

*Эффективное объектно-ориентированное
программирование*

3-е издание
охватывает Python 2.5

Изучаем

Python



O'REILLY®

Марк Лутц

Learning Python

Third Edition

Mark Lutz

O'REILLY®

Изучаем Python

Третье издание

Марк Лутц



Санкт-Петербург — Москва
2009

Марк Лутц

Изучаем Python, 3-е издание

Перевод А. Киселева

Главный редактор	<i>А. Галунов</i>
Зав. редакцией	<i>Н. Макарова</i>
Выпускающий редактор	<i>П. Щеголев</i>
Научный редактор	<i>Б. Попов</i>
Редактор	<i>Ю. Бочина</i>
Корректор	<i>С. Минин</i>
Верстка	<i>Д. Орлова</i>

Лутц М.

Изучаем Python, 3-е издание – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 848 с., ил.

ISBN 978-5-93286-138-7

Мощный, переносимый и легкий в использовании язык программирования Python идеально подходит для разработки самостоятельных программ и сценариев. Эта книга позволит быстро и эффективно овладеть базовыми основами языка Python независимо от степени предварительного знакомства с ним.

Третье издание «Изучаем Python» основано на материалах учебных курсов, которые автор, Марк Лутц, ведет уже на протяжении десяти лет. 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 инструментами языка Python – классами. Включены описания моделей и инструкций обработки исключений, а также обзор инструментов разработки, используемых при создании крупных программ. Обсуждаются изменения в ожидаемой версии 3.0. В конце глав представлены упражнения и вопросы, которые позволят применить новые знания на практике и проверить, насколько хорошо усвоен материал.

Если вы хотите понять, почему выбирают Python такие компании, как Google и Intel, Cisco и Hewlett-Packard, почему этот язык используют в NASA для научных вычислений, то эта книга станет для вас лучшей отправной точкой.

ISBN 978-5-93286-138-7

ISBN 978-0-596-51398-6 (англ)

© Издательство Символ-Плюс, 2009

Authorized translation of the English edition © 2008 O'Reilly Media, Inc. This translation is published 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 Законом РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, 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.

Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 21.11.2008. Формат 70×100¹/16. Печать офсетная.

Объем 53 печ. л. Тираж 2000 экз. Заказ №

Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Посвящается Вере.

Ты жизнь моя.

Оглавление

Предисловие	15
I. Введение	33
1. Python в вопросах и ответах	35
Почему программисты используют Python?	35
Является ли Python «языком сценариев»?	38
Все хорошо, но есть ли у него недостатки?	40
Кто в наше время использует Python?	41
Что можно делать с помощью Python?	42
В чем сильные стороны Python?	46
Какими преимуществами обладает Python перед языком X?	54
В заключение	55
Закрепление пройденного	55
2. Как Python запускает программы	58
Введение в интерпретатор Python	58
Выполнение программы	60
Разновидности модели выполнения	65
В заключение	70
Закрепление пройденного	71
3. Как пользователь запускает программы	72
Интерактивный режим	72
Системная командная строка и файлы	77
Щелчок на ярлыке файла	83
Импортирование и перезагрузка модулей	86
Пользовательский интерфейс IDLE	92
Другие интегрированные среды разработки	98
Встраивание вызовов	100
Фиксированные исполняемые двоичные файлы	101
Возможность запуска программ из текстового редактора	101
Прочие возможности запуска	101
Будущие возможности	102

Какие способы следует использовать?	102
В заключение	103
Закрепление пройденного	103
II. Типы и операции	109
4. Введение в типы объектов языка Python	111
Зачем нужны встроенные типы?	112
Числа	115
Строки	116
Списки	123
Словари	127
Кортежи	134
Файлы	135
Другие базовые типы	136
В заключение	139
Закрепление пройденного	140
5. Числа	142
Числовые типы в Python	142
Операторы выражений	145
Числа в действии	150
Другие числовые типы	160
В заключение	164
Закрепление пройденного	164
6. Интерлюдия о динамической типизации	166
Отсутствие инструкций объявления	166
Разделяемые ссылки	171
Динамическая типизация повсюду	176
В заключение	177
Закрепление пройденного	177
7. Строки	179
Литералы строк	181
Строки в действии	190
Форматирование строки	200
Строковые методы	204
Общие категории типов	211
В заключение	212
Закрепление пройденного	213
8. Списки и словари	215
Списки	215
Списки в действии	218

Словари	224
Словари в действии	227
В заключение	237
Закрепление пройденного	237
9. Кортежи, файлы и все остальное	239
Кортежи	240
Кортежи в действии	241
Файлы	244
Пересмотренный перечень категорий типов	252
Гибкость объектов	253
Сравнения, равенство и истина	257
Иерархии типов данных в языке Python	260
Другие типы в Python	262
Ловушки встроенных типов	262
В заключение	265
Закрепление пройденного	265
III. Инструкции и синтаксис	271
10. Введение в инструкции языка Python	273
Структура программы на языке Python	273
История о двух if	275
Короткий пример: интерактивные циклы	284
В заключение	289
Закрепление пройденного	290
11. Присваивание, выражения и print	292
Инструкции присваивания	292
Инструкции выражений	305
Инструкция print	307
В заключение	312
Закрепление пройденного	313
12. Условная инструкция if	315
Условные инструкции if	315
Синтаксические правила языка Python	318
Проверка истинности	323
В заключение	328
Закрепление пройденного	328
13. Циклы while и for	330
Циклы while	330
break, continue, pass и else	332

Циклы for	337
Итераторы: первое знакомство	342
Приемы программирования циклов	349
Генераторы списков: первое знакомство	357
В заключение	361
Закрепление пройденного	362
14. Документация	364
Источники документации в языке Python	364
Типичные ошибки программирования	377
В заключение	379
Закрепление пройденного	380
IV. Функции	383
15. Основы функций	385
Зачем нужны функции?	386
Создание функций	387
Первый пример: определения и вызовы	390
Второй пример: пересечение последовательностей	393
В заключение	396
Закрепление пройденного	396
16. Области видимости и аргументы	398
Правила видимости	398
Инструкция global	406
Области видимости и вложенные функции	410
Передача аргументов	418
Специальные режимы сопоставления аргументов	423
В заключение	436
Закрепление пройденного	437
17. Расширенные возможности функций	440
Анонимные функции: lambda	440
Применение функций к аргументам	447
Отображение функций на последовательности: map	449
Средства функционального программирования: filter и reduce	451
Еще раз о генераторах списков: отображения	452
Еще раз об итераторах: генераторы	459
Хронометраж итерационных альтернатив	466
Концепции проектирования функций	469
Типичные ошибки при работе с функциями	472
В заключение	477
Закрепление пройденного	477

V. Модули	483
18. Модули: общая картина	485
Зачем нужны модули?	486
Архитектура программы на языке Python	487
Как работает импорт	490
В заключение	498
Закрепление пройденного	499
19. Основы программирования модулей	501
Создание модуля	501
Использование модулей	502
Пространства имен модулей	509
Повторная загрузка модулей	514
В заключение	518
Закрепление пройденного	519
20. Пакеты модулей	521
Основы операции импортирования пакетов	521
Пример импортирования пакета	525
Зачем используется операция импортирования пакетов?	527
В заключение	531
Закрепление пройденного	532
21. Дополнительные возможности модулей	533
Соккрытие данных в модулях	533
Включение будущих возможностей языка	534
Смешанные режимы использования: <code>__name__</code> и <code>__main__</code>	535
Изменение пути поиска модулей	538
Расширение <code>import as</code>	539
Синтаксис относительного импорта	539
Концепции проектирования модулей	543
Типичные проблемы при работе с модулями	547
В заключение	555
Закрепление пройденного	555
VI. Классы и ООП	559
22. ООП: общая картина	561
Зачем нужны классы?	562
ООП с высоты 30 000 футов	564
В заключение	575
Закрепление пройденного	575

23. Основы программирования классов	577
Классы генерируют множество экземпляров объектов	577
Классы адаптируются посредством наследования	582
Классы могут переопределять операторы языка Python	586
Самый простой в мире класс на языке Python	589
В заключение	592
Закрепление пройденного	592
24. Подробнее о программировании классов	595
Инструкция class	595
Методы	599
Наследование	602
Перегрузка операторов	606
Пространства имен: окончание истории	624
Более реалистичный пример	631
В заключение	635
Закрепление пройденного	635
25. Шаблоны проектирования с классами	637
Python и ООП	637
Классы как записи	639
ООП и наследование: взаимосвязи типа «является»	641
ООП и композиция: взаимосвязи типа «имеет»	643
ООП и делегирование	648
Множественное наследование	649
Классы – это объекты: фабрики универсальных объектов	653
Методы – это объекты: связанные и несвязанные методы	655
Еще раз о строках документирования	657
Классы и модули	659
В заключение	659
Закрепление пройденного	660
26. Дополнительные возможности классов	661
Расширение встроенных типов	661
Псевдочастные атрибуты класса	665
Классы нового стиля	668
Статические методы и методы класса	677
Декораторы функций	681
Типичные проблемы при работе с классами	684
В заключение	690
Закрепление пройденного	690

VII. Исключения и инструменты	699
27. Основы исключений	701
Зачем нужны исключения?	702
Обработка исключений: краткий обзор	704
Инструкция try/except/else	709
Инструкция try/finally	716
Объединенная инструкция try/except/finally	718
Инструкция raise	722
Инструкция assert	725
Контекстные менеджеры with/as	726
В заключение	731
Закрепление пройденного	732
28. Объекты исключений	733
Исключения на основе строк	734
Исключения на основе классов	735
Общие формы инструкции raise	747
В заключение	748
Закрепление пройденного	749
29. Использование исключений	751
Вложенные обработчики исключений	751
Идиомы исключений	756
Советы по применению исключений	760
Типичные проблемы при работе с исключениями	764
Заклучение по основам языка	766
В заключение	771
Закрепление пройденного	772
VIII. Приложения	775
A. Установка и настройка	777
B. Решения упражнений	786
Алфавитный указатель	824

Предисловие

Эта книга представляет собой введение в язык программирования Python. Python – это популярный язык программирования, используемый как для разработки самостоятельных программ, так и для создания прикладных сценариев в самых разных областях применения. Это мощный, переносимый, простой в использовании и свободно распространяемый язык программирования.

Цель этой книги – позволить вам быстро овладеть основными принципами базового языка Python независимо от уровня вашей подготовки. Прочитав эту книгу, вы получите объем знаний, достаточный для использования Python в самых разных областях.

О третьем издании

За четыре года, прошедшие с момента выхода второго издания книги в 2003 году, произошли существенные изменения как в самом языке Python, так и в темах, которые я представляю на своих курсах обучения языку Python. Я постарался оставить как можно больше из предыдущего издания, кроме того, это новое издание отражает изменения последнего времени, появившиеся в самом языке Python и в обучении языку Python. Помимо этого была несколько изменена структура книги.

Изменения в языке Python

Если говорить о версии языка, это издание описывает Python 2.5 и отражает все изменения, появившиеся в языке с момента выхода второго издания книги. (Во втором издании описывался язык Python 2.2 и некоторые 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:

- Новая условная конструкция `B if A else C` (глава 12).
- Оператор контекста `with/as` (глава 27).
- Унификация конструкции `try/except/finally` (глава 27).
- Синтаксис относительного импорта (глава 21).
- Выражения-генераторы (глава 17).
- Новые особенности функций-генераторов (глава 17).

- Функции-декораторы (глава 26).
- Объектный тип множества (глава 5).
- Новые встроенные функции: `sorted`, `sum`, `any`, `all`, `enumerate` (главы 4 и 13).
- Объектный тип десятичных чисел с фиксированной точностью представления (глава 5).
- Расширенные сведения о файлах, списках, итераторах и многом другом (главы 13 и 17).
- Новые инструменты разработки: `Eclipse`, `dustutils`, `unittest` и `doctest`, расширения `IDLE`, `Shedskin` и т. д. (главы 3 и 29).

Менее значительные изменения в языке (такие, как широко используемые значения `True` и `False`, новая функция `sys.exc_info`, которая возвращает подробную информацию об исключении, и обработка строковых исключений, методы для работы со строками и встроенные функции `apply` и `reduce`) обсуждаются на протяжении всей книги. Кроме того, здесь приводится расширенное описание некоторых новых особенностей, впервые появившихся в предыдущем издании, включая третье измерение при работе со срезами и передачу произвольного числа аргументов функции, включая функцию `apply`.

Изменения в обучении языку Python

Кроме уже перечисленных изменений в самом языке это издание книги дополнено новыми темами и примерами, наработанными мною при преподавании на курсах обучения языку Python в последние годы. Например, здесь вы найдете:

- Новую главу о встроенных типах (глава 4).
- Новую главу о синтаксических конструкциях (глава 10).
- Полностью новую главу о динамической типизации с углубленным освещением этого вопроса (глава 6).
- Расширенное введение в ООП (глава 22).
- Новые примеры работы с файлами, областями видимости, вложенными инструкциями, классами, исключениями и т. д.

Множество изменений и дополнений было сделано, чтобы облегчить чтение книги начинающим программистам. Учитывая опыт преподавания языка Python на курсах, обсуждение некоторых моментов было перенесено в другие, более соответствующие тематике разделы. Например, описание списков и итераторов теперь приводится вместе с описанием оператора цикла `for`, а не с описанием функциональных инструментов, как это было ранее.

Поскольку книга фактически превратилась в стандартный учебник по Python, в этом издании существенно расширено описание основ языка, изложение материала стало более полным и наглядным.

В дополнение в этом издании приводится полностью обновленный комплект советов и рекомендаций, подобранных из опыта преподавания в течение последних 10 лет и практического использования Python в течение последних 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 ошибки, которые совершают начинающие программисты на моих курсах. Вообще основы языка в этом издании обсуждаются более широко, чем в предыдущих изданиях, во-первых, потому что сам язык был расширен, а во-вторых, потому что я добавил немало информации, которая имеет важное значение для практического применения.

Структурные изменения в этом издании

Учитывая разросшийся объем книги, материал, как и в предыдущем издании, разбит на несколько частей. Чтобы облегчить усвоение основ языка, весь материал поделен на несколько частей, каждая из которых содержит несколько глав. Например, типы и инструкции теперь описываются в двух разных частях, в каждой из которых основным типам и инструкциям отведены отдельные главы. Новая структура позволяет сказать больше, не перегружая читателя при этом. При переработке материала упражнения и описания наиболее распространенных ошибок были перемещены из конца главы в конец части, и теперь они приводятся в конце последней главы каждой части.

Упражнения, которые приводятся в конце каждой части, я также дополнил в этом издании кратким обзором и контрольными вопросами в конце каждой главы, что должно помочь вам «пролистать» прочитанный материал. Каждая глава заканчивается серией вопросов, которые помогут вам проверить, насколько хорошо вы поняли изложенный материал. В отличие от упражнений в конце каждой части, решения для которых приводятся в приложении В, ответы на вопросы в конце каждой главы следуют непосредственно за вопросами. Я рекомендую просматривать ответы, даже если вы уверены, что правильно ответили на вопросы, потому что эти ответы, кроме всего, 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. Она задумывалась как учебник по Python для программистов¹. Во многом она унаследовала из первых двух изданий и материал, и структуру, и акценты подачи материала. Где это необходимо, я расширил вводную

¹ Под «программистами» я подразумеваю всех, кто в прошлом написал хотя бы одну строчку программного кода на любом языке программирования. Если вы не относитесь к этой категории, эта книга все равно будет вам полезна, но вы должны знать, что она в основном обучает языку Python, а не основам программирования.

часть для начинающих и отделил более сложные темы от основного потока обсуждения, чтобы не усложнять описание основ. Поскольку это издание в значительной степени основано на проверенном временем опыте преподавания, оно, как и первые два, может служить вводным курсом для самостоятельного изучения языка Python.

Ограничение области применения книги

Третье издание представляет собой учебник по основам языка программирования Python и ничего больше. Здесь приводятся всесторонние сведения о языке, которые необходимо знать, прежде чем приступить 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 сети Интернет. Такой подход пригоден для опытных программистов – 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 необычными случаями необъяснимого изменения значений переменных, параметров по умолчанию и т. д. Цель этой книги состоит в том, чтобы дать твердое понимание основ языка Python, чтобы даже самые необычные случаи находили свое объяснение.

Это ограничение было введено преднамеренно. Ограничившись обсуждением основ языка, мы можем заняться более глубоким и полным их исследованием. Более полное обсуждение темы прикладного использования Python и справочные материалы, не вошедшие в эту книгу, вы найдете в других публикациях издательства O'Reilly, таких как «Programming Python», «Python Cookbook», «Python in a Nutshell» и «Python Pocket Reference». Цель книги, которую вы сейчас читаете, состоит исключительно в изучении языка Python, чтобы потом вы смогли применять его для решения прикладных задач, независимо от предметной области, в которой вы работаете.

По этой причине некоторые справочные материалы и разделы предыдущих изданий (примерно 15 процентов предыдущего издания) были вырезаны с целью расширить разделы с описанием основ языка Python. Благодаря этому читатели этого издания получают более полное представление об основах языка программирования и – как следствие – более полезный учебник по языку Python. В качестве заключительного упражнения в конце книги (глава 29) приводятся несколько усложненных примеров для самостоятельного изучения.

Об этой книге

В этом разделе приводятся некоторые наиболее важные замечания об этой книге в общем, не зависящие от номера издания. Никакая книга

не способна удовлетворить все нужды и потребности читателя, поэтому важно понимать основные цели книги.

Предварительные условия

В действительности книга не предъявляет никаких предварительных условий. Она с успехом использовалась как начинающими программистами, так и умудренными опытом ветеранами. Если у вас есть желание изучать Python, эта книга наверняка поможет вам. Наличие у читателя некоторого опыта в программировании не является обязательным, но будет совсем не лишним.

Эта книга задумывалась как введение в Python для программистов. Возможно, она не идеальна для тех, кто раньше никогда не имел дела с компьютерами (например, мы не будем тратить время, чтобы объяснять, что такое компьютер), но я не делал никаких 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 программирования.

Отношения этой книги с другими книгами

Эта книга охватывает все основные аспекты языка программирования Python, но при этом я старался ограничить круг обсуждаемых тем, чтобы уменьшить объем книги. Для сохранения простоты в ней рассматриваются самые основные понятия, используются небольшие и очевидные примеры и опущены некоторые незначительные детали, которые вы найдете в справочных руководствах. По этой причине данная книга скорее 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 сложная тема, которая, однако, является центральной для многих систем, основанных 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 упоминаться. Естественно, при таком подходе из поля зрения выпадает значительная часть общей картины.

Вообще говоря, Python стоит на более высоком качественном уровне относительно других языков в мире языков сценариев. Некоторые из его идей требуют более глубокого изучения, чем может вместить эта книга, поэтому с моей стороны было бы ответственно порекомендовать

продолжить его изучение после того, как вы закончите читать эту книгу. Я надеюсь, что большинство читателей продолжит изучение принципов разработки приложений на этом языке, обратившись к другим источникам информации.

