instance of Sub(Super, ListInstance), address 7841200:  
 name data1=spam  
 name data2=eggs  
 name data3=42  
>  
7. Композиция. Мое решение с комментариями, взятыми из описания, приво-  
дится ниже (файл lunch.py). Это один из случаев, когда свою мысль проще   
выразить на языке Python, чем на естественном языке:  
class Lunch:  
 def \_\_init\_\_(self): # Создать/встроить Customer и Employee  
 self.cust = Customer()  
 self.empl = Employee()  
 def order(self, foodName): # Начать имитацию оформления заказа  
 self.cust.placeOrder(foodName, self.empl)  
 def result(self): # Узнать у клиента название блюда  
 self.cust.printFood()  
   
class Customer:  
 def \_\_init\_\_(self): # Инициализировать блюдо значением None  
 self.food = None  
 def placeOrder(self, foodName, employee): # Передать заказ официанту  
 self.food = employee.takeOrder(foodName)  
 def printFood(self): # Вывести название блюда  
 print(self.food.name)  
   
class Employee:  
 def takeOrder(self, foodName): # Вернуть блюдо с требуемым названием  
 return Food(foodName)  
   
class Food:  
 def \_\_init\_\_(self, name): # Сохранить название блюда  
 self.name = name  
   
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 x = Lunch() # Программный код самопроверки выполняется,

1240   
Приложение B. Решения упражнений   
 x.order(‘burritos’) # если запускается как сценарий,  
 x.result() # а не импортируется как модуль  
 x.order(‘pizza’)  
 x.result()  
   
% python lunch.py  
burritos  
pizza  
8. Классификация животных в зоологии. Ниже приводится мой спо соб реа-  
лизации классификации животных на языке Python (файл zoo.py)� этот   
пример достаточно искусственный, но сам принцип применим ко многим   
реальным ситуациям, начиная от реализации графического интерфейса   
и заканчивая базами данных сотрудни ков. Обратите внимание, что в клас-  
се Animal выполняется обраще ние к методу self.speak, это приводит к выпол-  
нению независимого поиска метода speak в дереве наследования, начиная   
с подкласса. Протестируйте работу этого дерева классов, следуя описанию   
уп ражнения. Попробуйте дополнить эту иерархию новыми классами и соз-  
дать экземпляры различных классов в дереве:  
class Animal:  
 def reply(self): self.speak() # Вызов метода подкласса  
 def speak(self): print(‘spam’) # Собственное сообщение  
   
class Mammal(Animal):  
 def speak(self): print(‘huh?’)  
   
class Cat(Mammal):  
 def speak(self): print(‘meow’)  
   
class Dog(Mammal):  
 def speak(self): print(‘bark’)  
   
class Primate(Mammal):  
 def speak(self): print(‘Hello world!’)  
   
class Hacker(Primate): pass # Наследует класс Primate  
9. Сценка с мертвым попугаем. Ниже приводится мое решение этого упраж-  
нения (файл parrot.py). Обратите внимание на то, как работа ет метод line   
в суперклассе Actor: он дважды обращается к атрибу там аргумента self,   
дважды отсылая интерпретатор к экземпляру, и тем самым дважды иници-  
ирует поиск в дереве наследования – self.name и self.says() с целью отыскать   
требуемую информацию в подклассах:  
class Actor:  
 def line(self): print(self.name + ‘:’, repr(self.says()))  
   
class Customer(Actor):  
 name = ‘customer’  
 def says(self): return “that’s one ex-bird!”  
   
class Clerk(Actor):  
 name = ‘clerk’  
 def says(self): return “no it isn’t...”  
   
class Parrot(Actor):

Часть VII. Исключения и инструменты   
1241  
 name = ‘parrot’  
 def says(self): return None  
   
class Scene:  
 def \_\_init\_\_(self):  
 self.clerk = Clerk() # Встраивание некоторых экземпляров  
 self.customer = Customer() # Scene – это составной объект  
 self.subject = Parrot()  
   
 def action(self):  
 self.customer.line() # Делегировать выполнение встроенным  
 self.clerk.line() # экземплярам  
 self.subject.line()  
Часть VII. Исключения и инструменты  
Упражнения находятся в главе 35, в разделе «Упражнения к седьмой части».  
1. try/except. Ниже приводится моя версия функции oops (файл oops.py). Что   
касается вопросов, не связанных с реализацией про граммного кода: если   
изменить функцию oops так, чтобы вместо In dexError она возбуждала ис-  
ключение KeyError, то обработчик в инст рукции try не сможет перехваты-  
вать исключение (оно достигнет верхнего уровня и приведет к появлению   
сообщения об ошибке по умолчанию). Имена KeyError и IndexError опреде-  
лены во встроенной области видимости. Импортируйте модуль builtins   
(\_\_builtin\_\_ – в Python 2.6) и передайте его функции dir, чтобы убедиться   
в этом:  
def oops():  
 raise IndexError  
   
def doomed():  
 try:  
 oops()  
 except IndexError:  
 print(‘caught an index error!’)  
 else:  
 print(‘no error caught...’)  
   
if \_\_name\_\_ == ‘\_\_main\_\_’: doomed()  
   
% python oops.py  
caught an index error!  
2. Объекты  исключений  и  списки. Ниже показано, как я дополнил модуль   
своим собственным исключением:  
class MyError(Exception): pass  
def oops():  
 raise MyError(‘Spam!’)  
def doomed():  
 try:  
 oops()

1242   
Приложение B. Решения упражнений   
 except IndexError:  
 print(‘caught an index error!’)  
 except MyError as data:  
 print(‘caught error:’, MyError, data)  
 else:  
 print(‘no error caught...’)  
if \_\_name\_\_ == ‘\_\_main\_\_’:  
 doomed()  
% python oops.py  
caught error: <class ‘\_\_main\_\_.MyError’> Spam!  
Как и в случае любого другого исключения на основе класса, в виде допол-  
нительных данных передается сам экземпляр – теперь сообщение об ошиб-  
ке содержит имя класса (<...>) и его экземпляр (Spam!).   
Экземпляр исключения должен наследовать методы \_\_init\_\_ и \_\_repr\_\_ (или   
\_\_str\_\_) от класса Exception, в противном случае он будет выводиться как   
экземпляр класса. Подробнее о том, как действуют классы встроенных ис-  
ключений, рассказывается в главе 34.  
3. Обработка  ошибок. Ниже приводится мое собственное решение этого   
упражнения (файл safe2.py). Я добавил тесты непосредственно в файл, что-  
бы не проводить их в интерактивной оболочке, но результаты от этого не   
меняются.  
import sys, traceback  
   
def safe(entry, \*args):  
 try:  
 entry(\*args) # Перехватывать любые исключения  
 except:  
 traceback.print\_exc()  
 print(‘Got’, sys.exc\_info()[0], sys.exc\_info()[1])  
   
import oops  
safe(oops.oops)  
   
% python safe2.py  
Traceback (innermost last):  
 File “safe2.py”, line 5, in safe  
 entry(\*args) # Перехватывать любые исключения   
 File “oops.py”, line 4, in oops  
 raise MyError, ‘world’  
hello: world  
Got hello world  
4. Далее приводится несколько примеров для самостоятельного изучения на   
досуге – еще больше примеров программ на языке Python вы найдете в по-  
следующих книгах и в Сети:  
# Поиск наибольшего файла в единственном каталоге  
   
import os, glob  
dirname = r’C:\Python30\Lib’

Часть VII. Исключения и инструменты   
1243  
allsizes = []  
allpy = glob.glob(dirname + os.sep + ‘\*.py’)  
for filename in allpy:  
 filesize = os.path.getsize(filename)  
 allsizes.append((filesize, filename))  
   
allsizes.sort()  
print(allsizes[:2])  
print(allsizes[-2:])  
   
# Поиск наибольшего файла с исходным программным кодом на языке Python   
# в дереве каталогов  
   
import sys, os, pprint  
if sys.platform[:3] == ‘win’:  
 dirname = r’C:\Python30\Lib’  
else:  
 dirname = ‘/usr/lib/python’  
   
allsizes = []  
for (thisDir, subsHere, filesHere) in os.walk(dirname):  
 for filename in filesHere:  
 if filename.endswith(‘.py’):  
 fullname = os.path.join(thisDir, filename)  
 fullsize = os.path.getsize(fullname)  
 allsizes.append((fullsize, fullname))  
   
allsizes.sort()  
pprint.pprint(allsizes[:2])  
pprint.pprint(allsizes[-2:])  
   
# Поиск наибольшего файла с исходным программным кодом на языке Python  
# в пути поиска модулей  
   
import sys, os, pprint  
visited = {}  
allsizes = []  
for srcdir in sys.path:  
 for (thisDir, subsHere, filesHere) in os.walk(srcdir):  
 thisDir = os.path.normpath(thisDir)   
 if thisDir.upper() in visited:  
 continue  
 else:  
 visited[thisDir.upper()] = True  
 for filename in filesHere:  
 if filename.endswith(‘.py’):  
 pypath = os.path.join(thisDir, filename)  
 try:  
 pysize = os.path.getsize(pypath)  
 except:  
 print(‘skipping’, pypath)  
 allsizes.append((pysize, pypath))  
   
allsizes.sort()  
pprint.pprint(allsizes[:3])  
pprint.pprint(allsizes[-3:])

1244   
Приложение B. Решения упражнений   
# Сумма по столбцам, разделенным запятыми, в текстовом файле  
   
filename = ‘data.txt’  
sums = {}  
   
for line in open(filename):  
 cols = line.split(‘,’)  
 nums = [int(col) for col in cols]  
 for (ix, num) in enumerate(nums):  
 sums[ix] = sums.get(ix, 0) + num  
   
for key in sorted(sums):  
 print(key, ‘=’, sums[key])  
   
# То же, что и выше, но суммы накапливаются в списке, а не в словаре  
   
import sys  
filename = sys.argv[1]  
numcols = int(sys.argv[2])  
totals = [0] \* numcols  
   
for line in open(filename):  
 cols = line.split(‘,’)  
 nums = [int(x) for x in cols]  
 totals = [(x + y) for (x, y) in zip(totals, nums)]  
   
print(totals)  
   
# Регрессивное тестирование результатов работы нескольких сценариев  
   
import os  
testscripts = [dict(script=’test1.py’, args=’’), # Или подставьте свои   
 dict(script=’test2.py’, args=’spam’)] # значения script/args  
   
for testcase in testscripts:  
 commandline = ‘%(script)s %(args)s’ % testcase  
 output = os.popen(commandline).read()  
 result = testcase[‘script’] + ‘.result’  
 if not os.path.exists(result):  
 open(result, ‘w’).write(output)  
 print(‘Created:’, result)  
 else:  
 priorresult = open(result).read()  
 if output != priorresult:  
 print(‘FAILED:’, testcase[‘script’])  
 print(output)  
 else:  
 print(‘Passed:’, testcase[‘script’])  
   
# Создание ГИП с помощью tkinter (Tkinter – в Python 2.6): кнопка,   
# изменяющая цвет и размер  
   
from tkinter import \* # Используйте модуль Tkinter в Python 2.6  
import random  
fontsize = 25  
colors = [‘red’, ‘green’, ‘blue’, ‘yellow’, ‘orange’, ‘white’, ‘cyan’,   
 ‘purple’]

Часть VII. Исключения и инструменты   
1245  
def reply(text):  
 print(text)  
 popup = Toplevel()  
 color = random.choice(colors)  
 Label(popup, text=’Popup’, bg=’black’, fg=color).pack()  
 L.config(fg=color)  
   
def timer():  
 L.config(fg=random.choice(colors))  
 win.after(250, timer)  
   
def grow():  
 global fontsize  
 fontsize += 5  
 L.config(font=(‘arial’, fontsize, ‘italic’))  
 win.after(100, grow)  
   
win = Tk()  
L = Label(win, text=’Spam’,  
 font=(‘arial’, fontsize, ‘italic’), fg=’yellow’, bg=’navy’,  
 relief=RAISED)  
L.pack(side=TOP, expand=YES, fill=BOTH)  
Button(win, text=’press’, command=(lambda: reply(‘red’))).pack(side=BOTTOM,fill=X)  
Button(win, text=’timer’, command=timer).pack(side=BOTTOM, fill=X)  
Button(win, text=’grow’, command=grow).pack(side=BOTTOM, fill=X)  
win.mainloop()  
   