Ориентированная в основном на начинающих программистов, книга «Изучаем Python» может быть дополнена другими книгами издательства O'Reilly о языке Python. Например, существует еще одна моя книга «Programming Python», содержащая более объемные и полные примеры наряду с описанием приемов прикладного программирования, которая задумывалась как продолжение книги, которую вы сейчас читаете. Текущие издания книг «Изучаем Python» и «Programming Python» представляют собой две части курса обучения, который преподает автор, – основы языка и прикладное программирование. Кроме того, в качестве справочника можно использовать еще одну книгу издательства O'Reilly, «Python Pocket Reference», где приводятся некоторые подробности, опущенные здесь.

Для дальнейшего изучения можно порекомендовать книги, содержащие дополнительные 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'Reilly) и «Python Essential Reference» (Sams) содержат справочную информацию. Книга «Python Cookbook» (O'Reilly) представляет собой сборник примеров для тех, кто уже знаком с приемами прикладного программирования. Поскольку выбор книг является делом достаточно субъективным, я рекомендую вам самостоятельно поискать такие, которые наиболее полно будут отвечать вашим потребностям. Неважно, какие книги вы выберете, главное чтобы вы помнили, что для дальнейшего изучения Python вам необходимы более реалистичные примеры, чем приводятся здесь.

На мой взгляд, эта 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 созданию программ и сценариев на языке Python. К тому моменту, когда вы закончите чтение этой книги, вы изучите не только сам язык, но и начнете понимать, как лучше применить его к решению ваших повседневных задач. Кроме 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 трехдневных практических курсов изучения языка Python. В конце каждой главы содержится список контрольных вопросов, а в конце последней главы каждой части – упражнения. Ответы на контрольные вопросы приводятся непосредственно в самих главах, а примеры решения упражнений – в приложении В.

Контрольные вопросы подобраны так, что они представляют собой краткий обзор рассмотренного материала, а упражнения спроектированы так, чтобы сразу же научить вас правильному стилю программирования и, как правило, каждое упражнение соответствует одному из ключевых аспектов курса.

Я настоятельно рекомендую прорабатывать контрольные вопросы и упражнения в ходе чтения книги не только для того, чтобы получить опыт программирования на Python, но и потому, что в упражнениях поднимаются проблемы, которые не обсуждаются нигде в книге. Ответы на вопросы в главах и примеры решения упражнений в приложении В в случае необходимости помогут вам выйти из затруднительных положений (вы можете заглядывать в ответы так часто, как это потребуется).

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 особенности языка, а не частности. Мы будем двигаться от простого к сложному: от встроенных типов объектов к инструкциям, элементам программ и т. д. Каждая глава является полным и самостоятельным описанием одной темы, но каждая последующая глава основана на идеях, введенных в предыдущих главах (например, когда речь пойдет о классах, я буду исходить из предположения, что вы уже знаете, как создаются функции), поэтому для большинства читателей имеет смысл читать книгу последовательно.

В общих чертах, эта книга описывает язык программирования Python при движении от простого к сложному. Каждая часть посвящена отдельной крупной характеристике языка – типам, функциям и т. д. В большинстве своем примеры являются законченными небольшими сценариями (некоторые из них являются достаточно искусственными, но они иллюстрируют достижение поставленной цели). Если быть более точным, здесь вы найдете:

Часть I. Введение

Изучение Python мы начнем с общего обзора этого языка и с ответов на очевидно возникающие вопросы: почему кто-то использует этот язык, для решения каких задач он может использоваться и т. д. 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 состоит в том, чтобы дать вам 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 типов объектов, таких как числа, списки, словари и т. д. Обладая только этими инструментами,

вы уже сможете писать достаточно сложные программы. Это самая важная часть книги, потому что она закладывает основу для последующих глав. В этой части мы также рассмотрим динамическую типизацию и ссылки – ключевые аспекты языка Python.

Часть III. *Инструкции и синтаксис*

В следующей 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 основном сосредоточена на описании синтаксиса, тем не менее, здесь приводятся сведения о дополнительных инструментальных средствах, таких как система PyDoc, и рассматриваются альтернативные стили написания программного кода.

Часть IV. *Функции*

В этой части мы начнем рассматривать высокоуровневые способы структурирования программ на языке Python. Функции предоставляют простой способ упаковки программного кода многократного использования и предотвращения появления избыточного кода. В этой части мы исследуем правила видимости программных элементов в языке Python, приемы передачи аргументов 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. Здесь мы также рассмотрим некоторые более сложные темы, такие как пакеты модулей, перезагрузка модулей и переменная `__name__`.

Часть VI. *Классы и ООП*

Здесь мы приступим к исследованию объектно-ориентированного программирования (ООП). *Классы* – это необязательный, но очень мощный инструмент структурирования программного кода многократного использования. Здесь вы увидите, что классы по большей части используют идеи, которые будут описаны к этому моменту, а ООП в языке Python в основном представляет 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 разработки, особенно если речь идет о долгосрочных проектах.

Часть VII. *Исключения и инструменты*

Изучение языка мы закончим рассмотрением модели обработки исключительных ситуаций, а также кратким обзором инструментальных средств разработки, которые особенно удобны при разработке крупных программ (например, инструменты отладки и тестирования). Эта часть появляется в последний раз, в следующих версиях все исключения должны быть классами.

Часть VIII. Приложения

Книга заканчивается двумя приложениями, где приводятся рекомендации по использованию языка Python на различных платформах (Приложение А) и варианты решения упражнений, которые приводятся в конце каждой части (Приложение В). 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 материалами (эта книга – учебник, а не справочник). Как уже говорилось выше, в качестве справочников по синтаксису и встроенным особенностям языка Python вы можете использовать книгу «Python Pocket Reference» (O'Reilly) и справочники на сайте <http://www.python.org>.

Обновления книги

Книга продолжает улучшаться (и исправляются опечатки). Обновления, дополнения и исправления к этой книге можно найти в сети Интернет на одном из следующих сайтов:

<http://www.oreilly.com/catalog/9780596513986/>
(веб-страница книги на сайте 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 воспользоваться поисковой системой, чтобы восстановить ее. Если бы я был ясновидящим, 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 2.5. Я не стремлюсь предсказывать будущее, тем не менее, в ходе изучения мы будем обсуждать некоторые идеи, которые, как ожидается, будут реализованы в версии 3.0.

Однако, так как эта книга описывает базовые основы языка, можно быть уверенным, что большая часть из того, о чем здесь рассказывается, в следующих версиях Python изменится не очень сильно. Большая часть информации из этой книги применима и к более ранним версиям Python, кроме некоторых случаев; и, естественно, в случае использования расширений, которые появятся после выхода этой книги, ничего гарантировать нельзя.

Существует эмпирическое правило: лучшей версией Python является последняя его версия. Так как эта книга описывает основы языка, большинство сведений применимо к Jython – реализации Python на языке Java, а также к другим реализациям, описанным в главе 2.

Исходные тексты примеров, а также ответы к заданиям можно получить на веб-сайте книги по адресу <http://www.oreilly.com/catalog/9780596513986/>. Вас волнует вопрос, как запускать примеры? Он во всех подробностях обсуждается в главе 3, поэтому потерпите до этой главы.

В преддверии выхода Python 3.0

Первая альфа-версия Python 3.0 вышла как раз перед тем, как эта книга была отправлена в печать, уже после того, как она была написана. Официально это издание книги основано на линейке версий Python 2.x (в частности, на версии 2.5), но она была дополнена многочисленными примечаниями об ожидаемых изменениях в версии Python 3.0.

Официальная версия 3.0 выйдет как минимум через год после выхода этой книги и едва ли будет широко использоваться, по меньшей мере, еще два года. Однако, если вы приобрели эту книгу, когда версия 3.0 уже получила широкое распространение, в этом разделе вы найдете краткое описание изменений в языке, которое поможет вам выполнить переход на новую версию.

За редкими исключениями, в основном язык Python 3.0 будет соответствовать описываемому в книге и влияние этих изменений на типичный программный код будет весьма незначительным. То есть основы языка Python, о которых говорится в этой книге, не будут изменяться от версии к версии, и благодаря этому читатели могут с пользой для себя изучать эти основы, прежде чем переходить к рассмотрению особенностей, характерных для конкретных версий.

Однако, чтобы помочь вам в будущем, ниже приводится список основных отличий Python 3.0. Здесь же приводятся ссылки на главы, где обсуждаются или упоминаются эти изменения. Этот список упорядочен по возрастанию номеров глав. Некоторые из этих изменений могут быть реализованы в современной версии Python 2.5, а некоторые нет. Поскольку в данный момент для большинства читателей изучать этот список не имеет большого смысла, я рекомендую сначала прочитать эту книгу, чтобы изучить основы языка Python, а затем вернуться сюда позднее и ознакомиться с грядущими изменениями. В версии Python 3.0:

- Удалена нынешняя встроенная функция `execfile()`. Вместо нее следует использовать функцию `exec()` (глава 3).
- Встроенная функция `reload()` возможно будет удалена. Альтернатива пока не известна (главы 3 и 19).
- Преобразование в строковое представление с помощью обратных кавычек ``X`` будет недоступно: используйте функцию `repr(X)` (глава 5).

- Избыточная операция проверки на неравенство $X <> Y$ будет удалена: используйте операцию $X != Y$ (глава 5).
- Множества можно будет создавать с использованием синтаксиса литералов `{1, 3, 2}`, что эквивалентно используемой в настоящее время форме записи: `set([1, 3, 2])` (глава 5).
- Множества могут определяться программно: `{f(x) for x in S if P(x)}`, что эквивалентно используемой в настоящее время форме записи выражения-генератора: `set(f(x) for x in S if P(x))` (глава 5).
- Операция деления X/Y всегда возвращает число с плавающей точкой, даже если оба операнда являются целыми числами. Чтобы получить нынешнюю операцию деления с усечением дробной части, следует использовать $X//Y$ (глава 5).
- Существует единственный целочисленный тип `int`, который обеспечивает точность представления целых чисел, соответствующую нынешнему типу `long` (глава 5).
- Восьмеричные и двоичные литералы: текущая форма записи восьмеричных чисел `0666` будет вызывать ошибку: используйте вместо нее запись в форме `0o666`, соответствующим образом будет изменен и результат, возвращаемый функцией `oct()`. Также запись в форме `0b1010` будет эквивалентна числу 10, а функция `bin(10)` будет возвращать `"0b1010"` (глава 5).
- Строковый тип `str` поддерживает текст Unicode, а для представления строк с однобайтовыми символами создан новый тип `bytes` (например, для случаев, когда текст загружается из файлов в двоичном режиме). Тип `bytes` — это последовательность переменной длины малых целых чисел с интерфейсом, несколько отличающимся от `str` (глава 7).
- Появился новый дополнительный способ форматирования строк, так `"See {0}, {1} and {foo}".format("A", "B", foo="C")` вернет результат `"See A, B and C"` (глава 7).
- Метод словаря `D.has_key(X)` будет удален. Вместо него следует использовать проверку на членство `X in D` (главы 4 и 8).
- Сравнение (при сортировке) смешанных нечисловых типов вместо использования текущей реализации механизма упорядочения будет вызывать исключение (главы 8 и 9).
- Методы словаря `.keys()`, `.items()` и `.values()` вместо списков будут возвращать «представления» объектов, поддерживающие возможность выполнения итераций. Чтобы вернуться к прежней логике выполнения, следует выполнять принудительное преобразование с помощью `list()` (глава 8).
- Согласно предыдущему пункту следующий шаблон программирования будет неработоспособен: `k = D.keys(); k.sort();` вместо него следует использовать `k = sorted(D)` (главы 4 и 8).
- Встроенная функция `file()` может быть удалена. Вместо нее следует использовать функцию `open()` (глава 9).

- Встроенная функция `raw_input()` будет переименована в `input()`. Чтобы обеспечить поддержку логики работы нынешней функции `input()`, следует использовать `eval(input())` (глава 10).
- Инструкция выполнения строки программного кода `exec` снова станет встроенной функцией (глава 10).
- Появятся новые зарезервированные слова `as`, `with` и `nonlocal`. Согласно предыдущему пункту `exec` перестанет быть зарезервированным словом (глава 11).
- Инструкция печати станет функцией, поддерживающей большое число возможностей. Вместо `print x, y` следует использовать `print(x, y)`, а также можно использовать новые ключевые аргументы функции для настройки параметров печати: `file=sys.stdout`, `sep=" "` и `end="\n"` (глава 11).
- Появится расширенная реализация операции распаковки: инструкция, поддерживающая универсальную форму присваивания последовательности, такая как `a, b, *rest = some_sequence`, теперь будет работать, как и `*rest, a = stuff`. Таким образом, число элементов слева и справа от инструкции присваивания больше не должно совпадать (глава 11).
- Автоматический режим распаковки кортежей через присваивание последовательности для функций будет недоступен. Больше нельзя будет записать `def foo(a, (b, c)):`, вместо этого необходимо будет выполнять явное присваивание последовательности: `def foo(a, bc): b, c = bc` (глава 11).
- Встроенная функция `xrange()` будет переименована в `range()`. То есть в этой версии Python будет существовать только функция `range()` (глава 13).
- В протоколе итераций метод `X.next()` будет переименован в `X.__next__()`, и появится новая встроенная функция `next(X)`, которая будет вызывать метод `X.__next__()` объекта (главы 13 и 17).
- Встроенные функции `zip()`, `map()` и `filter()` будут возвращать итераторы. Чтобы вернуться к логике использования списков, следует применять `list()` (главы 13 и 17).
- Функции могут включать необязательные комментарии, описывающие аргументы и результаты: так, в результате объявления `def foo(x: "spam", y: list(range(3))) -> 42*2:` к объекту-функции на этапе времени выполнения будет присоединен атрибут-словарь `foo.func_annotations: {'x': "spam", 'y': [0, 1, 2], "return": 84}` (глава 15).
- Новая инструкция `nonlocal x, y` позволит выполнять присваивание переменным в области видимости функции (глава 16).
- Функция `apply(func, args, kws)` будет удалена. Вместо нее следует использовать синтаксическую конструкцию `func(*args, **kws)` (главы 16 и 17).

- Встроенная функция `reduce()` будет удалена. Вместо нее организуйте циклы, как показано в этой книге; `lambda`, `map()` и `filter()` будут сохранены в версии 3.0 (глава 17).
- Все импортирование по умолчанию будет выполняться по абсолютному пути, а собственный каталог пакета будет пропускаться: для выполнения импорта по относительному пути, как это делается сейчас, следует использовать синтаксическую конструкцию `from . import` (глава 21).
- Все классы будут классами нового стиля и будут поддерживать современные новые расширения (глава 26).
- Наследование `class Spam(object)`, необходимое для создания нынешних классов, будет необязательным для классов. В версии 3.0 и нынешняя «классика», и классы «нового стиля» будут относиться к тому, что сейчас называется классами нового стиля (глава 26).
- В инструкции `try` конструкция `except name` превратится в конструкцию `except name as value` (глава 27).
- В инструкции `raise` конструкция `raise E, V` должна будет записываться как `raise E(V)`, то есть экземпляр исключения должен создаваться явно (глава 27).
- Включен оператор контекста исключений `with/as`, описываемый в этой книге (глава 27).
- Все пользовательские и встроенные исключения описываются классами, а не строками (глава 28).
- Все исключения, определяемые пользователем, должны наследовать встроенный `BaseException` – корневой класс иерархии классов исключений (`Exception` – это его подкласс и его вполне можно использовать в качестве базового для организации своей иерархии). Встроенный класс `StandardException` будет ликвидирован (глава 28).
- Структура пакета стандартной библиотеки может быть существенно изменена (подробности – в примечаниях к выпуску Python 3.0).

Список нововведений может показаться устрашающим на первый взгляд, однако не следует забывать, что основы языка Python, описываемые в этой книге, останутся неизменными и в версии 3.0. В действительности большая часть из того, что перечислено выше, будет оказывать на программистов не слишком большое влияние, если это влияние вообще будет ощущаться.

Кроме того, следует отметить, что этот список по-прежнему остается достаточно гипотетическим и в конечном счете может оказаться неполным и неточным, поэтому за официальной информацией следует ознакомиться с примечаниями к выпуску Python 3.0. Если вы пишете программный код для линейки Python 2.0, обратите внимание на сценарий «2to3», который выполняет автоматическое преобразование программного кода, написанного для интерпретатора версии 2.x в программный

код для интерпретатора версии 3.0. Этот сценарий будет поставляться в составе Python 3.0.

Об этой серии

Книги издательства O'Reilly из серии «Изучаем...» предназначены для тех, кто хочет приобрести новые знания и предпочитает структурный подход к изучению. Каждая книга из этой серии использует принципы обучения, которые мы (не без вашей помощи) выработали для передачи знаний, необходимых вам, чтобы присоединиться к новому проекту, справиться с неожиданным заданием или быстро изучить новый язык программирования.

Чтобы получить максимум пользы от любой книги из серии «Изучаем...», мы рекомендуем прорабатывать главы последовательно. Вы увидите, что усвоение материала идет быстрее, если внимательно читать указания и рекомендации, включенные в них. Вы можете также использовать заключительные разделы для предварительного знакомства с ключевыми аспектами каждой главы и с тем, что вам предстоит изучать. Наконец, чтобы помочь вам проверить степень усвоения материала, каждая глава завершается разделом «Закрепление пройденного», который включает короткие контрольные вопросы. Кроме того, каждая часть включает практические упражнения.

Книги серии «Изучаем...» работают с вами как надежный и проверенный коллега или преподаватель; мы стремимся сделать ваше обучение как можно более приятным. Свои отзывы о нашей работе и предложения по улучшению можете направлять по адресу learning@oreilly.com.

Использование программного кода примеров

Данная книга призвана оказать вам помощь в решении ваших задач. Вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам *необходимо* получить разрешение от издательства O'Reilly. 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 документацию вам *необходимо* будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Learning Python, by Mark Lutz. Copyright 2008 O'Reilly Media, Inc., 978-0-596-51398-6».

За получением разрешения на использование значительных объемов программного 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 имен модулей, методов, инструкций и команд.

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 комментариях.

<Моноширинный шрифт>

Таким способом выделяются синтаксические элементы, которые должны замещаться действительным программным кодом.



Так выделяются советы, предложения или примечания общего характера, имеющие отношение к расположенному рядом тексту.



Так выделяются предупреждения или предостережения, имеющие отношение к расположенному рядом тексту.

В примерах этой книги символ % в начале системной командной строки обозначает приглашение к вводу независимо от того, какое приглашение используется на вашей машине (например, C:\Python25> в окне Dos). Вам не нужно вводить символ %. Точно также в листингах, отображающих сеанс работы с интерпретатором, не нужно вводить символы >>> и ..., которые показаны в начале строки, — это приглашения к вводу, которые выводятся интерпретатором Python. Вводите лишь текст, который находится сразу же за этими приглашениями. Чтобы помочь вам запомнить это правило, все, что должно вводиться пользователем, выделено жирным шрифтом. Кроме того, обычно не требуется вводить текст в листингах, начинающийся с символа #, так как это комментарии, а не исполняемый программный код.

Safari® Books Online



Если на обложке технической книги есть пиктограмма «Safari® Books Online», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи лучших технических книг, вырезать и вставлять примеры кода, загружать главы и находить быстрые ответы, когда требуется наиболее верная и свежая информация. Она свободно доступна по адресу <http://safari.oreilly.com>.

Как с нами связаться

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

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/9780596513986>

Свои пожелания и вопросы технического характера отправляйте по адресу:

bookquestions@oreilly.com

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

<http://www.oreilly.com>

Обновления и дополнения к книге вы также можете найти на сайтах, упоминавшихся выше в этом предисловии.

Благодарности

Учитывая, что я работаю уже над третьим изданием этой книги, я не могу не пребывать в настроении, что «сложное задание выполнено». Я использовал и пропагандировал Python на протяжении 15 лет и обучал этому языку 10 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, какими путями он будет развиваться. Но, чтобы не выглядеть безнадежно эго-

центричным, я хотел бы немного вспомнить о прошлом и сказать несколько слов благодарности.

Это была длинная и извилистая дорога. Огладываясь назад, когда в 1992 году я открыл для себя Python, я предположить не мог, какое влияние он будет оказывать на мою жизнь в следующие 15 лет. Через два года после начала работы над первым изданием «Learning Python» в 1995 году я начал путешествовать по стране и миру, обучая начинающих программистов этому языку программирования. После выхода первого издания в 1999 году преподавание Python и работа над книгами стали моей основной работой, во многом благодаря интенсивному росту популярности Python.

В середине 2007 года, когда я пишу эти слова, мною уже написано девять книг о Python; опыт преподавания Python насчитывает более 10 лет. Проведя более 200 курсов в США, Европе, Канаде и Мексике, я обучил более 3000 студентов. Помимо множества мучительных часов, проведенных в самолетах, эти курсы дали мне возможность существенно улучшить эту и другие книги о Python. За эти годы преподавание помогало улучшать качество книг, а работа над книгами – качество преподавания. Фактически 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 10 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, которые совершают одни и те же ошибки, свойственные начинающим программистам!) Это издание стало возможным в первую очередь благодаря тем, кто посещал мои курсы после 2003 года, и тем не менее, все, кто посещал мои курсы, начиная с 1997 года, так или иначе помогли в улучшении этой книги. Я особенно хотел бы поблагодарить компании, предоставившие помещения для проведения курсов в Дублине, Мехико, Барселоне, Лондоне, Эдмонтоне 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: Татьяне Апанди (Tatiana Apandi), работавшей над этим изданием, и многим другим редакторам, работавшим над предыдущими изданиями. Лизе Дейли (Liza Daly) за участие в техническом обзоре этой книги. И издательству O'Reilly, за то что я получил шанс работать над этими девятью 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 сожалению, убрали из книги, чтобы освободить место для новых материалов об основах языка Python. В это издание

я добавил больше усложненных программ и заключительное упражнение для самостоятельного изучения, но это не компенсирует все то, что пришлось вырезать. Если вам не хватает этого материала, прочитайте ранее приведенные в предисловии примечания по поводу книг, описывающих вопросы прикладного программирования.

За создание такого замечательного и полезного языка я должен поблагодарить Гвидо ван Россума (Guido van Rossum) и все сообщество разработчиков и пользователей Python. Подобно большинству программных продуктов с открытыми исходными текстами Python развивается благодаря героическим усилиям многих программистов. Обладая 15-летним опытом программирования на Python, я по-прежнему нахожу его серьезной забавой. Мне очень повезло, что я смог наблюдать, как из маленького младенца в семействе языков сценариев Python 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 поблагодарить и поздравить с достигнутыми успехами все сообщество Python.

Я также хотел бы поблагодарить своего первого редактора из издательства O'Reilly, Фрэнка Уиллисона (Frank Willison). Идея этой книги во многом принадлежит 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 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. Йону Стюарту (Jon Stewart) и Майклу Муру (Michael Moore) за патриотизм. И всем крупным корпорациям, с которыми мне приходилось иметь дело, за то, что напоминают мне о том, как это здорово – работать на самого себя.

Моим детям, Майку (Mike), Сэмми (Sammy) и Рокси (Roxu), за любую будущность, которую они выберут. 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 (Vera), 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 (Mark Lutz)
Бертуд, Колорадо
Июль 2007

I

Введение

1

Python в вопросах и ответах

Если вы купили эту книгу, вы, скорее всего, уже знаете, что такое Python и насколько важно овладеть этим инструментом. Если это не так, вы наверняка не сможете зарабатывать на Python, пока не изучите язык, прочитав оставшуюся часть этой книги, и не напишете пару проектов. Но прежде чем мы приступим к изучению деталей, давайте сначала рассмотрим основные причины высокой популярности языка Python. Перед тем как приступить собственно к языку Python, в этой главе рассматриваются некоторые вопросы, которые обычно задают начинающие программисты, и даются ответы на них.

Почему программисты используют Python?

Это самый типичный вопрос, который задают начинающие программисты, потому что на сегодняшний день существует масса других языков программирования. Учитывая, что число пользователей Python составляет порядка 1 миллиона человек, достаточно сложно однозначно ответить на этот вопрос. Выбор средств разработки иногда зависит от уникальных особенностей и личных предпочтений.

Однако после обучения примерно 200 групп и 3000 студентов за последние 10 лет у меня накопились некоторые общие мысли по этому поводу. Основные факторы, которые приводятся пользователями Python, примерно таковы:

Качество программного обеспечения

Для многих основное преимущество языка Python заключается в удобочитаемости, 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 обслуживание 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 создании этого кода. Кроме того, Python поддерживает самые современные механизмы многократного использования программного кода, каким является объектно-ориентированное программирование (ООП).

Высокая скорость разработки

По сравнению с компилирующими или строго типизированными языками, такими как C, C++ и Java, Python во много раз повышает производительность труда разработчика. Объем программного кода на языке Python обычно составляет треть или даже пятую часть эквивалентного программного кода на языке C++ или Java. Это означает меньший объем ввода с клавиатуры, 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, необходимые в некоторых других языках программирования, что еще больше увеличивает 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 Windows обычно заключается в простом копировании программного кода сценария 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 запуска программ и обработку каталогов, в языке Python реализованы переносимым способом.

Поддержка библиотек

В составе Python поставляется большое число собранных и переносимых функциональных возможностей, известных как *стандартная библиотека*. Эта библиотека предоставляет массу возможностей, востребованных в прикладных программах, начиная от поиска текста по шаблону и заканчивая 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 назвать инструменты создания веб-сайтов, программирование математических вычислений, доступ к последовательному порту, разработку игровых программ и многое другое. Например, расширение NumPy позиционируется как свободный и более мощный эквивалент системы программирования математических вычислений Mathlab.

Интеграция компонентов

Сценарии Python легко могут взаимодействовать с другими частями приложения благодаря различным механизмам интеграции.

Эта интеграция позволяет использовать Python для настройки и расширения функциональных 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 Java-компонентами, взаимодействовать с такими платформами, как COM и .NET, и производить обмен данными по сети с помощью таких протоколов, как SOAP, XML-RPC и CORBA. Python не является обособленным инструментом.

Удовольствие

Благодаря непринужденности языка Python и наличию встроенных инструментальных средств процесс программирования может приносить удовольствие. На первый взгляд это трудно назвать преимуществом, тем не менее, удовольствие, получаемое от работы, напрямую влияет на производительность труда.

Из всех перечисленных факторов наиболее существенными для пользователей являются первые два (качество и производительность).

Качество программного обеспечения

По своей природе Python имеет простой, удобочитаемый синтаксис и ясную модель программирования. Согласно лозунгу, выдвинутому на недавней конференции 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. Программистам, использующим язык Python, почти не приходится прибегать к справочным руководствам – это непротиворечивая система, на выходе которой получается профессиональный программный код.

Философия Python диктует использование минималистского подхода. Это означает, что даже при наличии нескольких вариантов решения задачи в этом языке обычно существует всего один очевидный путь, небольшое число менее очевидных альтернатив и несколько взаимосвязанных вариантов организации 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 простое лучше сложного.¹

¹ Чтобы получить более полное представление о философии Python, введите в командной 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. Аббревиатура EIBTI, происходящая от фразы «explicit is better than implicit» («явное лучше неявного»), превратилась в модное жаргонное словечко.

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 находится в центре внимания программистов.

Высокая скорость разработки

Во время бума развития Интернета во второй половине 1990-х годов, было сложно найти достаточное число программистов для реализации программных проектов – от разработчиков требовалось писать программы со скоростью развития Интернета. Теперь, в эпоху экономического спада, картина изменилась. Сегодня от программистов требуется умение решать те же задачи меньшим числом сотрудников.

В обоих этих случаях Python блистал как инструмент, позволяющий программистам получать большую отдачу при меньших усилиях. Он изначально оптимизирован для достижения *высокой скорости разработки* – простой синтаксис, 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 используется для создания сценариев. Обычно он определяется как *объектно-ориентированный язык сценариев* – такое определение смешивает поддержку ООП с общей ориентацией на сценарии. Действительно, для обозначения файлов с программным кодом на языке Python программисты часто используют слово «сценарий» вместо слова «программа». В этой книге термины «сценарий» и «программа» рассматриваются как взаимозаменяемые, с некоторым предпочтением термина «сценарий» для обозначения простейших программ, помещающихся в единственный файл, и термина «программа» для обозначения более сложных приложений, программный код которых размещается в нескольких файлах.

Термин «язык сценариев» имеет множество различных толкований. Некоторые 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 представляет 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 «связующий» слой, который используется для управления (то есть для описания сценария работы) другими прикладными компонентами. Программы на языке Python действительно нередко используются в составе крупных приложений. Например, при проверке аппаратных устройств программы на языке Python могут вызывать компоненты, осуществляющие низкоуровневый доступ к устройствам. Точно так же программы могут запускать программный код на языке Python для поддержки настройки программного продукта у конечного пользователя, что ликвидирует необходимость поставлять и пересобирать полный объем исходных текстов.

Простота языка Python делает его весьма гибким инструментом управления. Тем не менее, технически – это лишь одна из многих ролей, которые может играть Python. Многие программисты пишут на языке Python автономные 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 способствует применению зондирующего, поэтапного стиля программирования, который следует попробовать, чтобы оценить по достоинству.

Не надо заблуждаться, Python предназначен не только для решения простых задач. Скорее он упрощает решение задач, благодаря своей простоте и гибкости. Язык Python имеет небольшой набор возможностей, но он позволяет 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», вероят-

но, лучше использовать для описания быстрого и гибкого способа разработки, который поддерживается языком Python, а не для описания прикладной области программирования.

Все хорошо, но есть ли у него недостатки?

После 15 лет работы с языком Python и 10 лет преподавания единственный недостаток, который мне удалось обнаружить, – это скорость выполнения программ, которая не всегда может быть такой же высокой, как у программ, написанных 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 платформонезависимый формат. Однако, из-за того что Python не создает двоичный машинный код (например, машинные инструкции для микропроцессора Intel), некоторые 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 программ, зависит от того, какого рода программы вы пишете. Python многократно подвергался оптимизации, и в отдельных прикладных областях программный код на этом языке отличается достаточно высокой скоростью выполнения. Кроме того, когда в сценарии Python делается что-нибудь «значительное», например обрабатывается файл или конструируется графический интерфейс, ваша программа фактически выполняется со скоростью, которую способен дать язык C, потому что такого рода задачи решаются скомпилированным с языка C программным кодом, лежащим в недрах интерпретатора Python. Гораздо важнее, что преимущество в скорости разработки порой важнее потери скорости выполнения, особенно если учесть быстродействие современных компьютеров.

Тем не менее, даже при высоком быстродействии современных процессоров остаются такие области, где требуется максимальная скорость выполнения. Реализация математических вычислений и анимационных эффектов, например, часто требуют наличия базовых вычислительных компонентов, которые решают свои задачи со скоростью языка C (или еще быстрее). Если вы работаете как раз в такой области, вы все равно сможете использовать Python, достаточно лишь выделить из приложения компоненты, требующие максимальной скорости работы, в виде *скомпилированных 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 языка управления. Типичным примером такой двуязычной стратегии может служить расширение *NumPy*, содержащее реализацию математических вычислений для Python; благодаря комбинированию компилированных и оптимизированных 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 распоряжение мощный механизм оптимизации.

Кто в наше время использует Python?

К моменту, когда я пишу эти строки (2007 год), наиболее правдоподобной оценкой числа пользователей Python является число, близкое к 1 миллиону человек во всем мире (с небольшой погрешностью). Эта оценка основана на различных статистических показателях, таких как количество загрузок и результаты опросов разработчиков. Дать более точную оценку достаточно сложно, потому что Python является открытым программным обеспечением – для его использования не требуется проходить лицензирование. Более того, Python по умолчанию включается в состав дистрибутивов Linux, поставляется вместе с компьютерами Macintosh и некоторыми другими программными и аппаратными продуктами, что существенно искажает оценку числа пользователей.

Вообще же количество пользователей Python значительно больше и вокруг него сплотилось очень активное сообщество разработчиков. Благодаря тому что Python появился более 15 лет тому назад и получил широкое распространение, он отличается высокой стабильностью и надежностью. Python используется не только отдельными пользователями, он также применяется настоящими компаниями для создания продуктов, приносящих настоящую прибыль. Например:

- Компания Google широко использует Python в своей поисковой системе и оплачивает труд создателя Python.
- Служба коллективного использования видеоматериалов YouTube в значительной степени реализована на языке Python.
- Популярная программа BitTorrent для обмена файлами в пиринговых сетях (peer-to-peer) написана 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 пылесосов.
- ESRI использует Python в качестве инструмента настройки своих популярных 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. Фактически, с определенной долей уверенности можно утверждать, что Python так или иначе используется практически каждой достаточно крупной организацией, занимающейся разработкой программного обеспечения, – как для решения 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, обращайтесь на веб-сайт <http://www.python.org>.

Что можно делать с помощью Python?

Кроме того, будучи удачно спроектированным языком программирования, Python прекрасно подходит для решения реальных задач из разряда тех, которые разработчикам 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 программирования, 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 несколько широких категорий. Следующие несколько разделов описывают наиболее типичные области применения Python в наши дни, а также инструментальные средства, используемые в каждой из областей. У нас не будет возможности 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 *инструментами 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 многое другое.

Стандартная библиотека Python обеспечивает возможность связывания в соответствии с требованиями стандартов POSIX и поддерживает все типичные инструменты операционных систем: переменные окружения, файлы, сокеты, каналы, процессы, многопоточную модель исполнения, поиск по шаблону с использованием регулярных выражений, аргументы командной строки, стандартные интерфейсы доступа к потокам данных, запуск команд оболочки, дополнение имен файлов и многое другое. Кроме того, системные интерфейсы в языке Python созданы переносимыми, например, сценарий копирования дерева каталогов не требует внесения изменений, в какой бы операционной системе он ни использовался.

Графический интерфейс

Простота Python и высокая скорость разработки делают его отличным средством разработки графического интерфейса. В состав Python входит стандартный объектно-ориентированный интерфейс к Tk GUI API, который называется *Tkinter*, позволяющий программам на языке Python реализовать переносимый графический интерфейс с внешним видом, присущим операционной системе. Графические интерфейсы на базе Python/Tkinter без изменений могут использоваться в MS Windows, X Window (в операционных системах UNIX и Linux) и Mac OS (как в классической версии, так и в OS X). Свободно распространяемый пакет расширения *PMW* содержит дополнительные визуальные компоненты для набора Tkinter. Кроме того, существует 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 библиотеки вы также сможете использовать другие инструменты создания графического интерфейса, такие как Qt, GTK, MFC и Swing. Для разработки приложений с веб-интерфейсом или не предъявляющих высоких требований к интерфейсу можно использовать Jython (реализация Python на языке Java, описывается в главе 2) и CGI-сценарии, которые обеспечивают дополнительные возможности по созданию пользовательского интерфейса.

Веб-сценарии

Интерпретатор Python поставляется вместе со стандартными интернет-модулями, которые позволяют программам выполнять разнообразные сетевые операции как в режиме клиента, так и в режиме сервера. Сценарии могут производить взаимодействия через сокеты, извлекать информацию из форм, отправленных серверным CGI-сценариям; передавать файлы по протоколу FTP; обрабатывать файлы XML; передавать, принимать, создавать и производить разбор писем электронной почты; загружать веб-страницы с указанных адресов URL; производить разбор кода HTML и XML полученных веб-страниц; производить 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 создания 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 сценариев на языке Python под управлением веб-сервера Apache и поддерживает шаблоны механизма Python Server Pages. Система Jython обеспечивает беспровную интеграцию Python/Java и поддерживает серверные апплеты, которые выполняются на стороне клиента. Помимо этого для Python существуют полноценные пакеты веб-разработки, такие как Django, TurboGears, Pylons, Zope и WebWare, поддерживающие возможность быстрого создания полнофункциональных высококачественных веб-сайтов на языке Python.

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 системы на языке C и C++ делает его удобным и гибким языком для описания поведения 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 производить настройку программных продуктов без необходимости пересобирать эти продукты или поставлять их с исходными текстами.

Такие инструменты, как SWIG и SIP, автоматически генерирующие программный код, могут автоматизировать действия по связыванию скомпилированных компонентов в Python для последующего их использования в сценариях. А такие огромные платформы, как поддержка COM в MS Windows, Jython – реализация на языке Java, Iron-Python – реализация на базе .NET и разнообразные реализации CORBA, предоставляют альтернативные возможности организации взаимодействий с компонентами. Например, в операционной системе Windows

сценарии на языке Python могут использовать платформы управления такими приложениями, как MS Word 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 сценариев, 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 используемый 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 сторонних разработчиков, которая называется ZODB. Она представляет собой полностью объектно-ориентированную базу данных для использования в сценариях на языке Python. Существует еще один инструмент под названием SQLAlchemy, который отображает реляционные таблицы в модель классов языка Python. В 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, выглядят одинаково. Благодаря этому можно сначала создавать прототипы систем на языке Python, а затем переносить выбранные компоненты на компилирующие языки, такие как C и C++. В отличие от ряда других инструментов разработки прототипов, 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 ее сопровождение и использование.

Программирование математических и научных вычислений

Расширение NumPy для математических вычислений, упоминавшееся выше, включает такие мощные элементы, как объект-массив, интерфейсы к стандартным математическим библиотекам и многое другое. Расширение NumPy – за счет интеграции с математическими библиотеками, написанными на компилирующих языках программирования, – превращает Python в сложный, но удобный инструмент программирования математических вычислений, который зачастую может заменить

существующий программный код, написанный на традиционных компилирующих языках, таких как FORTRAN и C++. Дополнительные инструменты математических вычислений для Python поддерживают возможность создания анимационных эффектов и трехмерных объектов, позволяют организовать параллельные вычисления и т. д.