# То же, что и выше, но на основе классов, поэтому каждое окно может  
# иметь свою собственную информацию о состоянии  
   
from tkinter import \*  
import random  
   
class MyGui:  
 “””  
 ГИП с кнопками, которые изменяют цвет и размер надписи  
 “””  
 colors = [‘blue’, ‘green’, ‘orange’, ‘red’, ‘brown’, ‘yellow’]  
   
 def \_\_init\_\_(self, parent, title=’popup’):  
 parent.title(title)  
 self.growing = False  
 self.fontsize = 10  
 self.lab = Label(parent, text=’Gui1’, fg=’white’, bg=’navy’)  
 self.lab.pack(expand=YES, fill=BOTH)  
 Button(parent, text=’Spam’, command=self.reply).pack(side=LEFT)  
 Button(parent, text=’Grow’, command=self.grow).pack(side=LEFT)  
 Button(parent, text=’Stop’, command=self.stop).pack(side=LEFT)  
   
 def reply(self):  
 “ при нажатии кнопки Spam изменяет цвет случайным образом “  
 self.fontsize += 5  
 color = random.choice(self.colors)  
 self.lab.config(bg=color,  
 font=(‘courier’, self.fontsize, ‘bold italic’))  
   
 def grow(self):  
 “при нажатии кнопки Grow начинает увеличивать размер надписи”

1246   
Приложение B. Решения упражнений   
 self.growing = True  
 self.grower()  
   
 def grower(self):  
 if self.growing:  
 self.fontsize += 5  
 self.lab.config(font=(‘courier’, self.fontsize, ‘bold’))  
 self.lab.after(500, self.grower)  
   
 def stop(self):  
 “при нажатии кнопки Stop останавливает увеличение размера”   
 self.growing = False  
   
class MySubGui(MyGui):  
 colors = [‘black’, ‘purple’] # Настройка изменения цвета  
   
MyGui(Tk(), ‘main’)  
MyGui(Toplevel())  
MySubGui(Toplevel())  
mainloop()  
   
# Сканирование и обслуживание ящика электронной почты  
   
“””  
Проверяет ящик входящей электронной почты, извлекает заголовки писем,  
позволяя удалять сообщения не загружая их полностью  
“””  
   
import poplib, getpass, sys  
   
mailserver = ‘здесь требуется указать имя почтового сервера pop’ # pop.rmi.net  
mailuser = ‘здесь требуется указать имя пользователя’ # brian  
mailpasswd = getpass.getpass(‘Password for %s?’ % mailserver)  
   
print(‘Connecting...’)  
server = poplib.POP3(mailserver)  
server.user(mailuser)  
server.pass\_(mailpasswd)  
   
try:  
 print(server.getwelcome())  
 msgCount, mboxSize = server.stat()  
 print(‘There are’, msgCount, ‘mail messages, size ‘, mboxSize)  
 msginfo = server.list()  
 print(msginfo)  
 for i in range(msgCount):  
 msgnum = i+1  
 msgsize = msginfo[1][i].split()[1]  
 resp, hdrlines, octets = server.top(msgnum, 0) # Получить заголовки  
 print ‘-’\*80  
 print(‘[%d: octets=%d, size=%s]’ % (msgnum, octets, msgsize))  
 for line in hdrlines: print(line)  
   
 if input(‘Print?’) in [‘y’, ‘Y’]:  
 for line in server.retr(msgnum)[1]: print(line) # Получить сообщ.  
 if input(‘Delete?’) in [‘y’, ‘Y’]:  
 print(‘deleting’)

Часть VII. Исключения и инструменты   
1247  
 server.dele(msgnum) # Удалить на сервере  
 else:  
 print(‘skipping’)  
finally:  
 server.quit() # Закрыть почтовый ящик  
raw\_input(‘Bye.’) # Предотвратить самопроизвольное закрытие окна  
   
# CGI-сценарий для взаимодействия с броузером на стороне клиента  
   
#!/usr/bin/python  
import cgi  
form = cgi.FieldStorage() # Разбор данных формы  
print “Content-type: text/html\n” # заголовок и пустая строка  
print “<HTML>”  
print “<title>Reply Page</title>” # разметка html страницы  
print “<BODY>”  
if not form.has\_key(‘user’):  
 print(“<h1>Who are you?</h1>”)  
else:  
 print(“<h1>Hello <i>%s</i>!</h1>” % cgi.escape(form[‘user’].value))  
print(“</BODY></HTML>”)  
   
# Сценарий для заполнения и выполнения запросов к базе данных MySql  
   
from MySQLdb import Connect  
conn = Connect(host=’localhost’, user=’root’, passwd=’darling’)  
curs = conn.cursor()  
try:  
 curs.execute(‘drop database testpeopledb’)  
except:  
 pass # Отсутствует  
   
curs.execute(‘create database testpeopledb’)  
curs.execute(‘use testpeopledb’)  
curs.execute(‘create table people (name char(30), job char(10), pay int(4))’)  
   
curs.execute(‘insert people values (%s, %s, %s)’, (‘Bob’, ‘dev’, 50000))  
curs.execute(‘insert people values (%s, %s, %s)’, (‘Sue’, ‘dev’, 60000))  
curs.execute(‘insert people values (%s, %s, %s)’, (‘Ann’, ‘mgr’, 40000))  
   
curs.execute(‘select \* from people’)  
for row in curs.fetchall():  
 print(row)  
   
curs.execute(‘select \* from people where name = %s’, (‘Bob’,))  
print curs.description  
colnames = [desc[0] for desc in curs.description]  
while True:  
 print(‘-’ \* 30)  
 row = curs.fetchone()  
 if not row: break  
 for (name, value) in zip(colnames, row):  
 print(‘%s => %s’ % (name, value))  
   
conn.commit() # Сохранить добавленные записи

1248   
Приложение B. Решения упражнений   
# Сценарий для заполнения базы данных shelve объектами Python   
   
# Дополнительные примеры использования модуля shelve приводятся в главе 27,   
# а примеры использования модуля pickle – в главе 30  
   
rec1 = {‘name’: {‘first’: ‘Bob’, ‘last’: ‘Smith’},  
 ‘job’: [‘dev’, ‘mgr’],  
 ‘age’: 40.5}  
   
rec2 = {‘name’: {‘first’: ‘Sue’, ‘last’: ‘Jones’},  
 ‘job’: [‘mgr’],  
 ‘age’: 35.0}  
   
import shelve  
db = shelve.open(‘dbfile’)  
db[‘bob’] = rec1  
db[‘sue’] = rec2  
db.close()  
   
# Сценарий для вывода и изменения базы данных shelve,   
# созданной предыдущим сценарием  
   
import shelve  
db = shelve.open(‘dbfile’)  
   
for key in db:  
 print(key, ‘=>’, db[key])  
   
bob = db[‘bob’]  
bob[‘age’] += 1  
db[‘bob’] = bob  
db.close()

Алфавитный указатель  
Символы  
/ (деление) оператор, 157  
[] (квадратные скобки)  
и генераторы списков, 425  
#, комментарии, 442  
�� (оператор равенства), 203  
�� (оператор сравнения), 306  
% (остаток/формат) оператор, 157  
| (побитовое ИЛИ) оператор, 157  
& (побитовое И) оператор, 157  
^ (побитовое исключающее ИЛИ) опера-  
тор, 157  
..., приглашение к вводу, 82  
>>>, приглашение к вводу, 79,•82  
<< (сдвиг влево) оператор, 157  
>> (сдвиг вправо) оператор, 157  
: (символ двоеточия), 327, 454  
\* (умножение) оператор, 157  
A  
abs, функция, 175  
\_\_add\_\_, метод, 718,•795  
all, функция, 430  
\_\_all\_\_, переменная, 666  
and, оператор, 157,•386  
anydbm, модуль, (Python 2.6), 757  
any, функция, 430  
Apache, 53  
apply, встроенная функция (Python 2.6),   
521  
ArithmeticError, исключение (встроен-  
ное), 963  
ASCII, набор символов, 997  
кодирование символов ASCII, 1006  
AssertionError, исключение, 946  
assert, инструкция, 325,•919,•946  
проверка соблюдения ограничений,   
947  
AttributeError, исключение, 809  
B  
\_\_base\_\_, атрибут, 722  
BaseException, исключение (встроен-  
ное), 962  
\_\_bases\_\_, атрибут, 785  
BOM (маркер порядка следования бай-  
тов), 1002  
обработка в Python 3.0, 1028  
bool, тип данных, 190  
\_\_bool\_\_, метод, 796,•821  
break, инструкция, 324,•394,•397  
bsddb, модуль расширения, 760  
\_\_builtin\_\_, модуль, 176,•476,•480  
bytearray, строковый тип, 1000  
использование объектов bytearray,   
1018  
bytes, тип объектов, 996  
литералы, 1009  
строковый тип, 1000  
C  
\_\_call\_\_, метод, 796,•816  
chr, функция, 561  
classtree, функция, 788  
class, инструкция, 325,•695,•710,•769,•  
1171  
атрибуты данных, 770  
вложенные инструкции, 770  
общая форма, 770  
пример, 770  
пространство имен, 770  
сравнение с именами в модуле, 770  
сравнение с функциями, 770  
\_\_class\_\_, атрибут, 722,•785  
\_\_cmp\_\_, метод, 821  
codecs.open, функция (Python 2.6), 1013  
COM, поддержка в MS �indows, 50  
\_\_contains\_\_, метод, 796,•807  
contextlib, модуль, 951  
continue, инструкция, 324,•394,•396

1250   
Алфавитный указатель  
cProfile, модуль, 988  
CPython, 70  
csh, командная оболочка, 1206  
C, язык программирования, интегра-  
ция, 56  
C++, язык программирования, 45,•56  
D  
Dabo, инструмент, 49  
dbm, модуль, 757  
\_\_debug\_\_, флаг, 947  
def, инструкции, 772  
def, инструкция, 324,•463,•465  
return, yield, 461  
и lambda, 549  
del, инструкция, 325  
\_\_del\_\_, метод, 795,•824  
\_\_delattr\_\_, метод, 796,•1060  
\_\_delete\_\_, метод, 797,•1053  
\_\_delitem\_\_, метод, 796  
dict, функция-конструктор, 414  
\_\_dict\_\_, атрибут, 722,•785,•838  
dir, функция, 99,•130,•442  
примесные классы, список унаследо-  
ванных атрибутов, 853  
distutils, 620  
distutils, модуль, 989  
Django, 49  
\_\_doc\_\_, атрибут, 443,•790  
doctest, модуль, 987  
DOM, парсеры, 1037  
E  
EBCDIC, кодировка, 1009  
Eclipse, интегрированная среда разра-  
ботки, 108,•987  
ElementTree, пакет, 1037  
else, инструкция, 397  
encodings, модуль, 999  
enumerate, функция, 414,•430  
\_\_enter\_\_, метод, 797,•951  
env, команда в UNIX, 90  
etree, пакет, 1038  
eval, функция, 69,•172,•527  
использование функции eval для   
строк в объекты, 296  
Exception, исключение (встроенное),   
962  
except, предложение инструкции try,   
931  
exec, функция, 69  
\_\_exit\_\_, метод, 797,•951  
F  
False, предопределенное имя, 190  
filter, функция, 430,•434,•562  
find, метод, 228  
for/else, инструкция, 324  
for, циклы, 400  
readlines, метод, 420  
вложенные, 404  
внутри функции, 469  
генераторы списков, 566  
итераторы, 419  
пример, 417  
кортежи, 401  
переменные цикла, 402  
общий формат, 400  
построчное чтение файлов с помо-  
щью метода \_\_next\_\_, 419  
расширенная операция распаковыва-  
ния последовательностей, 403  
строки, 401  
типичные варианты использования,   
401  
freeze, 73,•989  
from \_\_future\_\_, инструкция, 652  
from \*, инструкция, 625,•684  
from, инструкция, 325,•625,•683  
и import, 628  
модули, 607  
потенциальные проблемы, 628  
рекурсия, 687  
from, предложение (инструкция raise),   
945  
G  
\_\_get\_\_, метод, 1051  
\_\_getattribute\_\_, метод, 796,•886,•1044,•  
1059  
вычисляемые атрибуты, 1064  
предотвращение зацикливаний, 1061  
пример, 1064  
реализация шаблона делегирования,   
1074  
сравнение с методом \_\_getattr\_\_,   
1066  
управление атрибутами встроенных   
операций, 1069  
\_\_getattr\_\_, метод, 796,•809,•837,•1044,•  
1059  
вычисляемые атрибуты, 1064  
предотвращение зацикливаний, 1061  
пример, 1062