Игры, изображения, искусственный интеллект, XML-роботы и многое другое

Язык программирования Python можно использовать для решения более широкого круга задач, чем может быть упомянуто здесь. Например, с помощью системы *pygame* можно создавать игровые программы и анимационные ролики; обрабатывать изображения с помощью пакета *PIL* и других; управлять роботом с помощью пакета библиотек *xml*, модуля *xmlrpc.lib* и расширений сторонних разработчиков; программировать искусственный интеллект с помощью эмулятора нейросетей 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-сайтах Vaults of Parnassus и PyPI (ищите ссылки с помощью Google или на сайте <http://www.python.org>).

Вообще говоря, многие из этих областей применения Python – всего лишь разновидности одной и той же роли под названием «интеграция компонентов». Использование Python в качестве интерфейса к библиотекам компонентов, написанных на языке C, делает возможным создание сценариев на языке Python для решения задач в самых разных прикладных областях. Как универсальный, многоцелевой язык программирования, поддерживающий возможность интеграции, Python может применяться очень широко.

В чем сильные стороны Python?

Естественно – это вопрос разработчика. Если у вас еще нет опыта программирования, язык следующих нескольких разделов может показаться немного непонятным, но не волнуйтесь, мы будем рассматривать все эти термины позднее, в ходе изложения материала. А для разработчиков ниже приводится краткое введение в некоторые особенности языка Python.

Он объектно-ориентированный

Python изначально является объектно-ориентированным языком программирования. Его объектная модель поддерживает такие понятия, как полиморфизм, перегрузка операторов и множественное наследование, однако, учитывая простоту синтаксиса и типизации Python, ООП не вызывает сложностей в применении. Если эти термины вам непо-

няты, 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 наличии соответствующего связующего программного кода программы на языке Python могут использовать механизм наследования от классов, реализованных на C++, Java и C#.

Как бы то ни было, но ООП *не является обязательным* в Python; вы сможете 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 тактических 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 тексты реализации Python. Нет никаких ограничений на его копирование, встраивание в свои системы или распространение в составе ваших продуктов. Фактически 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 скоростью, которой могли бы позавидовать большинство разработчиков коммерческих продуктов. Кроме того, свободное распространение кода способствует расширению команды экспертов по реализации. И хотя предоставляемая возможность изучать или изменять реализацию языка 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.

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. Изменения в языке принимаются только после прохождения формальной процедуры (известной как «программа совершенствования продукта», или PEP) и тщательно проверяются формальной системой тестирования и самим Пожизненным Диктатором. Это обеспечивает большую степень консерватизма Python в отношении изменений по сравнению с некоторыми другими языками программирования.

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 самом широком спектре устройств, начиная от наладонных компьютеров (PDA) и заканчивая суперкомпьютерами. Ниже приводится далеко не полный список операционных систем и устройств, где можно использовать Python:

- Операционные системы Linux и UNIX
- Microsoft Windows и DOS (все современные версии)
- Mac OS (обе разновидности: OS X и Classic)
- BeOS, OS/2, VMS и QNX
- Системы реального времени, такие как VxWorks
- 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 Windows 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 переносимый байт-код, который одинаково хорошо работает на любых платформах, где установлена совместимая версия Python (подробнее об этом будет рассказываться 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 Windows, и в любых других операционных системах, где установлен интерпретатор Python. В большинстве реализаций Python под определенные операционные системы имеется также поддержка специфических механизмов этих систем (например, поддержка COM в Windows), но ядро языка Python и библиотеки работают совершенно одинаково в любой системе. Как уже говорилось выше, Python включает в себя средства создания графического интерфейса Tk GUI под названием Tkinter, что позволяет программам на языке Python создавать графический интерфейс, совместимый со всеми основными графическими платформами без индивидуальной программной настройки.

Он мощный

С точки зрения функциональных возможностей Python можно назвать гибридом. Его инструментальные средства укладываются в диапазон

между традиционными языками сценариев (такими как Tcl, Scheme и Perl) и языками разработки программных систем (такими как C, C++ и Java). Python обеспечивает простоту и непринужденность языка сценариев и мощь, которую обычно можно найти в компилирующих языках. Превышая возможности других языков сценариев, такая комбинация делает Python удобным средством разработки крупномасштабных проектов. Для предварительного ознакомления ниже приводится список основных возможностей, которые есть в арсенале Python:

Динамическая типизация

Python сам следит за типами объектов, используемых в программе, благодаря чему не требуется писать длинные и сложные объявления в программном коде. В действительности, как вы увидите в главе 6, в языке Python вообще отсутствуют понятие типа и необходимость объявления переменных. 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 низкоуровневые 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 составляющие, применять ООП для создания программного кода многократного 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 высокой гибкостью и удобством. Например, встроенные объекты могут расширяться 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 конкатенация (объединения коллекций), получение срезов (извлечение части коллекции), сортировка, отображение и многое другое.

Библиотеки утилит

Для выполнения более узких задач в состав Python также входит большая коллекция библиотечных инструментов, которые поддерживают практически все, что только может потребоваться, – от поиска с использованием регулярных 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 работы с изображениями, распределенных объектов CORBA, XML, механизмов доступа к базам данных и многое другое.

Несмотря на широкие возможности, Python имеет чрезвычайно простой синтаксис и архитектуру. В результате мы имеем мощный инструмент программирования, обладающий простотой и удобством, присущими языкам сценариев.

Он соединяемый

Программы на языке Python с легкостью могут «склеиваться» с компонентами, написанными на других языках программирования. Например, прикладной интерфейс С API в Python позволяет программам на языке С 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 языках, как С или С++, например, превращает его в удобный язык для создания интерфейсов к этим библиотекам и в средство настройки программных продуктов. Как уже говорилось выше, все это делает Python прекрасным средством 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 С.

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, подобных С или С++. Интерпретатор Python немедленно выполняет программу, что позволяет программировать в интерактивном режиме и получать результаты сразу же после внесения изменений – в большинстве случаев вы сможете наблюдать

эффект 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 «исполняемым 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!

Программирование на языке Python – это технический процесс, а не искусство

Когда в начале 1990 годов Python впервые появился на сцене программного 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 – разработчики достаточно умны, чтобы самостоятельно сделать выводы. Однако в моей преподавательской практике мне часто приходится слышать вопросы на эту тему, поэтому я считаю необходимым сказать несколько слов по этому поводу.

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 писать, но сложно читать. Учитывая, что период жизни большинства программ длится много дольше, чем период их создания, многие усматривают в Python более эффективный инструмент программирования.

Вообще говоря, история этого противоречия отражает опыт проектировщиков двух языков программирования и подчеркивает некоторые из основных причин, по которым программисты отдают предпочтение языку Python. Создатель языка Python – математик по образованию, и потому он создал язык, обладающий высокой степенью однородности – его синтаксис и набор возможностей отличаются удивительной согласованностью. Более того, если говорить математическими терминами, язык Python обладает ортогональной архитектурой – большая часть выразительных возможностей языка следует из небольшого числа базовых концепций. Как только программист схватывает суть полиморфизма в Python, все остальное становится для него мелочью.

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 взаимодействуют между собой контекстно-зависимым, порой трудноуловимым способом, 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 выражения мыслей при создании программного кода. Программный код одного программиста может радикально отличаться от программного кода другого. И правда, искусство создания уникального хитросплетения инструкций всегда было предметом гордости программистов на Perl.

Однако любой, кто когда-либо занимался сопровождением программного кода, скажет вам, что свобода выражения хороша для искусства, но не для технологического процесса. В технологии нам требуется минимальный набор возможностей и высокая степень предсказуемости. Свобода выражения мыслей в технологии может превратить процесс сопровождения 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. Когда программист пишет сценарий, он пишет его не для себя самого. Более того, сценарий пишется даже не для компьютера. Хороший программист знает, что свой программный код он пишет для другого человека, который будет вынужден читать его в ходе сопровождения и использования. Если этот человек не сможет понять сценарий, он станет практически бесполезным в реальной жизни.

Многие находят в этом самое четкое отличие Python от других языков сценариев, подобных языку Perl. Синтаксическая модель Python вынуждает пользователя писать удобочитаемый программный код, поэтому программы на языке Python лучше вписываются в полный цикл разработки программного обеспечения. А такие свойства Python, как ограниченное число способов взаимодействия, единообразие, закономерность и непротиворечивость способствуют появлению программного 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 удовлетворенности. Программисты, использующие язык Python, могут быть не менее творческими натурами и, как мы увидим позднее, в некоторых случаях 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. Вы всегда должны сами оценивать подобные суждения при изучении Python. А приступить к изучению мы сможем в следующей главе.

Он прост в изучении

Это самый важный аспект данной книги: по сравнению с другими языками программирования базовый язык Python очень легко запоминается. В действительности 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, стремящихся 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 инструменты программирования, основа языка по-прежнему остается простой для изучения как начинающими, так и опытными программистами.

Он назван в честь Монти Пайтона (Monty Python)

Это не имеет отношения к технической стороне дела, но похоже, что эта тайна, которую я собираюсь открыть, на удивление хорошо охраняется. Несмотря на то что на эмблеме Python изображена рептилия, правда в том, что создатель Python, Гвидо ван Россум (Guido van Rossum), назвал свое детище в честь комедийного сериала «Летающий цирк Монти Пайтона» (Monty Python's Flying Circus), который транслировался по телеканалу BBC. Он большой поклонник Монти Пайтона (Monty Python), как и многие программисты (похоже, что есть какая-то аналогия между разработкой программ и цирком).

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 «egg». Встречающиеся иногда имена «Brian», «ni» и «shrubbery», точно также появились благодаря своим тезкам. Это даже оказывает влияние на сообщество в целом: дискуссии на конференциях по языку Python обычно имеют именование «Испанская инквизиция».

Все это, конечно, очень забавно, если вы знакомы с сериалом, в противном случае это кажется непонятным. Вам не требуется знать сериал, чтобы понимать примеры, где используются ссылки на Монти Пайтона (включая многие примеры в этой книге), но по крайней мере, вы теперь знаете, откуда что берется.

Какими преимуществами обладает Python перед языком X?

Наконец, перейдем к области, которая вам уже, возможно, знакома – сравнению Python с другими языками программирования, такими как Perl, Tcl и Java. Ранее мы уже говорили о проблеме производительности, поэтому здесь мы сосредоточим свое внимание на функциональных возможностях. Другие языки программирования также являются достаточно полезными инструментами, чтобы знать и использовать их, но многие находят, 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 унаследовала 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 зрелый и имеет более ясный синтаксис, чем Ruby. В отличие от Ruby и Java, объектно-ориентированный стиль программирования является необязательным в Python – он не вынуждает использовать ООП в проектах, где этот стиль неприменим.
- Обладает динамическими характеристиками таких языков, как SmallTalk и Lisp, но имеет более простой и традиционный синтак-

сис, доступный как для разработчиков, так и для конечных пользователей настраиваемых систем.

Многие считают, что Python, по сравнению с другими современными языками сценариев, гораздо лучше подходит для программ, которые делают нечто большее, 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 сопровождать.

Безусловно, автор является горячим поклонником Python с 1992 года, поэтому воспринимайте эти комментарии по своему усмотрению. Однако они в действительности 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 программирования своих задач. Здесь также было показано, как он используется, и приведены 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 Python, 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, достаточный для запуска примеров и выполнения упражнений в остальной части книги. Мы фактически не будем касаться вопросов программирования до главы 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 контрольных вопросов, которые помогут вам закрепить в памяти ключевые концепции. Ответы на вопросы следуют ниже, и вы можете прочитать эти ответы сразу, как только столкнетесь с затруднениями. Помимо контрольных вопросов в конце каждой части вы найдете

упражнения, предназначенные для того, чтобы помочь вам программировать на языке Python. Итак, перед вами первый 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 «`sram`» так часто используется в примерах программного кода на языке Python?
7. Какой ваш любимый цвет?

Ответы

Как дела? Ниже приводятся ответы, которые подготовил я, хотя на некоторые вопросы существует несколько правильных ответов. Напомню еще раз, даже если вы абсолютно уверены в правильности своих ответов, я советую прочитать мои ответы хотя бы ради того, чтобы получить некоторые дополнительные сведения. Если мои ответы кажутся вам бессмысленными, прочитайте текст главы еще раз.

1. Качество программного обеспечения, скорость разработки, переносимость программ, поддержка библиотек, интеграция компонентов и удовольствие. Из этих шести причин качество и скорость разработки являются наиболее существенными при выборе Python.
2. Google, Industrial Light & Magic, Jet Propulsion Labs, 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 тестирование и системное администрирование.
3. Основным недостатком Python является невысокая производительность, программы на языке Python не могут выполняться так же быстро, как программы на полностью компилируемых языках, таких как C и C++. С другой стороны, для большинства применений он обладает достаточно высокой скоростью выполнения и обычно программный код на языке Python работает 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, лежащие в основе языка. Как запустить эту инструкцию, вы узнаете в следующей главе.
6. Слово «spam» взято из известной пародии Монти Пайтона (Monty Python), где герои сериала пытаются заказать блюдо в кафетерии, а их заглушает хор викингов, поющих о консервах (spam). Ах да! Это еще и типичное имя переменной, которое используется в сценариях на языке Python...
7. Голубой. Нет, желтый!

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 вам следует запускать свои 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 приходилось использовать подобные инструменты в прошлом и которые стремятся побыстрее добраться до самого языка, могут пропустить эти главы, оставив их «для ознакомления в будущем». А со всеми остальными мы попробуем разобраться, 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 пакет программного обеспечения, который называется *интерпретатор*. Интерпретатор – это такая программа, которая выполняет другие программы. Когда вы пишете программу на языке Python, интерпретатор Python читает вашу программу и выполняет составляющие ее инструкции. В действительности, интерпретатор – это слой программной логики между вашим программным кодом и аппаратурой вашего компьютера.

В процессе установки пакета Python на компьютер создается ряд программных компонентов – как минимум, интерпретатор и библиотека поддержки. В зависимости от порядка использования интерпретатор Python может иметь вид исполняемой программы или набора библио-

тек, связанных с другой программой. В зависимости от используемой версии Python сам интерпретатор может быть реализован как программа на языке C, как набор классов Java и в каком-либо другом виде. Независимо от используемой вами разновидности Python ваш программный код на этом языке всегда будет выполняться этим интерпретатором. А чтобы обеспечить такую возможность, вы должны установить интерпретатор Python на свой компьютер.

Процедура установки Python отличается для разных платформ и подробно описывается в приложении А. В двух словах:

- Пользователи Windows должны получить и запустить инсталляционный исполняемый файл, который произведет установку Python на компьютер. Для этого нужно просто дважды щелкнуть на инсталляционном файле и отвечать Yes (Да) или Next (Далее) на все вопросы.
- В Windows Vista вам может потребоваться выполнить дополнительные действия с инсталляционным файлом Python 2.5 MSI. За подробной информацией по этому поводу обращайтесь к приложению А.
- 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 в большинстве версий UNIX) Python обычно устанавливается из файлов RPM или компилируется из исходных текстов.
- Процедура установки на других платформах зависит от этих платформ. Например, Python присутствует также в сотовых телефонах, игровых консолях и в проигрывателе iPod, но процедуры установки Python на эти устройства 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 приступить к установке, вы должны убедиться, что Python не был ранее установлен на ваш компьютер. Если вы пользуетесь операционной системой Windows, обычно Python можно найти в меню Start (Пуск), как показано на рис. 2.1 (эти пункты меню будут рассматриваться 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 обращайтесь к приложению А. В целях этой и следующей главы я буду исходить из предположения, что Python уже установлен и готов к работе.

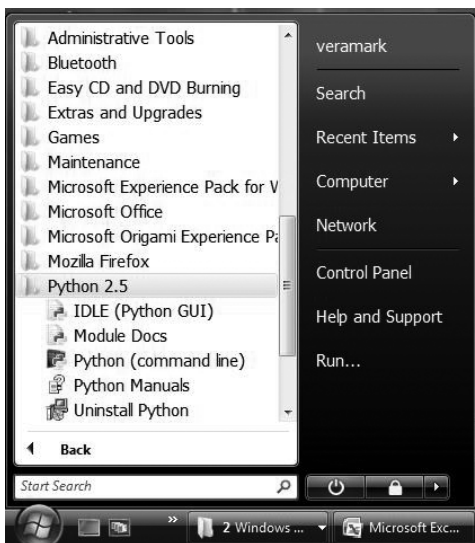


Рис. 2.1. Когда Python установлен в Windows, его можно найти в меню Start (Пуск). Набор пунктов меню может немного отличаться, в зависимости от версии, но пункт IDLE запускает среду разработки с графическим интерфейсом, а пункт Python запускает сеанс работы с интерактивной командной оболочкой. Кроме того, здесь же можно увидеть пункты вызова стандартного справочного руководства и запуска механизма документирования Pycos (пункт Module Docs)

Выполнение программы

Что стоит за словами «написать и запустить программу на языке Python», зависит от того, как вы смотрите на эту задачу, – как программист или как интерпретатор Python. Обе точки зрения определяют свой взгляд на программирование.