Алфавитный указатель   
1251  
реализация шаблона делегирования,   
1074  
сравнение с методом \_\_  
getattribute\_\_, 1066  
управление атрибутами встроенных   
операций, 1069  
\_\_getitem\_\_, метод, 796,•797,•800,•807  
global, инструкция, 325,•461,•464  
\_\_gt\_\_, метод, 820  
GTK, 49  
H  
help, функция, 131,•228,•446,•449  
hex, функция, 172  
HTML (Hyper Text Markup Language,   
гипертекстовый язык разметки), 49  
I  
\_\_iadd\_\_, метод, 795,•796,•814  
IDLE, интегрированная среда разработ-  
ки, 987  
if/elif/else, инструкция, 324  
if/else, трехместное выражение, 387  
if, инструкции  
 множественное ветвление, 377  
общая форма, 376  
простые примеры, 377  
import, инструкция, 95,•325,•610,•612,•  
624  
sys.path, список, 617  
выполняется только один раз, 626  
домашний каталог, 614  
дополнительные возможности выбо-  
ра модуля, 619  
запуск, 613  
и from, 628  
и пространства имен, 99  
использование расширений имен   
файлов, 455  
каталоги стандартной библиотеки,   
614  
когда необходимо использовать, 629  
компиляция, 613  
модули, 607  
переменная окружения   
P�THONPATH, 614  
примечания к использованию, 100  
содержимое файлов с расширением   
.pht, 614  
\_\_import\_\_, функция, 619,•678  
\_\_index\_\_, метод, 796  
IndexError, исключение, 922  
Informix, система управления базами   
данных, 50  
intersect, функция, 470  
int, функция, 172,•223,•336  
in, оператор проверки вхождения, 418,•  
429  
\_\_init\_\_.py, файлы, 643  
\_\_init\_\_, метод, 703,•719,•729,•795  
IronPython, 50,•70  
items, метод словарей, 437  
iter, функция, 418,•420,•421,•569  
\_\_iter\_\_, метод, 796,•802,•807  
J  
join, метод, 228,•230,•431  
Jython, 49,•70  
K  
keys, метод словарей, 437  
Komodo, интегрированная среда разра-  
ботки, 108,•987  
ksh, командная оболочка, 1206  
L  
lambda-выражения, 548  
вложенные, 552  
и def, 549  
когда используются, 549  
операторы, 157  
lambda, инструкция, 462  
lambda, тело, 548  
Latin-1, кодировка символов, 998  
LEGB, правило, 477,•483,•632  
len, функция, 126,•267,•463  
\_\_len\_\_, метод, 796,•821  
Linux, 53  
lister.py, файл, 850  
ListInherited, класс, 854  
ListInstance, класс, 850  
list, функция, 202,•230,•287,•431  
\_\_lt\_\_, метод, 796,•820  
M  
makedb.py, файл, 759  
Manager, класс, 728,•740  
map, функция, 411,•429,•434,•554,•574  
и генераторы списков, 511,•566  
и функциональное программирова-  
ние, 554  
math, модуль, 126

1252   
Алфавитный указатель  
\_\_metaclass\_\_, переменная, (Python 2.6),   
1172  
MFC, 49  
mins.py, файл, 526  
mod\_python, 49  
mybooks.xml, файл, 1037  
MySQL, система управления базами дан-  
ных, 50  
N  
\_\_name\_\_, атрибут, 667,•732  
.NET, платформа, 50  
NetBeans, интегрированная среда разра-  
ботки, 108,•987  
\_\_new\_\_, метод, 797  
next, функция, 420  
\_\_next\_\_, метод, 419,•802  
NLTK, пакет, 51  
None, объект, 309  
nonlocal, инструкция, 325,•494  
альтернативные решения для Python   
2.6, 499  
основы использования, 494  
примеры использования, 496  
граничные случаи, 497  
NotImplementedError, исключение, 779  
not, оператор, 157  
NULL, указатель в языке C, 309  
NumPy, расширение, 191  
O  
oct, функция, 172  
ODBC, 50  
open, функция, 145,•146,•463  
аргумент со строкой режима, 1002  
в Python 2.6, 1013  
Oracle, система управления базами дан-  
ных, 50  
ord, функция, 561  
or, оператор, 157,•386  
\_\_or\_\_, метод, 795  
OverflowError, исключение (встроен-  
ное), 963  
P  
Parrot, проект, 74  
pass, инструкция, 324,•394,•395  
PATH, переменная окружения, 78,•1204  
Perl, язык программирования, 53,•60  
person.py, файл, 729  
Person, класс, 728  
переносимости между версиями   
Python, 733  
создание подкласса, 740  
тестирование в процессе разработки,   
730  
pickle, модуль, 50,•757,•836  
для сериализации объектов (Python   
3.0), 1035  
PM�, 49,•73  
pop, метод, 269  
PostgreSQL, система управления базами   
данных, 50  
pow, функция, 555  
print30.py, файл, 530  
print, инструкция, 79,•88,•324,•362  
в интерактивном режиме, 82  
и sys.stdout, 372  
перенаправление потока вывода, 368  
программа Hello �orld, 367  
числа, 164  
print функция (Python 3.0), имитация,   
530  
использование аргументов, которые   
могут передаваться только по име-  
ни, 531  
property, встроенная функция, 1044  
вычисляемые атрибуты, 1047  
первый пример, 1046  
pstats, модуль, 988  
Psyco, 989  
динамический компилятор, 71  
.py, расширение имен файлов, 105  
py2exe, 73,•989  
PyChecker, 986  
PyDev, модуль расширения, 108  
PyDoc, 131,•441,•986  
help, функция, 442,•446,•449  
отчеты в формате HTML, 442,•449  
pygame, 51,•73  
PyGTK, библиотека создания графиче-  
ского интерфейса, 73  
PyInstaller, 73,•989  
Pylons, 49  
PyPI, веб-сайты, 51  
PyPy, 75  
PySol, программа, 51  
Python  
архитектура программы, 608  
импортирование и атрибуты, 609  
как организована программа, 609

Алфавитный указатель   
1253  
в сравнении с другими языками про-  
граммирования, 57  
параметры командной строки, 1208  
структура, 323  
сильные стороны Python, 41, 52  
высокая скорость разработки, 42  
интеграция компонентов, 42  
качество программного обеспече-  
ния, 41  
переносимость программ, 42  
поддержка библиотек, 42  
удовольствие от работы, 43  
Python 2.6  
nonlocal, инструкция� альтернатив-  
ные решения, 499  
классические классы и классы ново-  
го стиля, 869  
Python 3  
расширенная операция распаковыва-  
ния последовательностей  
в циклах for, 403  
Python 3.0  
аннотации функций, 545  
изменения в строковых типах, 996  
исключения, 933  
классы нового стиля, 869  
новые итерируемые объекты, 433  
поддержка Юникода и двоичных   
данных, 995  
расширенная операция распаковыва-  
ния последовательностей, 348  
смешивание типов, 261  
PythonCard, интегрированная среда раз-  
работки, 109  
Python for .NET, 71  
P�THONPATH, переменная окруже-  
ния, 614,•1204  
каталоги, 614  
P�THONSTARTUP, переменная окру-  
жения, 1204  
Python�in, интегрированная среда раз-  
работки, 109  
Python, язык программирования  
руководства и ресурсы, 1209  
строковая модель, 997  
так называемые необязательные осо-  
бенности языка, 1193  
PyUnit, 987  
Q  
Qt, 49  
R  
\_\_radd\_\_, метод, 795,•796,•814  
raise, инструкция, 325,•919,•943  
закрытие файлов и соединений с сер-  
вером, 977  
передача сигналов из функций по   
условию, 976  
предложение from, изменения   
в Python 3.0, 945  
random, модуль, 126,•176  
range, функция, 347,•393,•407,•433  
изменение списков, 410  
обход части последовательнсти, 409  
счетные циклы, 407  
raw\_input, функция, 335,•373,•975  
readlines, метод, 420  
reduce, функция, 430  
reload, функция, 95,•635  
вместе с инструкцией from, 685  
модули, 607  
основы использования, 636  
подробнее, 637  
пример использования, 637  
примечания к использованию, 100  
repr, функция, 164,•165,•223  
\_\_repr\_\_, метод, 796,•812,•818,•850  
return, инструкция, 324,•463,•464,•465  
re (регулярные выражения), модуль  
обработка строк в Python 3.0, 1032  
round, функция, 175  
Run Module, пункт меню в среде IDLE,   
100,•115  
S  
SAX, парсеры, 1038  
SciPy, расширение, 191  
self, 711  
set, функция, 147,•184  
\_\_set\_\_, метод, 1051,•1052  
\_\_setattr\_\_, метод, 796,•809,•1044,•1060  
вычисляемые атрибуты, 1064  
\_\_setitem\_\_, метод, 796,•797   
Shedskin, 72  
shelve, модуль, 757,•836  
обновление объектов в хранилище,   
762  
преимущества и недостатки, 761  
сохранение объектов в хранилище,   
758  
файлы базы данных, 760  
SIP, 50

1254   
Алфавитный указатель  
\_\_slots\_\_, атрибут, 860,•880  
дескрипторы и атрибут \_\_dict\_\_,   
1133  
и дескрипторы, 1059  
sorted, функция, 141,•429,•430,•455,•574  
sort, метод, 526  
SQLite, 50  
SQLObject, 50  
stdout, объект, 362  
StopIteration, исключение, 419,•569,•802  
struct, модуль для работы с двоичными   
данными (Python 3.0), 1033  
str, строковый тип, 1000  
сравнение в Python 3.0 и 2.6, 999  
str, функция, 165  
\_\_str\_\_, метод, 796,•812,•818  
метод перегрузки операции вывода   
объекта, 738  
sum, функция, 429,•574  
Swing, 49, 50  
Sybase, система управления базами дан-  
ных, 50  
SyntaxError, исключение, 935  
sys.exc\_info, функция, 979  
sys.exec\_info, функция, 950  
sys.modules, словарь, 674  
sys.path, список, 617,•672  
sys.stdout, объект, 372  
sys, модуль, 362  
T  
TCL\_LIBRAR�, переменная окружения,   
1204  
Tcl, язык программирования, 53  
testmixin.py, файл, 852  
testprint30.py, файл, 530  
text.py, файл, 1014  
Tkinter, 48,•49,•73  
GUI  
API, 696  
графический интерфейс, 103  
TK\_LIBRAR�, переменная окружения,   
1204  
True, предопределенное имя, 190  
try, инструкция  
обработка ошибок, 338  
отладка с помощью инструкции try,   
978  
синтаксис, 940  
try/else, инструкция, 934  
try/except, инструкция, 919,•925,•939  
вложение, 941  
вложенная, 972  
пример использования объединенной   
инструкции, 941  
try/except/else, инструкция, 929  
try/except/finally, инструкция, 325  
try/finally, инструкция, 919,•925,•936  
вложенная, 972  
программирование завершающих   
действий, 936  
tuple, функция, 287,•431,•509  
TurboGears, 49  
type, класс, 1170  
type, объект типа, 1168  
type, функция, 468  
U  
Unicode, строковый тип (в Python 2.X),   
999, 1011  
union, функция, 529  
unittest, модуль, 987  
update, метод, 269  
UTF-8, кодировка, 998  
V  
values, метод словарей, 437  
Vista, установка Python 2.5 из устано-  
вочного файла формата MSI, 1202  
W  
�eb�are, 49  
while/else, инструкция, 324  
while, циклы, 392  
имитация циклов while языка C, 399  
общий формат, 393  
примеры, 393  
win32all, пакет расширений для   
�indows, 662  
�ingIDE, интегрированная среда раз-  
работки, 109  
�ing, интегрированная среда разработ-  
ки, 987  
with/as, инструкция, 325,•919,•948  
wxPython, 49,•73  
X  
xmlrpclib, модуль, 51  
XML, язык разметки, 1036  
Y  
yield, инструкция, 325

Алфавитный указатель   
1255  
Z  
zip, функция, 407,•411,•430,•434  
конструирование словарей, 413  
ZODB, 50  
Zope, 49  
А  
абсолютный импорт, 651  
абстрактные суперклассы, 778  
в Python 2.6 и 3.0, 780  
автоматическое присвоение расширений   
файлам в �indows, 87  
автоматическое управление памятью, 55  
агрегирование, 832  
адаптация через наследование, 697  
алгоритм сопоставления, 1150  
альтернативные реализации  
CPython, 69  
IronPython, 70  
Jython, 70  
аннотации функций в Python 3.0, 545  
анонимные функции, 548  
аргументы  
со значениями по умолчанию, 490,   
516  
извлечение аргументов из коллек-  
ции, 519  
изменяемые объекты передаются по   
указателю, 506  
именованные, 516,•532  
командной строки, 222  
неизменяемые объекты передаются   
по значению, 506  
передача   
в функции, 464  
только по именам (Python 3.0),   
522  
когда использовать, 525  
порядок следования, 523  
через автоматическое присваива-  
ние объектов локальным име-  
нам, 505  
через операцию присваивания,   
511  
произвольное количество, 518  
встроенная функция apply   
(Python 2.6), 521  
обобщенные способы вызова   
функций, 520  
режимы сопоставления  
переменное число аргументов,   
512  
по именам, 512  
по позиции, 511  
по умолчанию, 512  
сбор аргументов в коллекцию, 518  
универсальные функции для работы   
с множествами, 528  
функция поиска минимума, 525  
три решения, 526  
ассоциативные массивы, 264  
атрибуты, 609,•729,•986  
def, инструкция, 770  
данных, 770  
имена, 782  
классов, 775  
механизма интроспекции, 1162  
псевдочастные, 839  
для чего нужны, 840  
функций, 501, 545  
экземпляров, 775  
Б  
базовые типы данных, 733  
базы данных, 50,•765  
сохранение объектов, 757  
модули pickle и shelve, 757  
байт-код, 45  
компиляция, 67  
библиотеки утилит, 55  
битовые операторы, 157  
блок else в циклах, 394  
быстрое создание прототипов, 50  
В  
ввод составных инструкций, 82  
веб-сайты, 764  
веб-службы, 764  
веб-сценарии, 49  
взаимодействие, 536  
взаимосвязи   
типа «имеет», 832  
типа «является», 830  
включение будущих возможностей язы-  
ка, 666  
вложенность   
модулей, 106  
функций, области видимости, 487  
циклов, 562  
внешние инструменты, 986  
возбуждение и обработка исключений,   
923  
собственных, 944  
возможности запуска, 111  
из текстового редактора, 111  
восьмеричные литералы, 155,•172