С точки зрения программиста

Программа на языке Python, в самой простой форме, – это обычный текстовый файл, содержащий инструкции Python. Например, следующий файл, с именем *script1.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
```

Этот файл содержит две инструкции `print`, которые просто выводят строку (текст в кавычках) и результат числового выражения (2 в степени 100) в выходной поток. Не надо сейчас стараться вникнуть в син-

таксис языка – в этой главе нас интересует лишь сам порядок запуска программ. Позднее я расскажу 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 общепринятым 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 позднее в этой книге, но большинству файлов с программами на языке Python даются имена с расширением `.py` для единообразия.

После того как инструкции будут введены в текстовый файл, можно потребовать 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 другими 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
1267650600228229401496703205376km
```

Например, ниже показано, что происходило, когда я попытался запустить этот сценарий в командной строке DOS на ноутбуке, где установлена операционная система Windows (обычно эту программу можно найти в меню Accessories (Стандартные) под названием Command Prompt (Командная строка)), чтобы убедиться, что я не допустил никаких опечаток:

```
D:\temp> python script1.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 запускаете этот файл с помощью интерпретатора. Однако, когда вы говорите интерпретатору «вперед», за кулисами много чего происходит. Хотя знание внутреннего устройства Python и не требуется для овладения навыками программирования на этом языке, тем не менее понимание того, как производится выполнение 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 программный код начнет «скрипеть колесами». В частности, сценарий сначала будет скомпилирован в нечто под названием «байт-код», а затем передан некоторому механизму, известному под названием «виртуальная машина».

Компиляция в байт-код

Когда вы запускаете программу, практически незаметно для вас Python сначала компилирует ваш *исходный текст* (инструкции в файле) в формат, известный под названием *байт-код*. Компиляция – это просто этап перевода программы, а байт-код – это низкоуровневое, платформонезависимое представление исходного текста программы. 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 параграфе вы могли заметить фразу – *практически* незаметно для вас. Если интерпретатор Python на вашем компьютере обладает правом записи, он будет сохранять байт-код вашей программы в виде файла с расширением *.pyc* («.рус» – это скомпилированный исходный файл «.ру»). Вы будете обнаруживать эти файлы после запуска программ по соседству с файлами, содержащими исходные тексты (то есть в том же каталоге).

Интерпретатор сохраняет байт-код для ускорения запуска программ. В следующий раз, когда вы попытаетесь запустить свою программу, Python загрузит файл *.pyc* и минует этап компиляции – при условии, что исходный текст программы 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 Python окажется не в состоянии записать файл с байт-кодом на диск, программа от этого не пострадает, просто байт-код будет сгенерирован в памяти и уничтожен по завершении программы.¹ Однако, поскольку файлы *.pyc* повышают скорость запуска программы, вам может потребоваться иметь возможность сохранять их, особенно для больших программ. Кроме того, файлы с байт-кодом – это еще один из

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 интерактивном режиме, который описывается в главе 3.

способов распространения программ на языке Python. Интерпретатор запустит файл *.pyc*, даже если нет оригинальных файлов с исходными текстами. (В разделе «Фиксированные двоичные файлы» описывается 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 универсальному механизму под названием виртуальная машина Python (PVM – для любителей аббревиатур). Аббревиатура PVM выглядит более внушительно, чем то, что за ней стоит на самом деле, – это не отдельная программа, которую требуется устанавливать. Фактически, PVM – это просто большой цикл, который выполняет перебор инструкций 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 программистов. Компиляция в байт-код производится автоматически, а PVM – это всего лишь часть системы Python, которую вы установили на свой компьютер. Повторю еще раз, что программисты просто создают программный код на языке Python и запускают файлы с инструкциями.

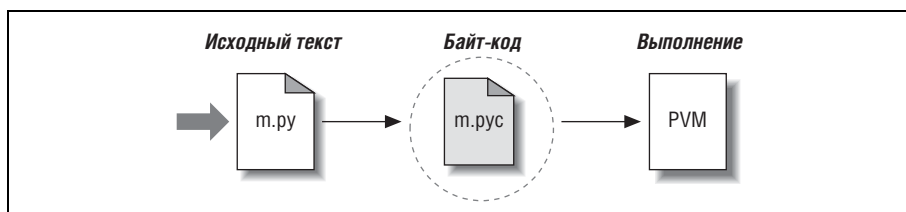


Рис. 2.2. Традиционная модель выполнения программ на языке Python: исходный текст, который вводится программистом, транслируется в байт-код, который затем исполняется виртуальной машиной Python. Исходный текст автоматически компилируется и затем интерпретируется

Производительность

Читатели, имеющие опыт работы с компилирующими языками программирования, такими как C и C++, могут заметить несколько отличий в модели выполнения 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 машинным 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 говорилось в главе 1. Обход инструкций выполняет виртуальная машина, а не микропроцессор, и чтобы выполнить байт-код, необходима дополнительная интерпретация, инструкции которого требуют на выполнение больше времени, чем машинные инструкции микропроцессора. С другой стороны, в отличие от классических интерпретаторов, здесь присутствует дополнительный этап компиляции – интерпретатору не требуется всякий раз снова и снова анализировать инструкции исходного текста. В результате Python способен обеспечить скорость выполнения где-то между традиционными компилирующими и традиционными интерпретирующими языками программирования. Подробнее о проблеме производительности рассказывается в главе 1.

Скорость разработки

С другой стороны, в модели выполнения Python отсутствуют различия между средой разработки и средой выполнения. То есть системы, которые компилируют и выполняют исходный текст, – это суть одна и та же система. Для читателей, 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, выполняющего программы.

Это существенно увеличивает скорость разработки. Не нужно всякий раз выполнять компиляцию и связывание программ, прежде чем запустить их, – вы просто вводите исходный текст и запускаете его. Это также придает языку некоторый 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. Благодаря такой возможности Python может использоваться для настройки продуктов – программный код Python может изменяться «на лету», а пользователи могут изменять части системы, написанные на языке Python, без необходимости перекомпилировать систему целиком.

Имейте в виду, 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 языках происходят перед выполнением, но в программах на языке Python происходят во время выполнения. В результате процесс программирования приобретает больший динамизм, чем тот, к которому привыкли некоторые читатели.

Разновидности модели выполнения

Прежде чем двинуться дальше, я должен заметить, что внутренний поток выполнения, описанный в предыдущем разделе, отражает современную стандартную 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 альтернативные реализации языка Python – *CPython*, *Jython* и *IronPython*, а также несколько второстепенных реализаций, таких как *Stackless Python*. В двух словах: 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.

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 обнаружили у себя предварительно установленную версию Python, то более чем вероятно, что 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, работает быстрее, устойчивее и лучше, чем альтернативные системы. Рисунок 2.2 отражает модель выполнения CPython.

Jython

Интерпретатор Jython (первоначальное название – JPython) – это альтернативная 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 расположенных 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 интеграцию с Java. Поскольку программный код на языке Python транслируется в байт-код Java, во время выполнения он ведет себя точно так же, как настоящая программа на языке Java. Сценарии на языке Jython могут выступать в качестве апплетов и сервлетов, создавать графический интерфейс с использованием механизмов Java и т. д. Более того, Jython обеспечивает поддержку 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, которым необходим язык сценариев в качестве интерфейса к Java-коду.

IronPython

Третья (и, к моменту написания этих строк, самая новая) реализация языка Python – это IronPython. Она предназначена для обеспечения интеграции программ Python с приложениями, созданными для работы в среде Microsoft .NET Framework операционной системы Windows, а также в Mono – открытом эквиваленте для Linux. Платформа .NET и среда выполнения языка C# предназначены для обеспечения взаимодействий между программными объектами – независимо от используемого языка программирования, в духе более ранней модели COM компании Microsoft. Реализация IronPython позволяет программам на языке Python играть роль как клиентских, так и серверных компонентов, доступных из других языков программирования .NET.

Модель реализации IronPython очень напоминает Jython (и фактически разрабатывается одним и тем же автором) – она подменяет два этапа на рис. 2.2 справа на эквиваленты среды .NET. Как и Jython, основной интерес IronPython представляет для разработчиков, которым необходима интеграция Python с компонентами .NET. Поскольку разработка ведется компанией Microsoft, от IronPython, помимо прочего, можно было бы еще ожидать существенной 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

¹ Jython и IronPython – это полностью независимые реализации языка Python, которые компилируют исходный программный код для различных архитектур времени выполнения. Из программ для CPython также можно получить доступ к программным компонентам Java и .NET: например, системы JPure и Python для .NET позволяют коду, исполняемому интерпретатором CPython, обращаться к компонентам Java и .NET.

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 динамический компилятор Psycodo и транслятор Shedskin C++, которые пытаются оптимизировать основную модель выполнения. Знание этих реализаций пока не является для вас обязательным, тем не менее краткий обзор их реализации модели выполнения поможет пролить свет на модель выполнения в целом.

Динамический компилятор Psycodo

Система Psycodo – это не другая реализация языка Python, а компонент, расширяющий модель выполнения байт-кода, что позволяет программам выполняться быстрее. В терминах схемы на рис. 2.2 Psycodo – 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 изменения в исходный программный код или производить дополнительную компиляцию в ходе разработки.

Грубо говоря, во время выполнения программы Psycodo собирает информацию о типах объектов, и затем эта информация используется для генерации высокоэффективного 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 Psycodo существенно уменьшается общее время выполнения программы. В идеале некоторые участки программного кода под управлением Psycodo могут выполняться так же быстро, как скомпилированный код языка C.

Поскольку эта компиляция из байт-кода производится во время выполнения программы, обычно Psycodo называют *динамическим* (just-in-time, JIT) компилятором. Однако в действительности Psycodo немного отличается от JIT-компиляторов, которые, возможно, некоторые читатели видели в языке Java. В действительности Psycodo – 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, Psycodo может генерировать различные версии машинного кода для поддержки каждой из комбинаций.

Применение Psycodo показывает существенное увеличение скорости выполнения программного кода Python. Согласно информации, которая приводится на домашней странице проекта, Psycodo обеспечивает увеличение скорости «от 2 до 100 раз, обычно в 4 раза, при использовании немодифицированного интерпретатора Python, неизменного исходного

текста, всего лишь за счет использования динамически загружаемого модуля расширения на языке С». 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 С с целью оптимизации. При использовании Psycodo необходимость в таком переносе теряет свою остроту.

До сих пор Psycodo не является стандартной частью Python – его нужно загружать и устанавливать отдельно. Кроме того, он до сих пор находится на экспериментальной стадии развития, поэтому вам нужно будет следить за его разработкой. В действительности, когда я пишу эти строки, Psycodo все еще можно загрузить и установить, но похоже, что большая его часть будет поглощена более новым проектом «PyPy», который представляет собой попытку переписать PVM на языке Python с целью обеспечения высокой степени оптимизации, как в Psycodo.

Пожалуй, самым большим недостатком Psycodo является то обстоятельство, что в настоящее время он способен генерировать машинный код только для архитектуры Intel x86, впрочем, на этой архитектуре работают такие операционные системы, как Windows, Linux и даже Mac. За дополнительной информацией о расширении Psycodo и других попытках реализации JIT-компилятора обращайтесь на сайт <http://www.python.org>. Кроме того, вы можете посетить домашнюю страницу проекта Psycodo, которая в настоящее время размещается по адресу <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 скомпилирован в машинный код. Кроме того, эта система реализует платформонезависимый подход к выполнению программного кода Python. К моменту написания этих строк система Shedskin еще находилась на экспериментальной стадии развития и ограничивала программы Python неявным использованием 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 Psycodo, и это весьма многообещающий проект. Сведения о текущем 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 созда-

вать из своих программ на языке Python самостоятельные исполняемые файлы. Это необходимо 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 превратить 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 программам. Существуют разные реализации такого подхода, но в конечном результате получается единственный исполняемый файл (например, файл с расширением *.exe* в Windows), который легко можно передать заказчику. Такую модель можно представить, если на рис. 2.2 объединить байт-код и PVM в единый компонент – фиксированный двоичный файл.

На сегодняшний день существует три основных инструмента создания фиксированных двоичных файлов: *py2exe* (для Windows), *PyInstaller* (напоминает *py2exe*, но также работает в Linux и UNIX и способен производить самоустанавливающиеся исполняемые файлы) и *freeze* (оригинальная версия). Вам придется загружать эти инструменты отдельно от Python, но они распространяются совершенно бесплатно. Кроме того, они постоянно развиваются, поэтому за дополнительной информацией об этих инструментах обращайтесь на сайт проекта Python (<http://www.python.org>) и Vaults of Parnassus (<http://www.vex.net/parnassus/>). Чтобы дать вам общее представление об области применения этих инструментов, замечу, что *py2exe* может создавать автономные программы, использующие библиотеки Tkinter, PMW, 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 результате работы настоящего компилятора, потому что они выполняют байт-код с помощью виртуальной машины. Следовательно, программы в фиксированных 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 большими. 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 программный код упакован в фиксированный двоичный файл, он надежно скрыт от получателя.

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 устанавливать (и даже знать о том, что это такое) Python, чтобы запустить распространяемую программу.

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 актуальность (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 реализации интерпретатора и разновидности байт-кода. Например:

- Недавно появившийся проект *Parrot* поставил перед собой цель выработать единый формат байт-кода, единую виртуальную машину и методики оптимизации для различных языков программирования (подробности на сайте <http://www.python.org>).
- *Stackless Python* – вариант стандартной реализации CPython, в которой не производится сохранение состояния на стеке вызовов языка C. Это упрощает перенос Python на архитектуры с небольшим размером стека и открывает новые возможности программирования, такие как сопрограммы.
- Новый проект *PyPy* – попытка реализовать PVM непосредственно на языке Python, что позволит использовать новые приемы программирования.

Подобные грядущие схемы реализации могут несколько изменить схему времени выполнения интерпретатора Python, однако скорее всего компилятор байт-кода останется стандартом еще какое-то время. Переносимость и гибкость байт-кода во время выполнения – это очень важные качества многих реализаций Python. Более того, добавление в язык конструкций объявления типов 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 сохраняют некоторые черты нынешней PVM.

В заключение

В этой главе была представлена модель выполнения Python (как Python запускает программы) и исследованы некоторые наиболее известные разновидности этой модели (динамические компиляторы и тому подобное). Чтобы писать сценарии на языке Python, вам необязатель-

но знать внутреннюю организацию интерпретатора и, тем не менее, некоторое знакомство с темой этой главы поможет 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?

Ответы

1. Интерпретатор Python – это программа, которая выполняет программы на языке Python.
2. Исходный программный код – это инструкции, составляющие вашу программу. Он состоит из текста в текстовых файлах, имена которых обычно имеют расширение *.py*.
3. Байт-код – это низкоуровневое представление программы после ее компиляции. Python автоматически сохраняет полученный байт-код в файлах с расширением *.pyc*.
4. PVM – это Python Virtual Machine (виртуальная машина Python) – механизм Python, который интерпретирует скомпилированный программный код.
5. Psyc, Shedskin и фиксированные двоичные файлы – все это разновидности модели выполнения.
6. CPython – это стандартная реализация языка. Jython и IronPython реализуют поддержку программирования на языке Python в среде Java и .NET, соответственно; они являются альтернативными компиляторами языка Python.

3

Как пользователь запускает программы

Итак, настал момент запустить какой-нибудь программный код. Теперь, когда вы получили представление, как программы запускаются, вы готовы приступить к программированию на языке Python. С этого момента я буду предполагать, что интерпретатор Python уже установлен у вас на компьютере, в противном случае вернитесь к предыдущей главе, а также прочитайте приложение А, 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 вводить программный код в *интерактивном* режиме, как сохранять его в *файлах*, которые можно будет запускать из командной строки, щелчком на ярлыке и импортировать в виде модулей. Мы также поговорим об интегрированных средах разработки с графическим интерфейсом, таких как IDLE и Eclipse, и о многом другом.

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 программ на языке Python. Я также рекомендую просмотреть разделы с описанием 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. Запустить эту командную строку можно разными способами – в интегрированной среде разработки, в системной консоли и т. д. Предположим, что интерпретатор установлен в вашей системе как выполняемая программа, тогда самый универсальный способ запустить интерактивный сеанс работы с интерпретатором заключается в том, чтобы ввести команду `python`, без аргументов, в командной строке вашей операционной системы. Например:

```
% python
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on
win 32
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). Понятие системной командной строки является универсальным, но как получить доступ к ней, зависит от используемой платформы:

- В операционной системе Windows 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), которое появляется после щелчка на кнопке Пуск (Start)), или в диалоге Пуск→Выполнить... (Start→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 PalmPilots, можно щелкнуть на ярлыке Python.

Если вы не включили путь к каталогу установки Python в переменную окружения `PATH`, вам вместо простого слова «python» может потребоваться ввести полный путь к выполняемой программе. В операционной системе Windows можно попробовать ввести команду `C:\Python25\python` (для версии 2.5); в UNIX и в Linux: `/usr/local/bin/python` (или `/usr/bin/python`). Или перед вводом команды «python» можно перейти в каталог, куда был установлен интерпретатор Python (для этого в операционной системе Windows, например, можно выполнить команду `cd c:\python25`).

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 интерпретатор Python переходит в режим ожидания ввода новой инструкции или выражения. При работе в интерактивном режиме результаты выполнения ваших инструкций будут выводиться

сразу же вслед за строкой с приглашением `>>>`. Ниже приводятся результаты выполнения двух инструкций `print`:

```
% 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 `>>>`.

При работе в интерактивном режиме, как показано в этом примере, вы можете вводить любое число команд Python, и каждая из них будет вы-

Тестирование программного кода в интерактивном режиме

Благодаря тому, что программный код выполняется немедленно, интерактивный режим превращается в замечательный инструмент для выполнения *экспериментов* 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 участок программного кода на языке Python, запустите интерактивный сеанс и попробуйте ввести этот фрагмент, чтобы посмотреть, что произойдет. В этом случае вы едва ли что-нибудь испортите (чтобы стать опасным для системы, вы должны хорошо знать системные интерфейсы).

Интерактивная оболочка может использоваться для тестирования программного кода, сохраненного в файлах, что избавит вас от необходимости вводить значительные объемы программного кода при работе в интерактивном режиме (ведь программный код, который здесь вводится, не сохраняется). В интерактивном режиме вы можете импортировать файлы модулей и тестировать функциональные возможности, которые они предоставляют, вводя вызовы функций в строке с приглашением к вводу. 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 многое другое. Интерпретатор поддерживает возможность проведения экспериментов и исследований при программировании, и вы найдете это удобным, начав работать с ним.

полняться сразу же после ввода. Более того, поскольку в интерактивном сеансе результаты выражений, которые вы вводите, печатаются автоматически, вам необязательно явно вводить «print» после подсказки:

```
>>> lumberjack = 'okay'
>>> lumberjack
'okay'
>>> 2 ** 8
256
>>> # Для завершения сеанса используйте комбинацию клавиш Ctrl-D или Ctrl-Z
%
```

В этом примере две последние введенные строки являются выражениями (`lumberjack` и `2 ** 8`), и результаты их вычисления отображаются автоматически. Чтобы завершить работу интерактивного сеанса, как показано в данном примере, и вернуться в системную командную строку, в UNIX-подобной системе нажмите комбинацию клавиш `Ctrl-D`, а в системах MS-DOS и Windows – комбинацию `Ctrl-Z`. В интегрированной среде разработки IDLE, которая будет рассматриваться ниже, нужно либо нажать комбинацию клавиш `Ctrl-D`, либо просто закрыть окно.

В приведенных примерах мы сделали немного – всего лишь ввели несколько инструкций `print`, одну инструкцию присваивания и несколько выражений, подробнее о которых мы поговорим позднее. Главное, на что следует обратить внимание, – интерпретатор немедленно выполняет введенный программный код, сразу же после нажатия клавиши `Enter`.

Например, когда в строке приглашения к вводу `>>>` была введена первая инструкция `print`, результат (строка) был немедленно выведен на экран. Нам не потребовалось создавать файл с исходным текстом программы и для выполнения программного кода не понадобилось сначала компилировать и компоновать его, что является обычным делом при использовании таких языков программирования, как C или C++. Как будет показано в последующих главах, при работе с интерактивной оболочкой вы можете также вводить многострочные инструкции – такие инструкции будут выполняться только после ввода всех строк.

Помимо ввода команды `python` в окне командной оболочки, вы также можете запустить подобный интерактивный сеанс в главном окне интегрированной среды разработки IDLE или, в Windows, выбрав пункт Python (command-line) (Python (командная строка)) в меню кнопки Пуск (Start), как показано на рис. 2.1. В любом случае вы получите строку приглашения к вводу `>>>` с одинаковыми функциональными возможностями – инструкции выполняются сразу же после ввода.

Использование интерактивного режима

Несмотря на то что интерактивный режим прост в использовании, я хочу дать несколько советов начинающим, которые следует запомнить:

- **Вводите только инструкции на языке Python.** Прежде всего, следует запомнить, что в интерактивном режиме допускается вводить только

программный 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 сравнению с непосредственным вводом команд.

- **Инструкция `print` необходима только в файлах.** Поскольку в интерактивном режиме интерпретатор автоматически выводит результаты вычисления выражений, вам не требуется вводить полные инструкции `print` при работе в интерактивном режиме. Это замечательная особенность, но она часто приводит пользователей в замешательство, когда они приступают к созданию 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. Запомните, вы должны использовать инструкцию `print` в файлах, но не в интерактивном режиме.
- **Не используйте отступы в интерактивном режиме (пока).** При вводе программ 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). Пока мы не подойдем к главе 10, все инструкции, которые вам придется вводить, будут не вложенными, поэтому пока данное правило распространяется 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 составными (многострочными) инструкциями до главы 10, но вы должны знать, что при вводе второй и каждой последующей строки составной инструкции в интерактивном режиме строка приглашения к вводу может изменяться. В простом окне с командной строкой приглашение к вводу `>>>` во второй и каждой последующей строке изменяется на `...`, в интегрированной среде IDLE все строки, кроме первой, автоматически получают отступы. В любом случае, чтобы в интерактивном режиме сообщить интерпретатору, что вы приступаете к вводу многострочной инструкции, необходимо начинать ее ввод с ввода пустой строки (делается это нажатием клавиши Enter в начале строки). В файлах, в противоположность этому, пустые строки интерпретатором игнорируются.

Почему это имеет важное значение, вы узнаете в главе 10. А пока, если вдруг случится, что вы получите приглашение к вводу `...` или пустую строку при вводе программного кода, это скорее всего будет означать, что каким-то образом 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 вводу. Пригла-

шения `>>>` и `...` также могут изменяться (они доступны во встроенном модуле `sys`), но в последующих листингах я буду предполагать, что они не изменялись.

Системная командная строка и файлы

Хотя интерактивная командная оболочка является прекрасным инструментом для проведения экспериментов, тем не менее у нее есть один существенный недостаток: программы, которые вы вводите во время интерактивного сеанса, исчезают сразу же после того, как интерпретатор Python выполнит их. Программный код, который вводится в интерактивном режиме, нигде не сохраняется, поэтому вы не сможете запустить его еще раз, не введя код с самого начала. Операция копирования и вставки, а также возможность повторного выполнения команды, могут оказать некоторую помощь, но они не способны принести пользу, когда вы начнете писать большие программы. Чтобы воспользоваться операцией копирования и вставки, вы должны исключить из копирования строку приглашения к вводу, результаты, которые программа выводит в процессе выполнения, и т. д.

Чтобы хранить программы длительное время, вам необходимо сохранять программный 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 интегрированной среды IDLE и другими способами. Независимо от выбранного вами способа интерпретатор 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 программой понимается последовательность заранее написанных инструкций, сохраненных 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 программах «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. В качестве первого упражнения откройте привычный для вас текстовый редактор (например, *vi*,

Notepad или редактор IDLE) и сохраните следующие две инструкции в файле с именем *spam.py*:

```
print 2 ** 8                                # Возведение в степень
print 'the bright side ' + 'of life'        # Знак + означает конкатенацию строк
```

Этот файл содержит две инструкции `print` и пару *комментариев* справа. (Текст, который следует за символом `#`, просто игнорируется и считается *комментарием*, созданным для удобства человека; комментарии не являются частью инструкций.) Пока не нужно вникать в синтаксис программного кода в этом файле. Главное здесь то, что программный код вводится в текстовый файл, а не в интерактивной командной оболочке интерпретатора Python. Итак, вы создали полноценный сценарий на языке Python.

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 spam.py
256
the bright side of life
```

И в этом случае также вы должны использовать командную оболочку, которая предоставляется вашей операционной системой, – в окне Командная строка (Command Prompt) в Windows, в *xterm* или в подобных им программах. Не забывайте заменять слово «python» на полный путь к исполняемому файлу интерпретатора, если переменная окружения `PATH` у вас не настроена. Вывод этого сценария можно наблюдать ниже строки, где была введена команда, – это результат работы двух инструкций `print`, находящихся в текстовом файле.

Обратите внимание, что файл модуля называется *spam.py*. Как и любой другой файл верхнего уровня, он мог бы также называться просто *spam*, однако файлы, которые должны импортироваться клиентом, обязаны иметь расширение *.py*. Вопросы импорта мы будем обсуждать ниже, в этой же главе.

Позднее вам может потребоваться импортировать тот или иной файл, поэтому всегда желательно использовать расширение *.py* в именах файлов с программным кодом на языке Python. Кроме того, некоторые текстовые редакторы определяют принадлежность файлов по расширению *.py* – если расширение отсутствует, вы можете лишиться таких функциональных возможностей редактора, как подсветка синтаксиса и автоматическое оформление отступов.

Поскольку в данной ситуации для запуска программ на языке Python используется командная оболочка, можно применять любые синтаксические конструкции, допускаемые командной оболочкой. Например, можно перенаправить вывод сценария Python в файл, чтобы детально исследовать полученные результаты позднее, как показано ниже:

```
% python spam.py > saveit.txt
```

В этом случае две строки, которые были показаны в предыдущем примере запуска сценария, не будут выводиться на экран, а будут записаны в файл *saveit.txt*. Это широко известная возможность *перенаправления потоков* – она может использоваться как для вывода текста, так и для ввода. Она присутствует в Windows и в UNIX-подобных системах. Она мало связана с Python (интерпретатор Python просто поддерживает ее), поэтому здесь мы не будем углубляться в подробности работы механизма перенаправления.

Если вы пользуетесь операционной системой Windows, этот пример также будет работать, хотя сама командная строка будет выглядеть несколько иначе:

```
C:\Python25> python spam.py
256
the bright side of life
```

Если у вас переменная окружения `PATH` не настроена и не был выполнен переход в каталог интерпретатора, вам необходимо вводить полный путь к исполняемому файлу интерпретатора Python:

```
D:\temp> C:\python25\python spam.py
256
the bright side of life
```

В новейших версиях Windows вы можете просто вводить имя файла сценария независимо от того, в каком каталоге вы находитесь, потому что новейшие версии системы Windows отыскивают программы, необходимые для запуска файлов, с помощью реестра Windows и вам не требуется явно указывать ее в командной строке. Например, в современных версиях Windows предыдущую команду можно упростить до:

```
D:\temp> spam.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`, но она должна запустить сценарий, расположенный в некотором другом каталоге:

```
D:\other> python c:\code\myscript.py
```

Использование системной командной строки и файлов

Запуск файлов программ из командной строки системы является достаточно простой задачей, особенно если вы и раньше работали с командной строкой. Тем не менее ниже описываются несколько ловушек, в которые часто попадают начинающие:

- **Остерегайтесь автоматического присвоения расширения файлам в операционной системе Windows.** Если для создания файлов про-

грамм в Windows вы пользуетесь редактором Блокнот (Notepad), перед сохранением выбирайте тип файла Все файлы (All Files) и явно указывайте расширение *.py*. В противном случае Блокнот (Notepad) будет присваивать файлам расширение *.txt* (например, *spam.py.txt*), что в некоторых ситуациях осложнит запуск таких файлов.

Хуже того, по умолчанию операционная система Windows скрывает расширения файлов, поэтому если вы забыли указать тип файла, вы можете даже не заметить, что создали обычный текстовый файл, а не файл, который должен запускаться интерпретатором Python. Здесь вам может помочь 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 запуска в результате щелчка мышью могут служить еще одним признаком этой проблемы.

Текстовый процессор Microsoft Word похожим образом по умолчанию добавляет расширение *.doc*. Мало этого, он добавляет в файл символы форматирования, 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 Windows или используйте более удобные для программистов текстовые редакторы, такие как IDLE. IDLE не добавляет расширение к имени файла автоматически, даже расширение *.py* – эта особенность нравится программистам, но не пользователям.

- **Указывайте расширение в командной строке, но не в инструкциях импорта.** Не забывайте указывать полное имя файла в командной строке, то есть используйте, например, команду `python spam.py`, а не `python spam`. Инструкция `import`, с которой мы познакомимся ниже, в этой же главе, требует, чтобы путь к файлу и его расширение были опущены (например, `import spam`). Это несложно, но данная особенность часто является источником ошибок.

Интерпретацией команды в командной строке занимается система, а не интерпретатор Python, поэтому в ней неприменимы правила поиска файлов, которые использует Python. Поэтому в командной строке необходимо всегда указывать расширение файла *.py* и при необходимости – путь к файлу, который требуется запустить. Например, чтобы запустить файл, находящийся в каталоге, отличном от того, в котором вы работаете, обычно необходимо указать полный путь к файлу (например, `C:\python25>python d:\tests\spam.py`). Однако в программном коде на языке Python вам достаточно просто указать инструкцию `import spam` и доверить интерпретатору 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, чтобы еще раз напомнить о ней. В отличие от интерактивного режима, чтобы вывести результаты работы файлов программ на экран, вы должны использовать инструкции `print`.

Исполняемые сценарии в UNIX (#!)

Если вы используете Python в UNIX, Linux или в другой UNIX-подобной операционной системе, вы можете превратить файлы с программным кодом на языке Python в исполняемые программы, точно так же, как программы на таких языках командной оболочки, как *ksh* или *ssh*. Такие файлы обычно называются *исполняемыми сценариями*. Проще говоря, исполняемые сценарии в UNIX-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 них установлено право на выполнение.** Обычно файлы сценариев помечаются как исполняемые файлы, чтобы сообщить системе, что они могут быть запущены как программы верхнего уровня. 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...'    # Это еще один комментарий
```

Первая строка в файле сообщает системе, где находится интерпретатор Python. С технической точки зрения для интерпретатора Python первая строка является комментарием. Как уже говорилось ранее, все, что начинается с символа `#` и до конца строки, является комментарием — в них размещается дополнительная 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 использует его для поиска интерпретатора, который будет выполнять остальной программный код в файле.

Кроме того, обратите внимание, что этот файл называется просто *brian*, в его имени отсутствует расширение *.py*, которое мы использовали ранее для обозначения 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 является вполне допустимым. Если дать этому файлу право на выполнение с помощью 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 Windows замечу, что метод, описываемый здесь, характерен для UNIX и может быть неприменим на вашей платформе. Однако вам незачем волноваться, просто используйте метод запуска, который описывался выше. Укажите имя выполняемого файла, как первый аргумент команды `python`.¹

```
C:\book\tests> 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

Трюк с использованием команды `env` в UNIX

В некоторых версиях системы UNIX можно избежать явного указания пути к интерпретатору Python, если специальный комментарий в первой строке оформить следующим образом:

```
#!/usr/bin/env python
...здесь находится программный код сценария...
```

При таком подходе программа `env` отыщет интерпретатор Python в соответствии с настройками пути поиска (то есть в большинстве командных оболочек UNIX поиск будет произведен во всех каталогах, перечисленных 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 сценарии 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 сценариев. Безусловно, этот способ предполагает, что во всех системах программа `env` находится в одном и том же каталоге (в некоторых системах она может располагаться в каталогах `/sbin`, `/bin` или где-то еще), в противном случае о переносимости не может быть и речи.

¹ Как уже говорилось при обсуждении командной строки, современные версии Windows позволяют указывать в командной строке лишь имя файла с расширением `.py` — эти версии Windows 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 программы для Windows действительно могут использовать и интерпретировать первую строку `#!`, но командная оболочка DOS в Windows полностью игнорирует ее.

хотите обеспечить переносимость процедуры запуска между UNIX и Microsoft Windows, 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.

Щелчок на ярлыке файла

Использование реестра в операционной системе Windows позволяет открывать файл просто щелчком мыши. При установке интерпретатор Python автоматически регистрирует себя в качестве программы, используемой для открытия файлов с программами на языке Python щелчком мыши. Это делает возможным запуск программ на языке Python простым щелчком (или двойным щелчком) мыши на ярлыке файла.

В операционных системах, отличных от Windows, вы наверняка сможете проделать аналогичный трюк, но сами ярлыки, программа просмотра файловой системы, система навигации и прочее могут несколько отличаться. В некоторых системах UNIX, например, может потребоваться зарегистрировать расширение *.py* в программе просмотра файловой системы, сделать свой 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 программ или с помощью других инструментальных средств. Если щелчок мышью не дает нужного результата, обращайтесь к документации используемой программы просмотра файловой системы за дополнительной информацией.

Щелчок на ярлыке в Windows

Чтобы продемонстрировать эту возможность, предположим, что у вас имеется следующая программа на языке Python, которая была сохранена в файле *script4.py*:

```
# Комментарий
import sys
print sys.platform
print 2 ** 100
```

Здесь не так много нового — всего лишь инструкция `import` и две инструкции `print` (`sys.platform` — это всего лишь строка, идентифицирующая платформу, с которой вы работаете; она определена в модуле с именем `sys`, который необходимо импортировать). Эту программу можно запустить из командной строки системы:

```
D:\LP3E\Examples> c:\python25\python script4.py
win32
1267650600228229401496703205376
```

Однако эту программу можно также запустить щелчком мыши, вообще ничего не вводя с клавиатуры. Можно попробовать отыскать ярлык этого файла, например, щелкнув на ярлыке Мой компьютер (My Computer)

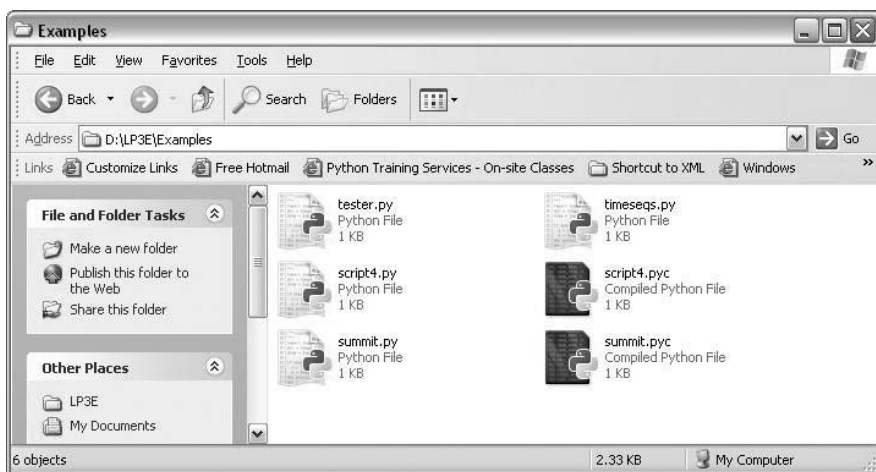


Рис. 3.1. В Windows файлы с программами на языке Python отображаются в проводнике в виде ярлыков и могут быть запущены двойным щелчком мыши (хотя при этом вы можете не увидеть текст, выводимый программой, и сообщения об ошибках)

и выполнив переход вглубь дерева каталогов на диске *D*, в этом случае вы получите изображение в проводнике, как показано на рис. 3.1 (этот снимок с экрана был получен в Windows XP). В случае использования Python 2.5 ярлыки файлов с исходными текстами программ 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 исходными текстами, чтобы увидеть последние изменения. Для запуска файла здесь нужно просто щелкнуть на ярлыке файла *script4.py*.

Использование функции `raw_input`

К сожалению, запуск файла в Windows щелчком на ярлыке может не привести к удовлетворяющему вас результату. В действительности, как в данном примере, запуск сценария 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 особенность обслуживания вывода программы в Windows.

По умолчанию интерпретатор Python открывает черное окно консоли DOS, которое будет служить местом для ввода и вывода программы. Если сценарий выводит 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 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 ярлыке файла, просто добавьте вызов встроенной функции `raw_input` в самом конце сценария. Например:

```
# Комментарий
import sys
print sys.platform
print 2 ** 100
raw_input()           # Добавленная строка
```

Вообще, функция `raw_input` считывает следующую строку с устройства стандартного 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 Windows, и только если сценарий выводит какие-либо сообщения, и только если сценарий запускается щелчком мыши на ярлыке. Вызов функции следует добавлять в самый конец файлов верхнего уровня и только при выполнении всех трех перечисленных условий. Во всех остальных случаях нет смысла добавлять этот вызов.¹



Рис. 3.2. При запуске программы в Windows щелчком мыши на ярлыке можно увидеть результаты ее работы, если добавить вызов функции `raw_input()` в самый конец сценария. Но делать это следует, только если это действительно необходимо!

¹ Кроме того, существует возможность полностью подавить появление окна консоли DOS при запуске файлов щелчком мыши в Windows. Программы, имена файлов которых имеют расширение `.pyw`, отображают только те окна, которые создаются самими 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 Windows обслуживаются специальным образом. Главным образом они используются для сценариев Python, которые сами создают окна пользовательского интерфейса, что часто сочетается с организацией вывода в файлы результатов работы и сообщений об ошибках.

Следует отметить, что функция `raw_input` — это средство для ввода информации, дополняющая инструкцию `print`, которая является средством вывода. Она представляет собой простейший способ чтения вводимых данных, и обладает более широкими возможностями, чем было продемонстрировано в этом примере. Например, функция `raw_input`:

- Может принимать строку в качестве аргумента, которая будет выводиться как подсказка (например, `raw_input('Press Enter to exit')`).
- Возвращает сценарию текстовую строку (например, `nextinput = raw_input()`).
- Поддерживает возможность перенаправления потока 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: например, в главе 10 эта функция будет использована для организации интерактивного цикла.

Другие ограничения на запуск щелчком мыши

Даже при использовании функции `raw_input` запуск файлов щелчком мыши имеет некоторые недостатки. Вы можете не заметить появление сообщений об ошибках. Если в ходе работы сценария возникает ошибка, текст сообщения о ней выводится в окно консоли, которое тут же закрывается. Хуже того, даже добавление вызова `raw_input` не поможет в такой ситуации, потому что работа сценария 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 интегрированной среды разработки IDLE (будет рассматриваться ниже, в этой же главе). Благодаря этому вы сможете увидеть сообщения об ошибках и обычный вывод от сценария, не прибегая к разного рода хитростям. Когда позднее в этой книге мы будем рассматривать исключения, вы узнаете, что существует возможность перехватывать и обрабатывать ошибки так, чтобы они не приводили к аварийному завершению программы. Обратите внимание на приведенное ниже обсуждение инструкции `try`, которая предоставляет альтернативный способ предотвратить преждевременное закрытие окна в случае возникновения ошибок.

Импортирование и перезагрузка модулей

Мы уже говорили об «импортировании модулей», но до сих пор я не давал никаких пояснений, что означает этот термин. Подробно о моду-

лях и об архитектуре 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 представление о них.

Проще говоря, каждый файл с исходным текстом на языке Python, имя которого оканчивается расширением *.py*, является модулем. Другие файлы могут обращаться к программным компонентам, объявляемым модулем, *импортируя* этот модуль. По сути, инструкция `import` выполняет загрузку другого файла и обеспечивает доступ к его содержимому. Содержимое модуля становится доступным внешнему миру через его атрибуты (определение этого термина я дам в следующем разделе).

Эта модульная модель является центральной идеей, лежащей в основе архитектуры программ на языке Python. Крупные программы обычно организованы в виде множества файлов модулей, которые импортируют и используют функциональные возможности из других модулей. Один из модулей определяется как основной файл *верхнего уровня*, который запускает всю программу.

Проблемы модульной архитектуры мы будем рассматривать подробнее позже, в этой же книге, а в этой главе основное внимание уделяется тому факту, что операция импорта приводит к *выполнению* программного кода загружаемого файла на заключительном этапе. 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 *script4.py*, созданный ранее, с помощью простой инструкции `import`:

```
D:\LP3E\Examples> c:\python25\python
>>> import script4
win32
1267650600228229401496703205376
```

Такой способ пригоден только для однократного запуска модуля в течение сеанса. После первой операции импорта все последующие попытки импортировать модуль не приводят ни к каким результатам, даже если изменить и сохранить исходный текст модуля в другом окне:

```
>>> import script4
>>> import script4
```

Так сделано преднамеренно – операция импорта требует слишком больших затрат вычислительных ресурсов, чтобы выполнять ее более одного раза в ходе выполнения программы. Как вы узнаете в главе 18, в ходе импорта производится поиск файлов, компиляция их в байт-код и выполнение этого байт-кода.

Если действительно возникает необходимость вынудить интерпретатор многократно запускать файл в рамках одного и того же сеанса (без

остановки и перезапуска сеанса), можно воспользоваться встроенной функцией `reload`:

```
>>> reload(script4)
win32
65536
<module 'script4' from 'script4.py'>
>>>
```

Функция `reload` загружает и запускает текущую 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 `script4.py` была изменена в другом окне так, чтобы она выводила результат выражения `2 ** 16`, после чего была выполнена перезагрузка модуля с помощью функции `reload`.

Функция `reload` ожидает получить имя уже загруженного модуля, поэтому, прежде чем перезагрузка станет возможной, модуль должен быть импортирован. Примечательно также, что имя модуля при вызове функции `reload` должно 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 дополнительная строка. Последняя строка в выводе выше — это всего лишь представление результата, возвращаемого функцией `reload` после перезагрузки модуля.

Более подробно функции будут обсуждаться в главе 15.

Важные сведения о модулях: атрибуты

Операции импортирования и перезагрузки модулей обеспечивают естественный способ запуска программы, так как на заключительном этапе этих операций 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 определенным объектом.

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 модулем, — функциями, классами, переменными и т. д., которые предназначены 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 с помощью текстового редактора создайте однострочный файл модуля Python с именем *myfile.py* со следующим содержимым:

```
title = "The Meaning of Life"
```

Это, пожалуй, один из самых простых модулей Python (он содержит единственную операцию присваивания), но его вполне достаточно для иллюстрации основных положений. При импортировании этого модуля выполняется его программный код, который создает атрибут модуля. Инструкция присваивания создает атрибут с именем *title*.

Доступ к атрибуту *title* можно получить из других программных компонентов двумя разными способами. Первый заключается в том, чтобы загрузить модуль 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 обращения к строковой переменной *title*, определенной внутри модуля *myfile*, — то есть *myfile.title*.

Кроме того, доступ к именам внутри модулей (фактически, копию 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
```

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, выполняющем импорт, благодаря чему на этот раз он может обратиться к импортированной 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*, это приводит к выполнению инструкций в файле

¹ Обратите внимание: в обеих инструкциях, *import* и *from*, имя модуля *myfile* указывается 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 указываться при вызове файла в системной командной оболочке и опускаться в инструкциях *import*.

myfile.py, а импортирующий компонент (в данном случае – интерактивная оболочка интерпретатора) получает доступ к именам, определенным в файле на верхнем уровне. В этом простом примере существует только одно такое имя – переменная *title*, которой присвоена строка, но сама концепция приобретает более важное значение, когда речь заходит об определении в модулях таких объектов, как функции и классы. Такие объекты становятся программными 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, использующие инструкцию *import*, получают модуль со всеми его атрибутами, а клиенты, использующие инструкцию *from*, получают копии имен из этого модуля:

```
% python
>>> import threenames                # Загрузить модуль целиком
dead parrot sketch
>>>
>>> threenames.b, threenames.c
('parrot', 'sketch')
>>>
>>> from threenames import a, b, c    # Скопировать несколько имен
>>> b, c
('parrot', 'sketch')
```

Результаты здесь выводятся в круглых скобках, потому что в действительности они являются *кортежами* (разновидность объектов, которая описывается в следующей части книги).

Как только вы начнете создавать модули, содержащие несколько имен, как в данном случае, вам наверняка пригодится встроенная функция *dir*. Она может использоваться для получения списка имен, доступных внутри модуля:

```
>>> dir(threenames)
['_builtins_', '_doc_', '_file_', '_name_', 'a', 'b', 'c']
```

При вызове функции `dir` передается имя импортированного модуля в круглых скобках, как показано выше, а возвращает она список всех атрибутов, определенных 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 инструкциями 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 являются 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 — т. к. 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 языке Python для упаковки переменных в категории, чтобы избежать конфликтов имен. Далее мы еще будем обсуждать модули и другие конструкции образования пространств имен (включая классы и функции). А пока будем использовать модули в качестве средства многократного использования программного кода, позволяющего не вводить его повторно с клавиатуры.

`import` и `reload`, примечания к использованию

Зачастую, узнав о возможности запуска файлов с помощью `import` и `reload`, начинающие разработчики концентрируют все свое внимание на этом способе и забывают о других возможностях запуска, позволяющих всегда запускать самую свежую версию программного кода (например, щелчок мышью на ярлыке, пункты меню интегрированной среды разработки IDLE и системная командная строка). Такой подход может быстро привести к появлению ошибок — вам придется запоминать, импортировали ли вы тот или иной модуль, чтобы иметь возможность перезагрузить 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.

Из-за этих сложностей (и некоторых других, с которыми мы еще столкнемся позднее) пока лучше избегать пользоваться операциями импорта и перезагрузки. Пункт меню Run→Run Module интегрированной среды IDLE, например, предоставляет более простой способ запуска файлов, менее подверженный ошибкам. С другой стороны, импорт и перезагрузку оправданно использовать в процессе тестирования классов Python. В этом случае вы можете отдать предпочтение этому способу, но если начнут возникать сложности – остановитесь.

Это еще не все сведения о модулях, на которых хотелось бы здесь остановиться. Например, встроенная функция `execfile('module.py')` – еще один способ запуска файлов из интерактивной командной строки, который избавляет от необходимости импортировать и затем перезагружать модули. Имея схожее действие, она технически не выполняет импорт модуля – по умолчанию всякий раз, когда вызывается функция `execfile`, она снова запускает файл, как если бы на место вызова функции было подставлено содержимое этого файла. Из-за этого функция `execfile`, как и упоминавшаяся ранее инструкция `from`, может попросту затереть значения переменных, которые уже использовались. С другой стороны, инструкция `import` запускает файл всего один раз в течение сеанса работы и превращает файл в отдельное пространство имен, благодаря чему становится невозможным изменение переменных в текущей области видимости.

Следует добавить, что в случае использования необычных способов применения модулей, отличных от тех, которые описываются в этой книге, вы можете столкнуться с некоторыми неприятностями. Например, если вам необходимо импортировать файл модуля, который хранится в каталоге, отличном от того, в котором вы работаете, дождитесь, пока мы не подойдем к главе 18, где вы узнаете о *пути поиска модулей*. А пока, чтобы избежать осложнений, храните все импортируемые файлы модулей в рабочем каталоге.



Для тех, кто не хочет ждать, пока повествование дойдет до главы 18, скажу, что интерпретатор Python выполняет поиск импортируемых модулей во всех каталогах, перечисленных в переменной `sys.path`, – в списке имен каталогов, определенной в модуле `sys`, которая инициализируется 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 переменной `PYTHONPATH`. За дополнительной информацией обращайтесь к главе 18.

Пользовательский интерфейс IDLE

До сих пор мы рассматривали запуск программного кода Python с помощью интерактивной командной строки интерпретатора, системной командной строки, с помощью щелчка мышью на ярлыке и с исполь-

зованием операции импорта. Если вам требуется более наглядный подход, интегрированная среда IDLE может предложить вам графический интерфейс пользователя (ГИП) для разработки программ на языке Python; IDLE является стандартной и свободно распространяемой частью системы Python. Обычно она называется *интегрированной средой разработки* (integrated development environment, IDE), потому что позволяет решать разнообразные задачи в единой оболочке.¹

Проще говоря, IDLE – это набор инструментальных средств с графическим интерфейсом, который способен работать на самых разных платформах, включая Microsoft Windows, X Window (в Linux, UNIX и других UNIX-подобных операционных 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 Windows не вызывает проблем – для нее создается отдельный пункт в разделе Python меню кнопки Пуск (Start) (рис. 2.1), а кроме того, ее можно запустить, выбрав пункт контекстного меню, щелкнув правой кнопкой мыши на ярлыке программы, написанной на языке Python. В некоторых UNIX-подобных системах для запуска начального сценария IDLE может потребоваться использовать командную строку или щелкнуть мышью на ярлыке файла *idle.pyw* или *idle.py*, размещенного в подкаталоге *idlelib* в каталоге *Lib*, где установлен интерпретатор Python. (В Windows IDLE является сценарием Python, который по умолчанию находится в каталоге *C:\Python25\Lib\idlelib*.²)

На рис. 3.3 показано, как выглядит среда IDLE, запущенная в операционной системе Windows. Окно с заголовком Python Shell (Оболочка Python), которое открывается первоначально, является основным окном среды, в котором запускается интерактивный сеанс работы с интерпретатором (обратите внимание на приглашение к вводу `>>>`). Это самый

¹ Официально название IDLE считается искаженной аббревиатурой IDE, но в действительности она была названа так в честь члена труппы цирка Монти Пайтона (Monty Python) – Эрика Эйдла (Eric Idle).

² IDLE – это программа на языке Python, которая создает графический интерфейс с помощью библиотеки Tkinter GUI, что обеспечивает ее переносимость, но также означает, что для использования IDLE вам придется обеспечить поддержку Tkinter в Python. Версия Python для Windows обладает такой поддержкой по умолчанию, но некоторым пользователям Linux и UNIX может потребоваться установить соответствующую поддержку Tkinter (для этого в некоторых дистрибутивах Linux можно использовать команду `yum tkinter`, более подробные сведения об установке вы найдете в приложении А). В системе Mac OS X все необходимое может быть уже установлено – поищите на своей машине команду `idle`.

обычный интерактивный сеанс, который играет роль инструмента проведения экспериментов, – программный код, который здесь вводится, немедленно исполняется.

В IDLE присутствуют привычные пункты меню и для выполнения наиболее распространенных операций имеются короткие комбинации клавиш. Чтобы создать (или отредактировать) файл с исходным программным кодом 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 существующий файл). В результате на экране появится новое окно. Это окно текстового редактора среды IDLE, где создается или изменяется программный код.

В книге это недостаточно четко видно, но IDLE обеспечивает *подсветку* синтаксиса программного кода, который вводится как в главном окне, так и во всех окнах текстового редактора, – ключевые слова выделяются одним цветом, литералы другим цветом, и т. д. Это позволяет визуально выделять элементы программного кода.

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 пункту меню горячей комбинацией клавиш). Если с момента открытия или последнего сохранения файла его содержимое изменялось, Python предложит сохранить его.

Когда сценарий запускается таким способом, весь вывод, который он генерирует, а также все сообщения об ошибках появляются в основном окне интерактивного сеанса работы с интерпретатором (командная оболочка Python). Например, на рис. 3.3 последние три строки являются результатом выполнения сценария, открытого в отдельном окне редактирования. Сообщение «RESTART» говорит о том, что пользовательский процесс был перезапущен с целью выполнить отредактированный сценарий, и позволяет визуально отделить вывод, полученный от сценария.



Совет дня: если потребуется повторно выполнить команду в основном окне интерактивного сеанса в среде IDLE, можно воспользоваться комбинацией Alt-P, выполняющей последовательный переход к началу истории команд, и Alt-N, выполняющей переход к концу истории команд (в некоторых системах Mac вместо этих комбинаций могут использоваться комбинации Ctrl-P и Ctrl-N). При нажатии этих комбинаций клавиш вы сможете вызывать предыдущие команды, изменять их и запускать. Кроме того, имеется возможность повторно вызывать команды, позиционируя в них курсор, и использовать операцию «скопировать и вставить», но часто это оказывается более трудоемко, чем ввести вручную. Вне среды разработки IDLE, в ходе интерактивного сеанса работы с интерпретатором в системе Windows, можно повторно возвращаться к предыдущим командам с помощью клавиш управления курсором.

```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****

IDLE 1.2
>>> 2 ** 100
1267650600228229401496703205376L
>>> 'spam!' * 10
'spam! spam! spam! spam! spam! spam! spam! spam! spam! '
>>> x = 'Spam'
>>> x + 'NI'
'SpamNI'
>>>
>>> import os
>>> os.getcwd()
'C:\Python25'
>>> ----- RESTART -----
>>>
I am: __main__
1
bb
>>> |
```

Рис. 3.3. Основное окно интерактивного сеанса работы с интерпретатором Python в интегрированной среде IDLE, запущенной под управлением операционной 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)

Использование IDLE

Интегрированная среда разработки IDLE проста в использовании, переносима и доступна в большинстве платформ. Я обычно рекомендую ее тем, кто только начинает программировать на языке Python, потому что она упрощает некоторые аспекты и не предполагает наличия опыта работы с системной командной строкой. Но, по сравнению с некоторыми коммерческими интегрированными средами разработки, она имеет некоторые ограничения. Ниже приводится список особенностей, которые должны приниматься во внимание начинающими пользователями 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 Windows. 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 сможете 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→Run Module в окне редактирования, а не за счет их импортирования или перезагрузки в окне интерактивного сеанса.** Ранее в этой главе было показано, что вполне возможно запустить файл, выполнив операцию импортирования 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→Run Module всегда приводит к запуску текущей версии файла. Кроме того, в случае необходимости будет предложено сохранить файл (еще одна распространенная ошибка при работе вне среды IDLE).
- **Вам по-прежнему может потребоваться выполнять перезагрузку импортируемых модулей.** Пункт меню Run→Run Module в среде IDLE всегда запускает текущую версию только файла верхнего уровня. Если изменениям подвергались модули, импортируемые сценарием, их необходимо будет перезагрузить вручную в интерактивной оболочке. Но, несмотря на это, использование пункта меню Run→Run Module позволяет избавиться от некоторых ошибок, связанных с операцией импортирования. Если вы предпочитаете использовать операции 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.** Похоже, что эта возможность является наиболее востребованной (возможно потому, что она присутствует в похожих интегрированных средах разработки), и в конечном счете когда-нибудь она будет добавлена. Однако в настоящее время нет никакой возможности выполнить очистку окна интерактивного 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.
- **Многопоточные 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 некоторых типов программ на языке Python, использующих библиотеку Tkinter. 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 процесса, но некоторые программы по-прежнему могут вызывать зависание графического интерфейса IDLE. Ваш программный код может и не вызывать 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 использовать другие способы, например щелчком мыши на ярлыке или из системной 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 графическим интерфейсом среды разработки, на определенных платформах могут проявляться проблемы с запуском (особенно часто проблема с запуском встречается на некоторых машинах Windows). Если вам доведется столкнуться с такими ошибками, попробуйте запустить IDLE из командной строки, что вынудит ее запускаться в виде единственного процесса и позволит избежать проблем с поддержанием соединения: для принудительного запуска в этом режиме используйте флаг `-n`. Например, в операционной системе Windows откройте программу Командная строка (Command Prompt) и запустите команду `idle.py -n` из каталога `C:\Python25\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 пределами графического интерфейса среды разработки. Например, IDLE запускает 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 может быть удобно, но может вызывать проблемы при работе вне среды IDLE, потому что в этом случае всегда необходимо импортировать имена из используемых файлов.

Дополнительные возможности IDLE

Помимо основных функций редактирования и запуска среда IDLE предоставляет целый ряд дополнительных возможностей, включая отладчик и инспектор объектов. Отладчик IDLE активируется с помощью меню Debug (Отладка), а инспектор объектов – с помощью меню File (Файл). Инспектор объектов позволяет переходить, перемещаясь по пути поиска модулей, к файлам и объектам в файлах – щелчок на файле или объекте приводит к открытию соответствующего исходного текста в окне редактирования.

Режим отладки в IDLE иницируется выбором пункта меню Debug→Debugger (Отладка→Отладчик) главного окна, после этого можно запустить отлаживаемый сценарий выбором пункта меню Run→Run Module (Запустить→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 сценария, просматривать значения переменных и т. д. Кроме того, вы сможете следить за ходом выполнения программ – в этом случае текущая выполняемая 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 программного кода, которая вызвала эту ошибку. Это позволяет быстро выяснить источник ошибки и ликвидировать ее. Помимо этого, текстовый редактор IDLE обладает обширным набором возможностей, которые пригодятся программистам, включая автоматическое оформление отступов, расширенный поиск текста и файлов и многое другое. Интегрированная среда IDLE обеспечивает интуитивно понятный графический интерфейс, и потому вы можете поэкспериментировать с ней, чтобы получить представление об имеющихся возможностях.

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 использовать интегрированную среду разработки. Я еще раз рекомендую использовать IDLE для выполнения упражнений из этой книги, если вы только начинаете знакомство с языком Python и пока не знакомы с принципами разработки, основанными на применении командной строки. Однако существует еще несколько 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 интегрированных сред разработки:

Eclipse и PyDev

Eclipse – это улучшенная и свободно распространяемая интегрированная среда разработки с графическим интерфейсом. Первоначально она создавалась 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, возможности которой намного шире возможностей IDLE. Один из ее недостатков заключается в том, что она слишком велика, а модуль расширения PyDev для получения дополнительных возможностей (включая интегри-

рованную 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, редактирование текста, отладка и другие. Кроме того, Komodo обладает множеством дополнительных возможностей, отсутствующими 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, реализующий графические интерфейсы, создаваемые в интерактивном режиме. К моменту написания этих строк среда Komodo не являлась свободно распространяемой. Найти ее можно на сайте <http://www.activestate.com>.

PythonWin

PythonWin – это свободно распространяемая интегрированная среда разработки на языке Python для операционной системы Windows. Она распространяется 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 полезных расширений, специфичных для Windows, например, PythonWin обладает поддержкой COM-объектов. В настоящее время IDLE обладает более широкими возможностями, чем PythonWin (например благодаря тому, что IDLE использует для своей работы два процесса, она реже зависает). Однако PythonWin предлагает инструменты для разработки программ под Windows, которые отсутствуют в IDLE. Более подробную информацию вы найдете на сайте <http://www.activestate.com>.

Прочие

Существует еще примерно с полдесятка других известных мне интегрированных 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 отдельных расширений. Например, редакторы Emacs и Vim обладают существенной 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 editors» (редакторы Python) – это должно привести вас на страницу Wiki, где содержится информация о множестве интегрированных сред разработки и текстовых редакторов для Python.

Встраивание вызовов

К настоящему моменту мы рассмотрели, как выполнять программный код в интерактивной командной оболочке интерпретатора и как запускать программный код, сохраненный в файлах, из системной командной строки, из исполняемых сценариев в системе UNIX, щелчком мыши, с помощью операции импортирования модулей и в интегрированной среде разработки, такой как IDLE. Это подавляющее большинство способов, которые встретятся вам в этой книге.