1256   
Алфавитный указатель  
вспомогательные модули, 156  
вспомогательные функции, 1164  
встраивание  
в сравнении с наследованием, 748  
кода на языке Python в C, 50  
расширение типов, 866  
встроенные  
инструменты, 55  
исключения  
операция вывода по умолчанию   
и сохранение информации, 964  
типы объектов, 55  
зачем нужны, 122  
расширение  
встраиванием, 866  
наследованием, 867  
функции  
abs, 175  
all, 430  
any, 430  
chr, 561  
dict, 414  
dir, 130,•442  
enumerate, 414,•430  
eval, 172,•527  
False, имя, 190  
filter, 430,•434,•562  
help, 131  
hex, 172  
\_\_import\_\_, 619,•678  
input, 975  
int, 172,•223,•336  
iter, 420,•421,•569  
len, 126,•463  
list, 202,•287,•431  
map, 411,•418,•429,•434,•554,•574  
\_\_name\_\_, атрибут, 667  
next, 420  
oct, 172  
open, 145,•146,•463  
ord, 561  
pow, 555  
random, модуль, 176  
range, 347,•393,•407,•433  
raw\_input, 335,•373  
reduce, 430  
reload, 635  
пример использования, 637  
repr, 164,•165,•223  
round, 175  
set, 147, 184  
sorted, 141,•429,•430,•574  
str, 165  
sum, 429,•574  
sys.path, список, 672  
True, имя, 190  
tuple, 287,•431,•509  
type, 468  
zip, 407,•411,•430,•434  
принудительное преобразование   
типов, 161  
числовые литералы, 156  
вызовы, 461,•466  
методов, 772, 773  
функций, 324  
выполнение программы, 64  
PVM (виртуальная машина Python),   
67  
компиляция в байт-код, 67  
производительность, 68  
скорость разработки, 69  
выражения-генераторы, 573,•857  
высокая скорость разработки, 43  
В  
генераторы, 567,•575  
send и next, методы, 572  
множеств, 147,•188,•584  
пример, 569  
словарей, 276, 584  
списков, 257, 417,•425,•429,•560  
for, циклы, 566  
map, функция, 561, 566  
выражения-генераторы, 573  
кортежи, 287  
матрицы, 564  
основы, 425  
проверки и вложенные циклы,   
562  
расширенный синтаксис, 427  
файлы, 426  
гибкость, нарушение, 148  
глобальная область видимости, 475  
глобальные переменные, минимизация   
количества, 483  
границы   
блоков, 379,•380  
инструкций, 379  
графический интерфейс пользователя,   
764  
групповое присваивание, 343,•344,•352  
разделяемые ссылки, 352  
В  
двоичные литералы, 155  
двоичные строки и строки Юникода,   
995

Алфавитный указатель   
1257  
двоичные файлы, 1002,•1021,•1022  
декодирование и кодирование, 998  
декораторы, 896,•1087,•1163  
аргументы, 1099  
и аннотации функций, 1152  
вложение, 1097  
классов, 899,•1088,•1094  
использование, 1095  
поддержка множества экземпля-  
ров, 1096  
программирование, 1100, 1116  
декораторы и управляющие   
функции, 1124  
изменение интерфейсов объ-  
ектов, 1119  
классы одиночных экземпля-  
ров, 1117  
реализация общедоступных   
атрибутов, 1134  
реализация частных атрибу-  
тов, 1130  
сохранение множества экзем-  
пляров, 1123  
частные и общедоступные   
атрибуты, 1130  
реализация, 1095  
сравнение с метаклассами, 1166,•  
1182,•1190  
определение и использование, 1089  
проверка типов, 1154  
управление   
вызовами и экземплярами, 1088  
функциями и классами, 1088,•  
1100,•1127  
нерешенные проблемы, 1137  
частные и общедоступные   
атрибуты, 1130  
функций, 820,•838,•897,•1088,•1090  
использование, 1090  
основы, 897  
поддержка декорирования мето-  
дов, 1093  
пример, 898  
проверка аргументов функций,   
1142  
нерешенные проблемы, 1151  
обобщение на именованные   
аргументы и аргументы со   
значениями по умолчанию,   
1146  
подробности реализации, 1149  
проверка позиционных аргу-  
ментов, 1143  
программирование, 1100  
декорирование методов клас-  
сов, 1106  
добавление аргументов, 1113  
сохранение информации о со-  
стоянии, 1102  
трассировка вызовов, 1100  
хронометраж вызовов, 1111  
реализация, 1091  
свойства, управление атрибута-  
ми, 1048  
делегирование, 748,•837  
в классах и встроенные операции,   
906  
деление  
истинное, 166  
классическое, 166  
с округлением вниз, 166  
дескрипторы, 886,•1050  
\_\_delete\_\_, метод, 1053  
\_\_get\_\_, метод, 1051  
\_\_set\_\_, метод, 1051,•1052  
аргументы методов, 1051  
атрибутов классов, 1162  
взаимосвязь со свойствами, 1058  
вычисляемые атрибуты, 1055  
использование собственных данных,   
1056  
методы, 1050  
основы, 1050  
первый пример, 1053  
реализация атрибута \_\_slots\_\_, 1059  
только для чтения, 1052  
файлов, 147  
деструктор, 824  
динамическая типизация, 55,•124,•194  
переменные, объекты и ссылки, 195  
разделяемые ссылки, 199  
и равенство, 203  
сборка мусора, 198  
типы, 197  
динамический компилятор (just-in-  
time, JIT), 71  
домашний каталог программы, 614  
дополняющее присваивание  
разделяемые ссылки, 355  
З  
завершающие действия  
с помощью инструкции try/finally,   
937  
заключительные операции, 921  
замыкания, 488

1258   
Алфавитный указатель  
записи, в виде классов, 723  
запуск Python  
интерпретатор, 66  
запутать программный код, как не, 551  
зарезервированные слова, 356  
значения по умолчанию, 598  
И  
иерархия   
типов, 310  
понятий, 323  
избыточность программного кода, 462  
извлечение срезов, 208, 798  
изменение внутри функции аргумента,   
который является изменяемым объ-  
ектом, 506  
изменение значений имен в других фай-  
лах, 627  
изменения в словарях в Python 3.0, 276  
генераторы словарей, 276  
использование оператора in вместо   
метода has\_key, 281  
представления словарей, 277  
и множества, 279  
сортировка ключей словаря, 280  
сравнивание словарей, 281  
изменения в соседних модулях, мини-  
мизация, 484  
изменяемые и неизменяемые строковые   
типы, 1004  
изменяемые   
аргументы, 508  
объекты, 598  
в операциях присваивания, 454  
типы, 250  
имена атрибутов, 479  
именованные аргументы, 532,•731  
имитация функции print в Python 3.0,   
530  
использование аргументов, которые   
могут передаваться только по име-  
ни, 531  
имитация частных атрибутов экземпля-  
ра, 811  
импорт   
и атрибуты, 609  
и области видимости, 633  
и перезагрузка, 94  
пример, 98  
примечания к использованию,   
100  
импортирование относительно пакетов,   
650  
в сравнении с импортированием по   
абсолютному пути, 655  
изменения в Python 3.0, 650  
основы, 651  
правила поиска модулей, 655, 656  
примеры, 657  
выбор модулей по относительно-  
му и абсолютному пути, 659  
импортирование внутри пакетов,   
658  
импортирование за пределами па-  
кетов, 657  
импортирование относительно   
текущего рабочего каталога,   
659,•661  
решение проблемы в Python 3.0, 654  
индексирование, 127, 208  
инкапсуляция, 705,•735,•828  
инструкции, 323  
в форме выражений, 551  
выражений, 360  
типичные, 360   
импортирования  
импортирование модулей по име-  
ни в виде строки, 677  
многострочные, 383  
не имеющие вложенных инструк-  
ций, 385  
определение, 323  
присваивания, 324, 342  
групповые, 343, 352  
разделяемые ссылки, 352  
дополнительные варианты, 345  
 дополняющие  
и разделяемые ссылки, 355  
комбинированные, 343,•353  
преимущества, 354  
кортежей и списков, 343  
последовательностей, 343  
свойства, 342  
формы, 343  
разделители, 332,•383  
специальные случаи, 332  
инструкции циклов вместо рекурсии,   
541  
инструменты разработки крупных про-  
ектов, 986  
интеграция  
компонентов, 50  
с языком C, 56

Алфавитный указатель   
1259  
интегрированные среды разработки, 987  
Eclipse, 108  
Komodo, 108  
NetBeans, 108  
PythonCard, 109  
Python�in, 109  
�ingIDE, 109  
интерактивные циклы, 334  
математическая обработка данных   
пользователя, 336  
пример простого цикла, 334  
проверка ввода, 337  
интерактивный режим, 77  
ввод составных инструкций, 82  
изменение строки приглашения   
к вводу, 82  
использование, 81  
отступы, 82  
интернет-модули, 49  
интерпретатор, 63  
соглашения по именованию, 358  
интерфейсы, программные, 48  
интроспекция функций, 544, 674,•1149  
информация о состоянии, 150,•729  
сохранение, 499,•1102  
с помощью атрибутов функций,   
501  
с помощью классов, 499  
исключения, 919,•985  
AssertionError, 946  
except, предложение инструкции try,   
пустое, 931  
IndexError, 922  
SyntaxError, 935  
вложенные обработчики исключе-  
ний, 971  
вложение в потоке управления,   
973  
синтаксическое вложение, 974  
возбуждение, 923  
и обработка собственных исклю-  
чений, 944  
встроенные  
ArithmeticError, 963  
BaseException, 962  
Exception, 962  
OverflowError, 963  
перехват, 936  
заключительные операции, 921,•924  
идиомы, 975  
sys.exc\_info, 979  
закрытие файлов и соединений   
с сервером, 977  
запуск тестов в рамках единого   
процесса, 978  
исключения не всегда являются   
ошибками, 975  
отладка с помощью внешних ин-  
струкций try, 978  
передача сигналов из функций по   
условию, 976  
использование, 971  
краткий обзор, 921  
назначение, 920  
на основе классов, 954,•956  
в сравнении с исключениями на   
основе строк, 956  
встроенные, 962  
иерархии исключений, 956  
категории встроенных исключе-  
ний, 963,•964  
определение текста исключения,   
965  
передача данных в экземплярах   
и реализация поведения, 966  
передача дополнительной инфор-  
мации об исключении, 966  
предоставление методов, 967  
преимущества, 954,•959  
создание, 957  
необычное управление потоком вы-  
полнения, 921  
обработка, 922  
особых ситуаций, 921  
ошибок, 920  
исключений по умолчанию, 921,   
922  
определяемые пользователем, 924  
повторное возбуждение исключений   
с помощью инструкции raise, 945  
пример, 921  
советы по применению исключений,   
980  
избежание пустых предложений   
except, 981  
используйте категории, 983  
что должно быть обернуто, 981  
строковые, 955  
уведомления о событиях, 920  
искусственный интеллект, 51  
исполняемые сценарии в UNIX, 88  
источники документации, 441  
dir, 442  
dir, функция, 442  
\_\_doc\_\_, атрибут, 443  
help, функция, 442

1260   
Алфавитный указатель  
PyDoc  
help, функция, 446,•449  
отчеты в формате HTML, 449  
веб-ресурсы, 442,•452  
комментарии #, 442  
отчеты в формате HTML, 442  
печатные издания, 442,•452  
стандартный набор руководств, 442,•  
452  
строки документирования, 442,•443,•  
445  
определяемые пользователем, 444  
стандарты, 445  
исходный код, компиляция, 67  
итераторы, 417,•567  
filter, функция, 434  
map, функция, 434  
range, функция, 433  
поддержка множественных ите-  
раторов, 436  
StopIteration, исключение, 419  
zip, функция, 434  
в Python 3.0, 433  
выражения-генераторы, 573  
другие контексты итераций, 429  
других встроенных типов, 423  
итерационный протокол, 419  
несколько итераторов в одном объ-  
екте, 804  
определяемые пользователями, 802  
представлений словарей, 437  
сканирование, 406  
файлов, 418  
итерации  
альтернативы, 586  
хронометраж, 586  
выполнение вручную, 420  
протокол, 142  
итерационный протокол, 419  
итерируемые объекты, 418  
К  
кавычки, 132  
каналы, 146  
каталоги стандартной библиотеки, 614  
категории встроенных исключений, 963  
категории  
на основе классов, 983  
типов, 249,•301  
отображения, 250  
последовательности, 250  
числа, 249  
квадратные скобки, 333  
и генераторы списков, 425  
классификация объектов, 301  
классы, 698,•700  
\_\_bases\_\_, атрибут, 785  
атрибуты, 710,•775  
зачем нужны, 696  
адаптация через наследование,   
697  
композиция, 696  
множество экземпляров, 697  
наследование, 696  
перегрузка операторов, 697  
и их хранение, 836  
и модули, 791,•863  
как записи, 723  
классические, 869  
метаклассы, 873  
множественное наследование, 849  
модули, 715  
нового стиля, 869  
\_\_getattribute\_\_, метод, 886  
\_\_slots\_\_, атрибут, 880  
изменения, 870  
изменения в модели типов, 871,•  
873,•874  
ромбоидальное наследование,   
876,•877  
свойства, 884  
статические методы и методы   
класса, 887  
операции присваивания внутри ин-  
струкции class, 710  
определяемые пользователем, 149  
основы программирования, 709  
вызов объектов, 710  
множество экземпляров, 709,•710,•  
711  
переопределение операторов язы-  
ка Python, 717,•718  
переопределение операторов языка   
Python, 717  
пример, 718  
программирование, 728, 769  
адаптация конструкторов, 745  
встраивание и делегирование, 747  
интроспекция, 750  
методы, 735  
определяющие поведение, 733  
перегрузка операторов, 737  
подклассы, 739  
создание экземпляров, 729  
сохранение объектов в базе дан-  
ных, 757  
псевдочастные атрибуты, 839  
для чего нужны, 840

Алфавитный указатель   
1261  
смеси, 849  
соглашение по именованию, 729  
создание деревьев классов, 701  
типичные проблемы, 901  
делегирование и встроенные опе-  
рации, 906  
изменение атрибутов, 901  
методы, классы и вложенные об-  
ласти видимости, 905  
многослойное обертывание, 907  
модификация изменяемых атри-  
бутов, 902  
шаблоны проектирования, 828  
перегрузка сигнатур вызова, 829  
классы встроенных исключений, 962  
иерархия классов, 962  
категории, 962  
код, компиляция, 67  
кодирование и декодирование, 998  
кодировки символов, 997  
командная строка интерпретатора   
Python, 84  
командные оболочки, 44  
комбинирование объектов, 747  
комбинированное присваивание, 344,•  
353  
инструкция, 343  
преимущества, 354  
комментарии, 162,•380  
компилируемые расширения, 985  
компиляция  
Psyco, динамический компилятор,   
71  
Shedskin, 72  
в байт-код, 67  
комплексные числа, 155,•171  
композиция, 696,•832  
пример, 832  
компоненты, интеграция, 50  
конец строки, 328  
конкатенация, 128,•208  
строк, 217  
конструкторы  
адаптация, 745  
программирование, 729  
концепции проектирования   
модулей, 681  
функций, 536  
копии и ссылки, 303  
короткая схема вычислений, 386  
кортежи, 99,•123,•144,•284  
в действии, 286  
гетерогенность, 285   
доступ к элементам по смещению,   
285  
запятые и круглые скобки, 286  
и генераторы списков, 287  
литералы кортежей и операции, 285  
массив ссылок на объекты, 285  
неизменяемые последовательности,   
285  
поддержка произвольного числа   
уровней вложенности, 285  
преобразования и неизменяемость,   
287  
упорядоченные коллекции объектов   
произвольных типов, 285  
фиксированная длина, 285  
циклы for, 401  
переменные цикла, 402  
косвенный вызов, 543  
круглые скобки, 327,•333,•454  
в выражениях, 160  
Л  
лексическая область видимости, 475  
литералы  
длинных целых чисел, 155  
чисел, 154  
чисел с плавающей точкой, 154  
шестнадцатеричные и восьмерич-  
ные, 155  
логические значения, 190,•389  
логические операторы, 157  
числа, 174  
локальная область видимости, 475  
локальные имена, 596  
локальные переменные, 471  
М  
математическая обработка данных поль-  
зователя, 336  
математические и научные вычисления,   
51  
математические функции, 156  
матрицы и генераторы списков, 564  
метаклассы, 873,•887,•896,•899,•1160,•  
1168  
добавление методов в классы, при-  
мер, 1179  
расширение вручную, 1180  
расширение с помощью метаклас-  
са, 1181  
области применения, 1161  
объявление, 1172  
подклассы класса type, 1170

1262   
Алфавитный указатель  
применение декораторов к методам,   
пример, 1186  
применение произвольных деко-  
раторов, 1189  
трассировка с декорированием   
вручную, 1186  
трассировка с использованием ме-  
таклассов и декораторов, 1187  
проблемы, связанные с использова-  
нием, 1161  
программирование, 1173  
использование фабричных функ-  
ций, 1175  
основы метаклассов, 1173  
перегрузка метода вызова про-  
цедуры создания в метаклассе,   
1176  
перегрузка метода вызова про-  
цедуры создания класса в обыч-  
ных классах, 1177  
расширение операций конструи-  
рования и инициализации,   
1174  
экземпляры и наследование, 1178  
сравнение с декораторами, 1166,•  
1182,•1190  
расширение с помощью декорато-  
ров, 1183  
управление экземплярами вместо   
классов, 1184  
метапрограммы, 674  
метод итераций X.\_\_next\_\_(), 423  
метод конструктора  
\_\_init\_\_, 729  
методы, 129,•736,•772  
find, 228  
join, 228,•230  
send и next, 572 , 891  
возможности, 774  
вызов, 227,•772,•773  
вызов конструкторов суперклассов,   
774  
класса, 843, 887  
обработки обращений к атрибутам,   
1162  
определяющие поведение, 733  
несвязанные, 888  
перегрузки операторов, 1162  
пример, 773  
проверка наличия подстроки в кон-  
це, 232  
проверка содержимого, 232  
расширение методов, 740  
связанные, 843, 848  
способы вызова методов, 773  
способы реализации, 735  
экземпляра, 843, 892  
мнимая часть, 171  
многократное использование программ-  
ного кода, 462  
многослойное обертывание в классах,   
907  
многострочные инструкции, 383  
множества, 147,•183  
генераторы множеств, 188  
литералы, 147,•185  
фиксированные, 187  
множественное ветвление, 377  
множественное наследование, 849,•903  
множество экземпляров, 697  
модель метаклассов, 1168  
подклассы класса type, 1170  
протокол инструкции class, 1171  
экземпляры класса type, 1168  
модель отношений реального мира, 830  
модули, 84,•607,•986  
from \*, инструкция, 625  
from, инструкция, 607,•625  
потенциальные проблемы, 628  
import, инструкция, 607,•624  
выполняется только один раз, 626  
когда необходимо использовать,   
629  
P�THONPATH, переменная окруже-  
ния, 614   
reload, функция, 607  
sys.path, список, 617  
вложенные, 106  
домашний каталог программы, 614  
дополнительные возможности, 665  
выбор модуля, 619  
изменение значений имен в других   
файлах, 627  
изменение пути поиска, 672  
и классы, 791,•863  
имена файлов, 623  
и пространства имен, 99  
использование, 624  
каталоги стандартной библиотеки,   
614  
классы, 715  
концепции проектирования, 681  
взаимозависимость, 681  
интроспекция, 674  
согласованность, 681  
метапрограммы, 674

Алфавитный указатель   
1263  
назначение, 608  
определение, 623  
основы программирования, 623  
повторная загрузка, 635  
повторное использование программ-  
ного кода, 608  
присваивание, 626  
пространства имен, 630  
вложенные, 634  
квалификация имен атрибутов,   
632  
путь поиска модулей, 612  
расширений, 624  
разделение системы пространств   
имен, 608  
реализация служб или данных для   
совместного пользования, 608  
смешанные режимы использования,   
667  
 тестирование модулей, 668  
соглашение по именованию, 729  
содержимое файлов с расширением   
.pht, 614  
создание, 623  
сокрытие данных, 665  
\_\_all\_\_, переменная, 666  
предотвращение копирования,   
666  
типичные проблемы, 682  
from \*, инструкция, 684  
from, инструкция, 683  
reload, функция, 678,•685  
импортирование модулей по име-  
ни в виде строки, 677  
порядок следования инструкций,   
683  
рекурсивный импорт инструкци-  
ей from, 687  
тестирование в интерактивной   
оболочке, 685  
эквивалентность инструкций import   
и from, 628  
модульное программирование, 55  
Н  
наследование, 696,•713,•773,•775,•828,•  
830  
object.attribute, 714  
class, 713  
абстрактные суперклассы, 778  
иерархия, 695  
изменения в подклассах, 714  
классы и суперклассы, 713  
множественное, 849,•903  
модель поиска в дереве наследова-  
ния, 776  
переопределение унаследованных   
методов, 776  
приемы организации взаимодей-  
ствия классов, 777  
пример, 714,•876  
ромбоидальное, 876  
пример, 876  
явное разрешение конфликтов   
имен, 877  
создание дерева атрибутов, 775  
специализация унаследованных ме-  
тодов, 776  
экземпляры наследуют атрибуты   
всех доступных классов, 714  
явное разрешение конфликтов имен,   
877  
настройка переменных окружения,   
1203  
не-ASCII сиволы  
кодирование и декодирование, 1007  
неизменяемость строки, 128  
необычное управление потоком выпол-  
нения, 921  
несвязанные методы, 888  
класса, 843  
несоставные инструкции, 385  
неформатированные строки, 208,•210,•  
214  
подавление экранирования, 214  
О  
обертывание, 981  
области видимости, 474,•781  
LEGB, правило, 477  
аргументы со значениями по умолча-  
нию, 490  
вложенные, 487, 905  
встроенная, 476,•480  
глобальные, 476  
замыкания, 488  
изменения в соседних модулях, ми-  
нимизация, 484  
локальные, 476  
объемлющие, 487  
def, инструкция, 476  
сохранение состояния с помощью   
аргументов по умолчанию, 490  
пример, 479

1264   
Алфавитный указатель  
присваивание именам, 476  
произвольное вложение, 493  
рекурсия, 476  
фабричные функции, 488  
обработка   
исключений, 922  
особых ситуаций, 921  
ошибок, 920  
проверкой ввода, 337  
обработчики  
исключений по умолчанию, 921,•922  
событий, 818  
объединение разделенных инструкций,   
385  
объектно-ориентированный язык сцена-  
риев, 44  
объектно-реляционные отображения,   
765  
объекты, 196  
вызов, 710  
два стандартных поля, 197  
исключений, 954  
итерируемые объекты, 418  
и функции, 542  
несвязанные методы класса, 843  
пространства имен, 732  
связанные методы экземпляра, 843  
система хранения, 50  
состояние и поведение, 710  
сохранение в базе данных, 757  
модули pickle и shelve, 757  
числа, 153,•162  
объекты-обертки, 1088  
объемлющие инструкции def, 477  
объемлющие области видимости, 487  
объявление кодировки по умолчанию   
в файлах, 1014  
ООП (объектно-ориентированное про-  
граммирование), 53, 695  
важные концепции, 747  
взаимосвязи типа «имеет», 832  
взаимосвязи типа «является», 830  
идеи, лежащие в основе, 697  
классы, 696, 698,•700  
адаптация через наследование,   
697  
композиция, 696  
множество экземпляров, 697  
наследование, 696  
перегрузка операторов, 697  
инкапсуляция, 705  
многократное использование про-  
граммного кода, 703  
платформы, 706  
поиск унаследованных атрибутов,   
697  
создание деревьев классов, 701  
фабрики, 861  
зачем нужны, 862  
шаблоны проектирования, 706  
экземпляры, 699  
операторы выражений, 156  
группировка подвыражений с помо-  
щью круглых скобок, 160  
 определение старшинства, 159  
перегрузка, 161  
смешивание операторов, 159  
смешивание типов, 160  
числа, 162  
операции  
над строками, 217  
присваивания  
внутри инструкции class, 710  
влияние на вызывающую про-  
грамму, 505  
оригинальный модуль string, 233  
остаток от деления, оператор, 157  
открытое программное обеспечение, 53  
отладка программ на языке Python, 112  
отладчик (IDLE), 107  
отладчики, 988  
отображения, 250  
отрицания оператор, 157  
отрицательные смещения в строках, 218  
отступы, 380,•454  
конец, 328  
правила, 380  
пробелы и символы табуляции,   
382  
правила оформления, 329  
очереди, 146  
ошибки при работе с функциями, 596  
значения по умолчанию, 598  
локальные имена, 596  
переменные цикла в объемлющей об-  
ласти видимости, 600  
функции, не возвращающие резуль-  
тат, 599  
П  
пакеты модулей  
from и import, инструкции, 646  
операция импортирования, 641, 647  
\_\_init\_\_.py, файлы, 643  
инициализация пакета, 644  
пространства имен модуля,   
644

Алфавитный указатель   
1265  
настройка пути поиска, 642  
поведение инструкции from \*,   
644  
точечный путь, 642  
пример импортирования, 645  
перегрузка операторов, 302,•697,•737  
имитация частных атрибутов экзем-  
пляра, 811  
итераторы  
несколько итераторов в одном   
объекте, 804  
определяемые пользователями,   
802  
ключевые идеи, 794  
общие методы, 795  
\_\_add\_\_, 795  
\_\_bool\_\_, 796,•821  
\_\_call\_\_, 796,•816  
\_\_cmp\_\_, 821  
\_\_contains\_\_, 796,•807  
\_\_del\_\_, 795  
\_\_delattr\_\_, 796  
\_\_delete\_\_, 797  
\_\_delitem\_\_, 796  
\_\_enter\_\_, 797  
\_\_exit\_\_, 797  
\_\_getattr\_\_, 796,•809  
\_\_getattribute\_\_, 796  
\_\_getitem\_\_, 796,•797,•800,•807  
\_\_gt\_\_, 820  
\_\_iadd\_\_, 796,•814  
\_\_index\_\_, 796  
\_\_init\_\_, 795  
\_\_iter\_\_, 796,•802,•807  
\_\_len\_\_, 796,•821  
\_\_lt\_\_, 796,•820  
\_\_new\_\_, 797  
\_\_next\_\_, 802  
\_\_or\_\_, 795  
\_\_radd\_\_, 796,•814  
\_\_repr\_\_, 796,•812  
\_\_setattr\_\_, 796,•809  
\_\_setitem\_\_, 796,•797  
\_\_str\_\_, 796,•812  
основные идеи, 717  
перегрузка сигнатур вызова, 829  
передача аргументов, 505  
переменные, 196  
DOS, 1206  
и основные выражения, 162  
использование, 195  
окружения, 1203  
создание, 195  
типы, 195  
переменные цикла, 492  
переносимый ANSI C, 54  
переносимый прикладной программный   
интерфейс баз данных, 50  
перечислимые типы данных, 347  
платформы, 706  
поведение объектов, 710  
по умолчанию, 934  
повторение последовательности, 314  
повторение строки, 128  
повторное возбуждение исключений   
с помощью инструкции raise, 945  
повторное использование программного   
кода, 608  
подавление экранированных последова-  
тельностей, 214  
подвыражения, 160  
подклассы, 699,•740  
замещение унаследованных атрибу-  
тов, 776  
программирование, 740  
наследование, адаптация и рас-  
ширение, 744  
полиморфизм, 743  
расширение методов, 740  
расширение встроенных типов, 867  
поиск унаследованных атрибутов, 697  
полиморфизм, 128,•149,•161,•468,•828  
intersect, функция, 470  
пример, 743  
положительные смещения в строках,   
218  
получение среза, 127  
пользовательский интерфейс IDLE, 102  
Tkinter, 103,•106  
вложенные модули, 106  
дополнительные возможности, 107  
запуск в �indows, 103  
запуск редактируемых файлов, 104  
запуск сценариев, 105  
многопоточные программы, 106  
настройка, 106  
отладчик, 107  
очистка экрана, 106  
ошибки соединения, 106  
повторное выполнение команд, 105  
подсветка синтаксиса, 103  
файлы с исходным текстом, 103  
поразрядный сдвиг, чисел, 174  
последовательности, 126,•207,•250  
отображение функций на, 554  
правила видимости, 475  
правила именования переменных, 355

1266   
Алфавитный указатель  
предложения инструкции try, 931  
формы, 931  
предупреждения, 976  
приглашение к вводу, 79  
приемы   
организации взаимодействия клас-  
сов, 777  
программирования циклов, 407  
приложения баз данных, 50  
пример использования объединенной   
инструкции try, 941  
примесные классы, 849  
создание, 850  
список атрибутов с привязкой   
к объектам в дереве классов,   
855  
список атрибутов экземпляра,   
850  
список унаследованных атрибу-  
тов, 853  
присваивание, 626  
именам, 476  
классификация имен, 782  
кортежей и списков, 343  
последовательностей, 343,•344  
дополнительные варианты, 345  
пробелы, 380  
проверка  
ввода, 337  
и вложенных циклов, 562  
истинности, 385  
and и or, операторы, 386  
короткая схема вычислений, 386  
оператор and, 387  
понятия «истина» и «ложь», 308  
ошибок, 926  
соблюдения ограничений, 947  
программа Hello �orld, 367  
программирование  
интерпретатор, 63  
модульное, 55  
системное, 48  
типичные ошибки, 453  
программные компоненты, 739  
программные ловушки импорта, 619  
программный код, 84  
многократное использование, 608,   
703  
самый простой класс на языке   
Python, 721  
производительность, 68  
произвольное вложение областей види-  
мости, 493  
пространства имен, 97,•99,•474,•608,•630,•  
781,•911  
вложенные, 634  
имена атрибутов, 782  
квалификация имен атрибутов, 632  
классификация имен, 782  
классы, 770  
объявление, 482  
простые имена, 782  
разделение системы пространств   
имен, 608  
словари, 785  
ссылки, 788  
протокол дескрипторов, 1045  
профилировщики, 987  
процедурная декомпозиция, 462  
прямая и косвенная рекурсия, 540  
псевдочастные атрибуты, 839  
для чего нужны, 840  
пустые предложения except, 931  
как избегать, 981  
пустые строки, 208,•380  
Р  
равенство, 306  
разделение системы пространств имен,   
608  
разделяемые ссылки, 199  
и равенство, 203  
расширение встроенных типов  
встраиванием, 866  
наследованием, 867  
расширения на языке Python, 985  
расширения файлов, 87  
рациональные числа, 147  
режимы сопоставления, 511  
аргументов, которые могут пере-  
даваться только по именам   
(Python 3.0), 522  
переменного числа аргументов, 512  
по именам, 512  
по позиции, 511  
по умолчанию, 512  
рекурсивные функции, 538  
альтернативные решения, 539  
вычисление суммы, 539  
и инструкции циклов, 541  
обработка произвольных структур   
данных, 541

Алфавитный указатель   
1267  
рекурсия, 476,•788  
ромбоидальное наследование, 876  
пример, 876  
явное разрешение конфликтов имен,   
877  
С  
сборка мусора, 55,•140,•198  
свободное программное обеспечение, 53  
свойства, 884,•1044,•1045  
вычисляемые атрибуты, 1047  
классов, 884, 1162  
определение с помощью декораторов,   
1048  
основы, 1046  
первый пример, 1046  
применение в классах нового стиля,   
1047  
связанные методы экземпляра, 843, 848  
связность, 536  
сдвига операторы, 157  
символы подчеркивания, 666  
синтаксис генераторов, 583  
синтаксические правила, 376,•379  
многострочные инструкции, 383  
системная командная строка и файлы,   
84  
исполняемые сценарии в UNIX, 88  
использование, 87  
сканирование файлов, 406  
словари, 123,•137,•264  
len, функция, 267  
pop, метод, 269  
update, метод, 269  
базовые операции, 267  
вложенность, 138  
генераторы, 276  
гетерогенность, 265   
для хранения записей, 723  
дополнительные методы, 269  
доступ к элементам по ключу, 264  
другие способы создания, 275  
замечания по использованию, 271  
категории изменяемых отображе-  
ний, 265  
изменение, 268  
имитация гибких списков, 271  
интерфейсы словарей, 274  
использование в качестве записей,   
273  
итерации и оптимизация, 142  
ключи, 138  
литералы и операции, 266  
неупорядоченные коллекции произ-  
вольных объектов, 264  
операции над последовательностями,   
271  
операции отображения, 138  
отсутствующие ключи, 143  
ошибки обращения к, 272  
переменная длина, 265  
пространств имен, 785  
сортировка по ключам, 140  
структуры разреженных данных,   
272  
таблица языков, 270  
таблицы ссылок на объекты (хеш-  
таблицы), 265  
число уровней вложенности, 265   
службы или данные для совместного   
пользования, 608  
смешивание типов, операторы выраже-  
ний, 160  
соглашения по именованию, 358  
содержимое файлов с расширением .pht,   
614  
создание дерева атрибутов, 775  
сокеты, 146  
составные   
инструкции, 376  
объекты, 747  
типы, 543  
сохранение информации о состоянии,   
499  
с помощью атрибутов функций, 501  
с помощью классов, 499  
сохранение объектов, 757  
модули pickle и shelve, 757  
специальные режимы сопоставления,   
511  
специальные символы, 998  
специальный случай оформления блока,   
334  
списки, 123,•133,•253  
базовые операции, 256  
вложенные, 135  
генераторы списков, 136  
гетерогенность, 254   
доступ к элементам по смещению,   
254  
изменение, 258  
индексы, 258  
категории изменяемых объектов,   
254  
литералы и операции, 255  
массивы ссылок на объекты, 254

1268   
Алфавитный указатель  
матрицы, 258  
методы, 260  
специфичные для типа, 134  
операции над последовательностями,   
133  
основные свойства, 253  
присваивание по индексам, 259  
присваивание срезам, 259  
проверка выхода за границы, 134  
срезы, 258  
упорядоченные коллекции объектов   
произвольных типов, 253  
часто используемые операции, 263  
число уровней вложенности, 254  
способы оптимизации, 989  
сравнение, 306  
операторы, 157  
языков, 57  
средства оптимизации, 71  
Psyco, динамический компилятор,   
71  
Shedskin, 72  
ссылки, 196  
и копии, 303  
на пространства имен, 788  
стандартная библиотека, 611  
каталоги, 614  
статически вложенные области видимо-  
сти, 487  
статические методы, 887  
альтернативы, 890  
в Python 2.6 и 3.0, 888  
использование, 891  
стратегический режим, 696  
строгая типизация, 124  
строки, 123,•126,•207,•997  
16- и 32-битные значения Юникода,   
1008  
bytes, строковый тип  
создание объектов bytes, 1017  
bytes объекты   
использование в Python 3.0, 1015  
методы, 1015  
операции над последователь-  
ностями, 1016  
bytearray, объекты, использование,   
1018  
dir, функция, 130  
help, функция, 131  
базовые операции, 217  
блоки в тройных кавычках, 208  
в апострофах, 210  
в кавычках, 208  
в тройных кавычках, 215  
выражения форматирования, 234  
другие способы представления, 131  
извлечение среза, 208, 219  
изменение, 225  
индексирование, 208, 218  
инструменты преобразования, 223  
инструменты синтаксического ана-  
лиза разметки XML, 1036  
использование файлов Юникода,   
1026  
чтение и запись в Python 3.0,   
1026  
кавычки, 132  
кодирование символов   
ASCII, 1006  
не-ASCII, 1007  
Юникода, 1006  
в Python 2.6, 1011  
кодировки, 997  
преобразования между, 1010  
конкатенация, 128,•208  
литералы, 208  
байтов, 210  
в апострофах, 210  
в кавычках, 210  
в тройных кавычках, 210  
в Юникоде, 210  
экранированные последователь-  
ности, 210,•211  
литералы и основные свойства, 1003  
метод форматирования, 234,•239  
методы  
вызов функции, 227  
извлечение атрибутов, 227  
специфичные для типа, 129  
наиболее типичные литералы строк   
и операции, 208  
неизменяемость, 128  
неформатированные, 208,•214  
обход в цикле, проверка на вхожде-  
ние, 209  
операции над последовательностями,   
126  
повторение, 128  
поиск по шаблону, 132  
преобразование типов, 1005  
примеры использования в Python   
3.0, 1003  
пустые, 208  
расширенная операция извлечения   
среза, 221  
символы обратного слеша, 212

Алфавитный указатель   
1269  
смешивание строковых типов в вы-  
ражениях, 1018  
форматирование, 208  
дополнительные возможности,   
236  
спецификаторы формата, 236  
строк из словаря, 238  
циклы for, 401  
строки документирования, 380,•443,•  
445,•790,•986  
определяемые пользователем, 444  
основное преимущество, 790  
стандарты, 445  
строковые исключения, 955  
строковые типы в Python 3.0, использо-  
вание, 1021  
структура организации программ, 609  
структуры данных, 122  
разреженных, 272  
суперклассы, 699,•777  
абстрактные, 778  
классы-смеси, 849  
конструкторы, 774  
наследование, 713  
расширение методов, 776  
суффиксы  
комплексные числа, 171  
сценарии, 84  
счетчик ссылок, 197  
Т  
тактический режим, 696  
текст, изменение строк, 225  
текстовые файлы, 1001,•1021  
в Python 3.0, 1023  
и двоичные файлы, 293  
текущий рабочий каталог, 617  
тестирование  
в интерактивной оболочке, 685  
в процессе разработки, 730  
запуск тестов в рамках единого про-  
цесса, 978  
Тим Петерс, 1161  
типы  
в модулях стандартной библиотеки,   
313  
изменяемые, 250  
ловушки, 313  
неизменяемые типы, 315  
повторение последовательности,   
314  
присваивание, 313  
циклические структуры данных,   
314  
объектов, 55,•121,•985  
базовые типы, 123  
зачем нужны встроенные типы,   
122  
и переменные, 197  
преобразование, 160  
сравнение, 307  
строк, 999  
точность представления чисел и длин-  
ные целые, 155  
У  
уведомления о событиях, 920  
удобство в использовании, 45  
указатель типа, 197  
унарные операторы, 157  
упакованные двоичные данные, сохра-  
нение в файлах и интерпретация, 298  
управление атрибутами  
\_\_delattr\_\_, метод, 1060  
\_\_getattribute\_\_, метод, 1059  
вычисляемые атрибуты, 1064  
пример, 1064  
\_\_getattr\_\_, метод, 1059  
вычисляемые атрибуты, 1064  
пример, 1062  
\_\_get\_\_, метод, 1051  
property, встроенная функция, 1046  
\_\_setattr\_\_, метод, 1060  
вычисляемые атрибуты, 1064  
встроенных операций, 1069  
вычисляемые атрибуты с помощью   
\_\_getattr\_\_ и \_\_getattrbute\_\_,   
1064  
дескрипторы  
аргументы методов, 1051  
взаимосвязь со свойствами, 1058  
вычисляемые атрибуты, 1055  
использование собственных дан-  
ных, 1056  
методы, 1050  
основы, 1050  
пример, 1053  
только для чтения, 1052  
предотвращение зацикливаний  
в методе \_\_getattr\_\_   
в методе \_\_getattrbute\_\_, 1061  
пример проверки атрибутов, 1078  
на основе дескрипторов, 1080  
на основе метода \_\_getattr\_\_,   
1081

1270   
Алфавитный указатель  
на основе метода \_\_  
getattribute\_\_, 1083  
на основе свойств, 1078  
реализация шаблона делегирования,   
1074  
свойства  
вычисляемые атрибуты, 1047  
определение с помощью декорато-  
ров, 1048  
первый пример, 1046  
сравнение методов \_\_getattr\_\_   
и \_\_getattribute\_\_, 1066  
сравнение приемов управления, 1067  
управление контекстом, 948  
управляемые атрибуты, 1043  
управляющие функции, 1164  
управляющий язык, 44  
упражнения,•решения, 1211–1242  
установка и настройка Python, 64,•1199  
возможно уже установлен, 1199  
где получить, 1200  
настройка, 1203  
переменных окружения, 1203  
переменные DOS, 1206  
установка, 1201  
интерпретатора, 1199  
файлы путей, 1207  
установка поддержки Tkinter в Linux,   
1205  
утилиты сторонних разработчиков, 56  
Ф  
фабрики, 861  
зачем нужны, 862  
фабричные функции, 488  
файлы, 123,•145,•289  
аргумент функции open со строкой   
режима, 1002  
в Python 3.0, 1001  
вызов метода close, 291  
выполнение, 66  
выталкивание выходных буферов на   
диск, 290  
закрытие файла вручную, 290  
запись строк в файл, 290  
из списка, 290  
изменение текущей позиции в фай-  
ле, 290  
инструменты, 299  
операции над файлами, 289  
открытие  
для записи, 290  
для чтения, 290  
пространства имен, 630  
путей, 1207  
с доступом по ключу, 146  
сохранение и интерпретация   
объектов Python в файлах, 294  
упакованных двоичных данных   
в файлах, 298  
средства, напоминающие файлы, 146  
текстовые и двоичные, 1021  
несоответствие типа и содержимо-  
го, 1024  
обработка маркера BOM   
в Python 3.0, 1028  
поддержка Юникода в Python 2.6,   
1031  
текстовые и двоичные в Python 3.0,   
293  
чтение   
следующей текстовой строки, 290  
следующих N байтов, 290  
файла целиком в единственную   
строку, 289  
файла целиком в список строк,   
290  
фиксированные файлы  
двоичные, 73  
исполняемые, 110  
Юникода файлы, 1026  
декодирование при чтении, 1027  
кодирование вручную, 1026  
кодирование при записи, 1027  
ошибки декодирования, 1027  
фигурные скобки, 333  
функции, 461,•985  
filter, 556  
lambda, 548  
reduce, 556  
аннотации, 542, 545  
анонимные, 548  
атрибуты, 545  
вызовы, 455, 466  
выполняющие непосредственные из-  
менения в объектах, 455  
для работы с числами, 175  
задачи, 462  
инструкции и выражения, имеющие   
отношение к функциям, 461  
интроспекция, 544  
как объекты, 542  
концепции проектирования, 536  
косвенный вызов, 543

Алфавитный указатель   
1271  
обратного вызова, 848, 553  
обработчики, 550  
определение, 461  
передача сигналов из функций по   
условию, 976  
пересечение последовательностей,   
469  
вызов, 470  
локальные переменные, 471  
определение, 469  
цикл for внутри функции, 469  
расширенные возможности, 536  
рекурсивные, 538  
альтернативные решения, 539  
вычисление суммы, 539  
и инструкции циклов, 541  
обработка произвольных струк-  
тур данных, 541  
прямая и косвенная рекурсия,   
541  
создание  
def, инструкция, 463,•465  
global, инструкция, 464  
return, инструкция, 463,•464,•465  
аргументы, 464  
функции-генераторы  
и выражения-генераторы, 574  
имитация функций zip и map, 576  
инструменты итераций  
итераторы однократного пользо-  
вания, 581  
создание собственной версии   
функции map(func, ...), 577  
создание собственных версий   
функций zip(...) и map(None,   
...), 579  
функциональное программирование,   
556  
filter, функция, 556  
reduce, функция, 556  
функциональные интерфейсы, 818  
Х  
хеши, 264  
хронометраж итерационных альтерна-  
тив, 586  
альтернативные реализации модуля   
хронометража, 590  
другие предложения, 595  
модуль time, 586  
модуль хронометража  
использование аргументов, кото-  
рые могут передаваться только   
поименам, 593  
результаты, 588  
сценарий, 587  
Ц  
целые числа, 154  
литералы, 154  
циклические структуры данных, 314  
циклы, 392  
break, инструкция, 394,•397  
continue, инструкция, 394,•396  
else, инструкция, 397  
for, 400  
pass, инструкция, 394,•395  
readlines, метод, 420  
while, 392  
общий формат, 393  
примеры, 393  
имитация циклов while языка C,   
399  
блок else в циклах, 394  
интерактивные, 334  
математическая обработка дан-  
ных пользователя, 336  
проверка ввода, 337  
простой пример, 334  
итераторы, 417  
общий формат, 394  
параллельный обход, 411  
приемы программирования, 407  
изменение списков, 410  
обход части последовательности,   
409  
счетные циклы, 407  
Ч  
числа, 123,•153,•249  
print, инструкция, 164  
базовые операторы, 162  
комплексные числа, 171  
литералы, 154  
рациональные, 147,•179  
смешивание числовых типов в выра-  
жении, 160  
с фиксированной точностью, 147,•177  
форматы отображения, 158,•164  
числа с плавающей точкой, 154

1272   
Алфавитный указатель  
Ш  
шаблоны проектирования, 706,•828  
шестнадцатеричные литералы, 155, 172  
Э  
экземпляры, 699,•729  
\_\_class\_\_, атрибут, 785  
атрибуты, 775  
имитация частных атрибутов, 811  
множество, 697  
передача данных в экземплярах и ре-  
ализация поведения, 966  
создание экземпляров, 729  
конструкторы, 729  
тестирование в процессе разработ-  
ки, 730  
экранированные последовательности,   
подавление, 214  
Ю  
Юникод, стандарт, 998  
кодирование строк, 1006  
Я  
языки  
сравнение, 57  
ярлыки, 90  
в �indows, 91  
ограничения, 94

Роберт ГЛАСС  
Факты и заблуждения   
профессионального программирования  
240 стр., книга в продаже  
Эти факты и заблуждения индустрии ПО фундаментальны, по­  
это му забывая о них или пренебрегая ими, вы действуете на свой   
страх и риск!   
Автор, имеющий огромный опыт работы в индустрии ПО, по­  
святил свой труд ее фактам, мифам и заблуждениям, представив   
55 фактов и 10 заблуждений, от носящихся к менеджменту, жиз­  
ненному циклу, качеству, исследованиям и обра зованию в сфере   
разработки ПО. Очень многие из них разработчикам известны   
и слишком многие оказались забытыми. Основное внимание   
уделяется менеджменту как глав ной про блеме современной ин­  
дустрии ПО, отрицательной роли рекламных кам паний, которые   
побуждают людей гоняться за миражами, и че ловечес кому фак­  
тору – спе ци али стам, без которых создание программ немыслимо.  
Роберт Гласс не избегает полемики. Каждый факт и каждое   
за блуж де ние в книге сопровождаются описанием споров и раз­  
но гласий, их окружающих. Издание адресовано широкому кругу   
читателей – от тех, кто управляет программными проектами, до   
программистов.  
Стив МАККОННЕЛЛ  
Профессиональная разработка   
программного обеспечения   
240 стр., книга в продаже  
Вы способны поставить в срок 90% своего ПО, не выйти из бюд­  
жета и соблюсти все реальные требования заказчика? А хотите?  
Стив Макконнелл, автор бестселлеров по разработке ПО,   
приво дит убедительные аргументы, доказывая, что отдельные   
успехи разра ботки ПО можно превратить в повседневную прак­  
тику, если сделать совершеннее саму профессию разработчика ПО   
на всех уровнях, начиная с отдельного специалиста и заканчивая   
отраслью в це лом. Он не только показывает, почему и как отрасль   
пришла к своему современному состоянию, но и описывает шаги,   
которые должен предпринять каждый, кто хочет подняться на   
новый уровень в создании ПО.   
Вы найдете ответы на многие вопросы. Почему устаревшие   
и неэффективные методики разработки ПО так живучи? Что такое   
«культ карго» в разработке ПО и кто его адепты? Насколько повы­  
шает рентабельность инвестиций применение лучших методик   
работы с ПО? Как подтвердить рентабельность проекта? Как стро­  
ится карьера профессионального разработчика ПО? Что больше   
влияет на ход проекта: хорошие кадры или хорошие методы?

Джон ЭРИКСОН   
Хакинг: искусство эксплойта, 2-е издание  
512 стр., книга в продаже  
Хакинг – это искусство творческого решения задач, подразуме­  
вающее нестандартный подход к сложным проблемам и исполь­  
зование уязвимостей программ.   
Автор не учит применять известные эксплойты, а объясняет   
их работу и внутреннюю сущность. Вначале читатель знакомится   
с основами программирования на C, ассемблере и языке команд­  
ной оболочки, учится исследовать регистры процессора. А усвоив   
материал, можно приступать к хагингу – перезаписывать память   
с помощью переполнения буфера, получать доступ к удаленному   
серверу, скрывая свое присутствие, и перехватывать соединения   
TCP. Изучив эти методы, можно взламывать зашифрованный тра­  
фик беспроводных сетей, успешно преодолевая системы защиты   
и обнаружения вторжений.  
Книга дает полное представление о программировании, ма­  
шинной архитектуре, сетевых соединениях и хакерских приемах.   
С этими знаниями ваши возможности ограничены только во­  
ображением. Материалы для работы с этим изданием имеются   
в виде загрузочного диска Ubuntu Linux, который можно скачать   
и использовать, не затрагивая установленную на компьютере ОС.  
Майкл САТТОН, Адам ГРИН и Педрам АМИНИ   
Fuzzing: исследование уязвимостей   
методом грубой силы  
560 стр., книга в продаже  
Фаззинг – это процесс отсылки намеренно некорректных данных   
в исследуемый объект с целью вызвать ситуацию сбоя или ошибку.   
Настоящих правил фаззинга нет. Это такая технология, при ко­  
торой успех измеряется исключительно результатами теста. Для   
любого отдельно взятого продукта количество вводимых данных   
может быть бесконечным. Фаззинг – это процесс предсказания,   
какие типы программных ошибок могут оказаться в продукте,   
какие именно значения ввода вызовут эти ошибки. Таким образом,   
фаззинг – это более искусство, чем наука.  
Настоящая книга – первая попытка отдать должное фаззингу   
как технологии. Знаний, которые даются в книге, достаточно для   
того, чтобы начать подвергать фаззингу новые продукты и стро­  
ить собственные эффективные фаззеры. Ключ к эффективному   
фаззингу состоит в знании того, какие данные и для каких про­  
дуктов нужно использовать и какие инструменты необходимы   
для управления процессом фаззинга.  
Книга представляет интерес для обширной аудитории: как для   
тех читателей, которым ничего не известно о фаззинге, так и для   
тех, кто уже имеет существенный опыт.

Нил ФОРД, Майкл НАЙГАРД, Билл де OРА и др.  
97 этюдов   
для архитекторов программных систем  
224 стр., книга в продаже  
Успешная карьера архитектора программного обеспечения   
требует хорошего владения как технологической, так и деловой   
сторонами вопросов, связанных с проектированием архитектуры.   
В этой необычной книге ведущие архитекторы ПО со всего света   
обсуждают важные принципы разработки, выходящие далеко за   
рамки технологий.   
Архитектор ПО выполняет роль посредника между командой   
разработчиков и бизнес­руководством компании, поэтому чтобы   
добиться успеха в этой профессии, необходимо не только овладеть   
различными технологиями, но и обеспечить работу над проек­  
том в соответствии с бизнес­целями. В книге несколько десятков   
архитекторов рассказывают о том, что считают самым важным   
в своей работе, дают советы, как организовать общение с другими   
участниками проекта, как снизить сложность архитектуры, как   
оказывать поддержку разработчикам. Они щедро делятся мно­  
жеством полезных идей и приемов, которые вынесли из своего   
многолетнего опыта.   
Федерико БЬЯНКУЦЦИ, Шейн УОРДЕН  
Пионеры программирования  
Диалоги с создателями наиболее популярных   
языков программирования  
608 стр., книга в продаже  
В книге собраны 27 интервью с людьми, стоявшими у истоков   
создания различных языков программирования, с гуру, чьи име­  
на на слуху в мире разработки ПО. Их размышления позволят   
читателю подняться на новый уровень осмысления проблем   
развития компьютерной отрасли, увидеть скрытые процессы,   
которые привели к тем или иным конструктивным решениям,   
узнать, какие цели ставили перед собой разработчики, на какие   
компромиссы им приходилось идти и какое влияние оказала их   
работа на современное программирование.  
Судьбы языков складывались по­разному – одни, сыграв свою   
роль, уступили место новациям, другие смогли чудесно воз­  
родиться с появлением новых технологий, но все они оставили   
значительный след в истории информатики.

Ян ГОЙВЕРТС, Стивен ЛЕВИТАН  
Регулярные выражения.   
Сборник рецептов  
608 стр., книга в продаже   
Сборник содержит более 100 рецептов, которые помогут научить­  
ся эффективно оперировать данными и текстом с применением   
регулярных выражений. Книга знакомит читателя с функция­  
ми, синтаксисом и особенностями этого важного инструмента   
в различных языках программирования: C#, Java, JavaScript, Perl,   
PHP, Python, Ruby и VB.NET. Предлагаются пошаговые решения   
наиболее часто встречающихся задач: работа с адресами URL   
и путями в файловой системе, проверка и форматирование ввода   
пользователя, обработка текста, а также обмен данными и работа   
с текстами в форматах HTML, XML, CSV и др.   
Данное руководство поможет как начинающему, так и уже   
опытному специалисту расширить свои знания о регулярных   
выражениях, познакомиться с новыми приемами, узнать все тон­  
кости работы с ними, научиться избегать ловушек и ложных совпа­  
дений. Освоив материал книги, вы сможете полнее использовать   
все те возможности, которые предоставляет умелое применение   
регулярных выражений, и тем самым сэкономите свое время.  
Джеффри ФРИДЛ   
Регулярные выражения, 3-е издание  
608 стр., книга в продаже  
Эта книга откроет перед вами секрет высокой производитель­  
ности. Тщательно продуманные регулярные выражения помогут   
избежать долгих часов утомительной работы и решить свои   
проблемы за 15 секунд. Ставшие стандартной возможностью во   
многих языках программирования и популярных программных   
продуктах, включая Perl, PHP, Java, Python, Ruby, MySQL, VB.NET,   
C# (и другие языки платформы .NET), регулярные выражения по­  
зволят вам автоматизировать сложную и тонкую обработку текста.  
Написанное простым и доступным языком, это издание по­  
зволит программистам легко разобраться в столь сложной теме.   
Рассматривается принцип действия механизма регулярных выра­  
жений, сравниваются функциональные возможности различных   
языков программирования и инструментальных средств, подроб­  
но обсуждается оптимизация, которая дает основную экономию   
времени! Вы научитесь правильно конструировать регулярные   
выражения для самых разных ситуаций, а большое число сложных   
примеров даст возможность сразу же использовать предлагаемые   
ответы для выработки элегантных и экономичных практических   
решений широкого круга проблем.

Ноа ГИФТ, Джереми ДЖОНC  
Python в системном администрировании   
UNIX и Linux  
512 стр., книга в продаже  
Книга сисадмина с 10­летним стажем и инженера­программиста   
демонстрирует, как можно с помощью языка Python эффектив­  
но решать разнообразные задачи управления серверами UNIX   
и Linux. Каждая глава посвящена определенной задаче, например   
многозадачности, резервному копированию данных или созда­  
нию собственных инструментов командной строки, и предлагает   
практические методы ее решения на языке Python.   
Среди рассматриваемых тем: организация ветвления про­  
цессов и передача информации между ними с использованием   
сетевых механизмов, создание интерактивных утилит с графиче­  
ским интерфейсом, организация взаимодействия с базами данных   
и создание приложений для Google App Engine. Авторами создана   
виртуальная машина на базе Ubuntu, которая включает исходные   
тексты примеров из книги.  
Ричард СТИВЕНС, Стивен РАГО  
UNIX. Профессиональное   
программирование, 2е издание  
1040 стр., книга в продаже  
Издание представляет собой подробнейшее справочное руковод­  
ст во для любого профессионального программиста, работаю­  
щего с UNIX. Стивену Раго удалось обновить и дополнить текст   
фундамен таль ного классического труда Стивенса, сохранив при   
этом стиль ори гинала. Содержание всех тем, примеров и приклад­  
ных программ обновлено в соответствии с последними версиями   
наиболее по пулярных реализаций UNIX.   
Среди важных дополнений главы, посвященные потокам и раз­  
ра ботке многопоточных программ, использованию интерфейса   
сокетов для организации межпроцессного взаимодействия (IPC),   
а также широкий охват интерфейсов, добавленных в последней   
версии POSIX.1. Аспекты прикладного программного интерфейса   
разъясняются на простых и понятных примерах, протестирован­  
ных на 4х платформах: FreeBSD 5.2.1, Linux 2.4.22, Solaris 9 и Mac   
OS X 10.3. Описывается множество ловушек, о которых следует   
помнить при написании программ для различных реализаций   
UNIX, и показывается, как их избежать, опираясь на стандарты   
POSIX.1 и Single UNIX Specification.

Адриан ГОЛОВАТЫЙ, Джейкоб КАПЛАН-МОСС  
Django. Подробное руководство,   
2-е издание  
560 стр., книга в продаже  
Эта книга посвящена Django 1.1 – последней версии фреймворка   
для разработки веб­приложений, который позволяет создавать   
и поддерживать сложные и высококачественные веб­ресурсы   
с минимальными усилиями. Django – это тот инструмент, кото­  
рый превращает работу в увлекательный творческий процесс,   
сводя рутину к минимуму. Данный фреймворк предоставляет   
общеупотребительные шаблоны веб­разработки высокого уров­  
ня абстракции, инструменты для быстрого выполнения часто   
встречающихся задач программирования и четкие соглашения   
о способах решения проблем.   
Авторы подробно рассматривают компоненты Django и методы   
работы с ним, обсуждают вопросы эффективного применения   
инструментов в различных проектах. Эта книга отлично подходит   
для изучения разработки интернет­ресурсов на Django – от основ   
до таких специальных тем, как генерация PDF и RSS, безопасность,   
кэширование и интернационализация. Издание ориентировано   
на тех, кто уже имеет навыки программирования на языке Python   
и знаком с основными принципами веб­разработки.  
Джефф ФОРСЬЕ, Пол БИССЕКС, Уэсли ЧАН  
Django.   
Разработка веб-приложений на Python  
456 стр., книга в продаже  
На основе простой и надежной платформы Django можно созда­  
вать мощные веб­решения на Python всего из нескольких строк   
кода. Авторы, опытные разработчики, описывают все приемы,   
инструменты и концепции, которые необходимо знать, чтобы   
оптимально использовать Django 1.0, включая все основные осо­  
бенности новой версии.  
Это полное руководство начинается с введения в Python, затем   
подробно обсуждаются основные компоненты Django (модели,   
представления и шаблоны) и порядок организации взаимодейст­  
вия между ними. Описываются методы разработки конкретных   
приложений: блог, фотогалерея, система управления содержи­  
мым, инструмент публикации фрагментов кода с подсветкой   
син таксиса. После этого рассматриваются более сложные темы:   
расширение системы шаблонов, синдицирование, настройка при­  
ложения администрирования и тестирование веб­приложений.  
Авторы раскрывают разработчику секреты Django, давая по­  
дробные разъяснения и предоставляя большое количество приме­  
ров программного кода, сопровождая их построчным описанием   
и иллюстрациями.

Марк САММЕРФИЛД  
Программирование на Python 3.   
Подробное руководство  
608 стр., книга в продаже  
Третья версия языка Python сделала его еще более мощным, удоб­  
ным, логич ным и выразительным. Книга «Программирование на   
Python 3» написана од ним из ведущих специалистов по этому   
языку, обладающим многолетним опытом работы с ним. Издание   
содержит все необходимое для практического освоения языка:   
написания любых программ с использованием как стандарт ной   
биб лиотеки, так и сторонних библиотек для языка Python 3, а так­  
же со здания собственных библиотечных модулей.  
Автор начинает с описания ключевых элементов Python, зна­  
ние которых необ ходимо в качестве базы. Затем обсуждаются бо­  
лее сложные темы, поданные так, чтобы читатель мог постепенно   
наращивать свой опыт: распре деление вычислительной нагрузки   
между несколькими процессами и потока ми, использование слож­  
ных типов данных, управляющих структур и функ ций, создание   
приложений для работы с базами данных SQL и с файлами DBM.   
Книга может служить как учебником, так и справочником.   
Текст сопровожда ется многочисленными примерами, доступ­  
ными на сайте изда ния. Весь код примеров был протестирован   
с окончательным релизом Python 3 в ОС Windows, Linux и Mac OS X.  
Дэвид БИЗЛИ  
Python. Подробный справочник  
864 стр., книга в продаже  
Это авторитетное руководство и детальный путеводитель по языку   
программирования Python. Книга детально описывает не только   
ядро языка, но и наиболее важные части стандартной библиотеки   
Python. Дополнительно освещается ряд тем, которые не рассмат­  
риваются ни в официальной документации, ни в каких ­либо   
других источниках.  
Читателю предлагается практическое знакомство с особен­  
ностями Python, включая генераторы, сопрограммы, замыкания,   
метаклассы и декораторы. Подробно описаны новые модули,   
имеющие отношение к разработке многозадачных программ,   
использующих потоки управления и дочерние процессы, а также   
предназначенные для работы с системными службами и органи­  
зации сетевых взаимодействий.  
В полностью переработанном и обновленном четвертом   
издании улучшена организация материала, что позволяет еще   
быстрее находить ответы на вопросы и обеспечивает еще боль­  
шее удобство работы со справочником. Книга отражает наиболее   
существенные нововведения в языке и в стандартной библиотеке,   
появившиеся в Python 2.6 и Python 3.

Издательство “СИМВОЛПЛЮС”  
Основано в 1995 году  
Наша специализация – книги компьютерной и деловой тематики.   
Наши из да ния – плод сотрудничества известных зарубежных   
и оте чест вен ных авторов, высококлассных переводчиков   
и компе тент ных науч ных редакторов. Среди наших деловых   
парт не ров из да тель ст ва: O’Reilly, Pearson Education, NewRiders,   
Addison Wesley, Wiley, McGrawHill, No Starch Press, Packt, Dorset   
House, Apress и другие.  
О нас  
Где купить  
Наши книги вы можете купить во всех крупных книжных ма га зи­  
нах России, Украины, Белоруссии и других стран СНГ. Однако   
по минималь ным ценам и оптом они продаются:  
СанктПетербург:  
главный офис издательства –  
В. О. 16 линия, д. 7 (м. Василеостровская),  
тел. (812) 324­5353  
Москва:  
московский филиал издательства –  
ул. Беговая, д. 13 (м. Динамо),   
тел. (495) 945­8100  
www.symbol.ru  
Заказ книг   
через Интернет http://www.symbol.ru   
   
Приглашаем   
к сотрудничеству  
Мы приглашаем к сотрудничеству умных и талантливых авторов,   
переводчиков и редакторов. За более подробной информацией   
обращайтесь, пожалуйста, на сайт издательства www.symbol.ru.   
Также на нашем сайте вы можете высказать свое мнение   
и замечания о наших книгах. Ждем ваших писем!  
Бесплатный каталог книг высылается по запросу.