Но в некоторых особых случаях программный код на языке Python может также запускаться 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 и т. д. В этом случае уже не вы, а другая программа предлагает интерпретатору выполнить программный код, созданный вами. Такой способ запуска программного кода обычно используется для обеспечения поддержки возможности настройки у конечного пользователя, например, игровая программа может позволять изменять ход игры, запуская в ключевые моменты внедренный программный код пользователя на языке Python.

Например, вполне возможно создавать и выполнять строки программного кода Python из программ на языке C, вызывая функции API времени выполнения интерпретатора Python (набор служб, экспортируемых библиотеками, созданными при компиляции Python на вашей машине):

```
#include <Python.h>
...
Py_Initialize();
PyRun_SimpleString("x = brave + sir + robin");
```

В этом фрагменте программа на языке C, скомпонованная с библиотеками Python, инициализирует интерпретатор и передает ему для выполнения строку с инструкцией присваивания. Программы на языке C могут также получать доступ к объектам Python и взаимодействовать с ними, используя другие функции API языка Python.

Эта книга не описывает вопросы интеграции Python/C, но вы должны знать, что в зависимости от того, как ваша организация планирует использовать Python, вы можете оказаться одним из тех, кому действительно придется запускать программы на языке Python.¹ При этом бо-

¹ О встраивании программного кода Python в программы на языке C/C++ подробно рассказывается в книге «Programming Python» (O'Reilly). Используя прикладной интерфейс встраиваемого интерпретатора, вы сможете напрямую вызывать функции Python, загружать модули и производить прочие действия. Кроме того, система Jython позволяет программам на языке Java вызывать программный код на языке Python, используя прикладной интерфейс на языке Java (класс интерпретатора Python).

лее чем вероятно, что вы по-прежнему сможете использовать интерактивную оболочку интерпретатора и приемы запуска файлов, описанные выше, чтобы протестировать программный код отдельно от программ, куда этот код внедряется.

Фиксированные исполняемые двоичные файлы

Фиксированные исполняемые двоичные файлы, описанные в предыдущей главе, представляют собой комбинацию байт-кода программы и интерпретатора Python, объединенных в одном исполняемом файле. Благодаря этому такие программы 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 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 подробная информация об этой возможности приводится в предыдущей главе.

Возможность запуска программ из текстового редактора

Как упоминалось ранее, большинство текстовых редакторов для программистов хотя и не являются полноценными интегрированными средами разработки, тем не менее поддерживают возможность редактирования и запуска программ на языке Python. Такая поддержка может быть изначально встроена 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 способы запуска программ Python. Например, в некоторых системах Macintosh выполнить программу на языке Python можно, перетащив мышью ярлык файла программы на ярлык интерпретатора Python. В Windows сценарии можно запускать с помощью пункта Выполнить... (Run...) меню кнопки Пуск (Start). Наконец, в состав стандартной библиотеки Python входят вспомогательные функции, позволяющие запускать программы на языке Python из других программ на языке

Python (такие как `execfile`, `os.popen`, `os.system`), однако обсуждение этих функций выходит за рамки этой главы.

Будущие возможности

В этой главе отражены существующие ныне способы запуска, многие из которых характерны для определенной платформы и в определенное время. В действительности многие методы запуска и выполнения, представленные здесь, появились между выпусками разных изданий этой книги. Поэтому вполне возможно, 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 программы на языке Python способом, имеющим смысл для машин, которые вы используете сейчас или будете использовать в будущем – стилем 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 исполняемые файлы, которые запускаются так же, как и фиксированные двоичные файлы ныне). Если бы я знал, что будет в будущем, я бы, наверное, начал переговоры с биржевым маклером, а не писал бы сейчас этих слов!

Какие способы следует использовать?

Ознакомившись с таким богатством возможностей, возникает вполне естественный вопрос – какой способ лучше? Вообще, если вы начинающий разработчик, для вас было бы желательно использовать интегрированную среду разработки IDLE. Она предоставляет дружелюбный графический интерфейс и скрывает некоторые детали, связанные с необходимостью настройки. Кроме того, в ее состав входит платформонезависимый текстовый редактор, 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 использовать командную строку системы или щелчок мышью на ярлыке (именно таким способом автор разрабатывает программы на языке Python, но это привычка, 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, которая вам нравится.

В заключение

В этой главе мы познакомились с наиболее часто используемыми способами запуска программ на языке Python: запуск программного кода в интерактивном сеансе работы с интерпретатором и запуск файлов с программным кодом из системной командной строки, щелчком мыши на ярлыке файла, за счет выполнения операции импортирования и с помощью интерфейса интегрированной среды разработки, такой как IDLE. Мы охватили здесь значительную часть темы запуска. Цель этой главы состояла в том, чтобы дать вам достаточный объем знаний, владея которыми, вы сможете приступить к работе с программным кодом, который мы начнем писать в следующей части книги. В этой части мы приступим к изучению самого языка Python, начав 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 усвоение всех тем, рассматривавшихся в этой части. За справками и для того, чтобы освежить свои знания, обращайтесь к приложению В.

Закрепление пройденного

Контрольные вопросы

1. Как запустить интерактивный сеанс работы с интерпретатором?
2. Где следует вводить команду, которая запустит файл сценария?
3. Назовите две ловушки, связанные с щелчком мыши на ярлыках в Windows.
4. Почему может потребоваться перезагрузить модуль?
5. Как запустить сценарий из среды разработки IDLE?
6. Назовите две ловушки, связанные со средой разработки IDLE.
7. Что такое пространство имен, и какое отношение они имеют к файлам модулей?

Ответы

1. В операционной системе Windows интерактивный сеанс работы с интерпретатором можно запустить, щелкнув на кнопке Пуск (Start), открыть пункт меню Все программы (All Programs), выбрать пункт меню Python и затем щелкнуть на пункте меню Python (command line) (Python (командная строка)). Тот же результат можно получить в Windows и на других платформах, введя команду `python` в системной командной строке, в окне консоли (Командная строка (Command Prompt) в Windows). Как вариант, можно запустить интегрированную среду

разработки IDLE, главное окно которой представляет собой интерактивную командную оболочку интерпретатора. Если в вашей системе переменная окружения `PATH` не включает каталог, в который был установлен интерпретатор Python, вам может потребоваться выполнить команду `cd`, чтобы перейти в каталог установки Python, или указать полный путь к команде `python` (например, `C:\Python25\python` в Windows).

2. Системные команды вводятся в программе, которая в вашей системе используется в качестве консоли: Командная строка (Command Prompt) в Windows; `xterm` или окно терминала в UNIX, Linux и Mac OS X; и т. д.
3. Если сценарий просто выводит какие-то данные и завершает работу, окно с этой информацией исчезает немедленно, еще до того, как вы сможете увидеть, что было выведено (поэтому в таких ситуациях удобно использовать функцию `raw_input`). Сообщения об ошибках, возникших в ходе работы сценария, также приводят к немедленному закрытию окна еще до того, как вы успеете исследовать его содержимое (поэтому предпочтительнее в ходе разработки использовать системную командную строку или среду разработки IDLE).
4. 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 перезагрузку, модуль должен быть импортирован. Запуск программного кода из системной командной строки, щелчком мыши на ярлыке или в интегрированной среде разработки, такой как IDLE, обычно вообще снимает эту проблему, так как в таких случаях система каждый раз выполняет текущую версию программного кода.
5. В окне редактирования текста для файла, который требуется запустить, выберите пункт меню `Run→Run Module` (Запустить→Запустить модуль). В результате программный код в окне редактирования будет запущен как файл сценария верхнего уровня, а вывод, сгенерированный с этим сценарием, появится в главном окне интерактивной командной оболочки Python.
6. Интегрированная среда разработки IDLE может зависать при запуске некоторых типов программ, особенно с графическим интерфейсом пользователя, которые выполняются в нескольких потоках (рассмотрение усовершенствованной методики выходит за рамки данной книги). Кроме того, среда IDLE обладает некоторыми удобными особенностями, которые не поддерживаются при работе вне этой среды: переменные сценария автоматически импортируются в интерактивную командную оболочку IDLE, но в командной строке Python такая возможность отсутствует.

7. Пространство имен – это просто пакет переменных (то есть имен). В Python он приобретает форму объекта с атрибутами. Каждый файл модуля автоматически становится пространством имен, то есть пакетом переменных, отражающих выполненные операции присваивания на верхнем уровне файла. Пространства имен позволяют избежать конфликтов имен в программах на языке Python: поскольку каждый модуль – это самостоятельное 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 представлены 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 приложении с решениями (приложение В), где приводятся ответы – упражнения и их решения иногда содержат дополнительные сведения, не рассматривавшиеся в основном тексте части, поэтому вам определенно следует ознакомиться с ответами, даже если вам удастся ответить на вопросы самостоятельно.

1. *Взаимодействие.* Используя системную командную строку, IDLE или другой 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 потребоваться сначала выполнить команду `cd`, ввести полный путь к каталогу, куда был установлен выполняемый файл интерпретатора Python, или добавить путь к этому каталогу в переменную окружения `PATH`. При желании значение переменной `PATH` в системах UNIX можно установить в файле `.cshrc` или `.kshrc`; в Windows для этой цели можно использовать файл `setup.bat`, `autoexec.bat` или выполнить настройку переменной окружения с использованием 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 ярлыке, вызовом интерпретатора Python из командной строки, передав ему имя файла в виде аргумента (например, `python module1.py`) и т. д. Попробуйте поэкспериментировать с разными способами запуска, которые обсуждались 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 импортировать его снова из первоначального каталога (то есть запустите Python в каталоге, где производился импорт в первый раз). Что произошло? (Подсказка: посмотрите, остался ли в первоначальном каталоге файл с байт-кодом *module1.pyc*?)
4. *Сценарии.* Если ваша платформа поддерживает такую возможность, добавьте комбинацию символов `#!` в начало файла модуля *module1.py*, дайте файлу право на выполнение и попробуйте запустить его как обычный исполняемый файл. Что должна содержать первая строка? Обычно комбинация символов `#!` имеет особое значение только на платформе UNIX, Linux и других UNIX-подобных системах, таких как MAC OS X. Если вы работаете в Windows, попробуйте просто запустить файл, введя его имя без предшествующего ему слова «python» (этот способ работает в последних версиях Windows), или с помощью диалога Пуск→Выполнить... (Start→Run...).
5. *Ошибки.* Поэкспериментируйте с математическими выражениями и операциями присваивания в интерактивной командной оболочке Python. Для начала введите выражение `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 узнаете, что с технической точки зрения ваши действия привели к вызову того, что известно под названием *обработчик исключений по умолчанию*, – программного кода, который выводит стандартные сообщения об ошибках. Если вы не выполняете перехват ошибок в своих программах, это за вас сделает обработчик по умолчанию, который выведет сообщение об ошибке.

Для отладки сложного программного кода среда разработки IDLE предоставляет отладчик с графическим интерфейсом (описывается в разделе «Дополнительные возможности IDLE», в этой же главе), а в состав стандартной библиотеки модулей Python входит модуль с именем `pdb`, реализующий отладчик с интерфейсом командной строки. (Подробнее о модуле `pdb` можно узнать из руководства к стандартной библиотеке.) Для начала вам вполне будет достаточно стандартного механизма обработки ошибок – он позволит узнать причину ошибки, а также покажет, какие строки кода выполнялись в момент ее появления.

6. *Прерывание программы.* В командной строке интерпретатора Python введите следующие инструкции:

```
L = [1, 2]
L.append(L)
L
```

Что произошло? Если вы используете версию Python более новую, чем Release 1.5, скорее всего вы увидите кажущийся странным результат, который будет описан в следующей части книги. При использовании версий Python, старше 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 комбинации клавиш Ctrl-C?



Если вы используете версию Python более старую, чем 1.5.1, прежде чем выполнить это упражнение, обязательно проверьте, имеется ли возможность прерывать работу программ комбинацией клавиш Ctrl-C, в противном случае вам придется ждать очень долго.

7. *Документация.* Потратьте по крайней мере 17 минут на исследование библиотеки Python и руководства по языку программирования, чтобы получить представление о стандартной библиотеке и о структуре комплекта документации. Вам нужно понять по крайней мере, где в руководстве находятся описания основных тем. После этого вы легко сможете отыскать интересующую вас информацию. В системе Windows 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 Интернете 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 14. Если у вас есть свободное время, займитесь исследованием веб-сайтов Python и Vaults of Parnassus, а также веб-сайта расширений сторонних разработчиков PyPy. В частности, ознакомьтесь со страницами документации и поиска на сайте *Python.org* – они могут оказаться для вас весьма важными ресурсами.

II

Типы и операции

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 некоторые действия над чем-то. «Некоторые действия» принимают форму операций, таких как сложение или конкатенация, а под «чем-то» подразумеваются *объекты*, над которыми выполняются операции. В этой части книги мы сосредоточимся на этом «что-то» и на действиях, которые могут выполняться программой.

Говоря более формальным языком, данные в языке Python представлены в форме *объектов* – встроенных, предоставляемых языком Python, или объектов, которые мы создаем с помощью других инструментов, таких как библиотеки расширений, написанные на языке C. Мы уточним это определение позднее, но если говорить по сути, объекты – 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 объектных 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, выражения 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.

Зачем нужны встроенные типы?

Если вам приходилось использовать языки программирования более низкого уровня, такие как С или С++, то вы уже знаете, что значительная доля работы приходится на реализацию *объектов*, известных также как *структуры данных*, которые предназначены для представления компонентов внутри приложений. В таких языках программирования необходимо заниматься проработкой структур данных, управлять выделением памяти, реализовывать функции поиска и доступа к элементам структур и т. д. 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 непосредственно в язык, поэтому обычно нет никакой необходимости создавать собственные реализации объектов, предназначенных для решения поставленных задач. Фактически, если у вас нет потребностей в специальных видах обработки, которые не обеспечиваются встроенными типами объектов, вам всегда лучше использовать встроенные объекты вместо реализации своих собственных. И вот почему:

- **Встроенные объекты упрощают создание программ.** Для решения простых задач часто вполне достаточно встроенных типов для представления структур данных проблемной области. В вашем распоряжении имеются такие мощные инструментальные средства, как коллекции (списки) и таблицы поиска (словари), поэтому вы можете использовать их непосредственно. Благодаря встроенным объектным типам языка Python вы можете выполнить значительный объем работы.
- **Встроенные объекты – это компоненты расширений.** Для решения сложных задач вы можете создавать собственные объекты, используя для этого классы языка Python или интерфейсы языка С. Однако, как будет показано ниже, объекты, реализованные вручную, обычно основаны на таких встроенных 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 оптимизированные структуры данных, реализованные на языке С для достижения 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 степени Python многое заимствует как из языков, полагающихся на использование встроенных инструментальных средств (таких как LISP), так и полагающихся на мастерство программиста, который должен выполнить собственную реализацию инструментов и структур данных (таких как C++). В языке Python вы можете создавать собственные типы объектов, но в самом начале делать это не рекомендуется. Более того, из-за того, что встроенные компоненты являются стандартными составляющими языка Python, они всегда остаются неизменными, тогда как собственные структуры имеют свойство изменяться от случая к случаю.

Другими словами, встроенные типы объектов не только упрощают процесс программирования, но они обладают большей эффективностью и производительностью, чем большинство типов, созданных вручную. Даже если вы создаете 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 приходилось работать с другими языками программирования. Например, числа и строки представляют числовые и текстовые значения соответственно, а файлы обеспечивают интерфейс для работы с файлами, хранящимися в компьютере.

Таблица 4.1. Некоторые встроенные объекты

Тип объекта	Пример литерала/создания
Числа	1234, 3.1415, 999L, 3+4j, Decimal
Строки	'spam', "guido's"
Списки	[1, [2, 'three'], 4]
Словари	{'food': 'spam', 'taste': 'yum'}
Кортежи	(1, 'spam', 4, 'U')
Файлы	myfile = open('eggs', 'r')
Прочие типы	Множества, сами типы, None, логические значения

¹ В этой книге под термином *литерал* подразумевается выражение, создающее объект, которое иногда также называется *константой*. Следует иметь в виду, что термин «константа» не означает объектов и переменных, которые никогда не изменяются (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 этой же главе).

Таблица 4.1 содержит далеко не полный список, потому что объектами являются все данные, которые приходится обрабатывать в программах на языке Python. Например, когда на языке Python мы реализуется поиск текста по шаблону – создаются объекты шаблонов, когда программируются сетевые взаимодействия – используются объекты сокетов. Существуют и другие типы объектов, которые создаются в результате импорта и использования модулей, и все они обладают своим собственным поведением.

Типы объектов, перечисленные в табл. 4.1, мы называли *базовыми*, потому 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 генерирует и возвращает новый строковый объект. Такова специфика синтаксиса Python создания этого объекта. Похожим образом выражение, заключенное в квадратные скобки, создает список, заключенное в фигурные скобки – словарь и т. д. Хотя, как вы сможете убедиться, в языке Python отсутствует конструкция объявления типа, сам синтаксис выполняемых выражений задает типы создаваемых и используемых объектов. Фактически, выражения, создающие 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 ассоциирован со своим собственным набором операций на протяжении всего времени 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 определяется автоматически, и их не требуется объявлять в программном 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, которые обеспечивают поддержку коллекций и функций поиска, которые в низкоуровневых языках программирования приходится реализовывать вручную. Говоря коротко, списки обеспечивают поддержку упорядоченных коллекций других объектов, а словари реализуют возможность хранения объектов по ключам. И списки, и словари могут быть вложенными, по мере необходимости могут увеличиваться и уменьшаться в размерах и могут содержать объекты любых типов.

В последующих главах мы подробно изучим каждый из типов объектов, перечисленных в табл. 4.1. Но прежде чем углубляться в детали,

давайте познакомимся с базовыми типами объектов Python в действии. Оставшаяся часть главы представляет собой обзор операций, которые мы более подробно будем исследовать в последующих главах. Не ждите в этой главе исчерпывающих объяснений, поскольку основная ее цель состоит в том, чтобы разжечь ваш аппетит и представить некоторые базовые идеи. Однако 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 набор объектов языка Python включает в себя вполне ожидаемые типы: целые числа (числа без дробной части), числа с плавающей точкой (грубо говоря, числа с десятичной точкой) и более экзотические типы («длинные» целые числа с неограниченной точностью, комплексные числа с мнимой частью, числа с фиксированной десятичной точкой и множества).

Несмотря на наличие некоторых необычных типов, базовые числовые типы в языке Python действительно являются базовыми. Числа в Python поддерживают набор самых обычных математических операций. Например, символ «плюс» (+) означает сложение, символ «звездочка» (*) – умножение, а два символа «звездочка» (**) – возведение в степень:

```
>>> 123 + 222      # Целочисленное сложение
345
>>> 1.5 * 4        # Умножение вещественных чисел
6.0
>>> 2 ** 100       # 2 в степени 100
1267650600228229401496703205376L
```

Обратите внимание на символ *L* в конце результата последней операции: когда возникает необходимость в более высокой точности, Python автоматически преобразует целые числа в «длинные» целые. Например, вы можете попробовать отыскать 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 продолжительное время!). Посмотрите, что происходит при выводе чисел с плавающей точкой:

```
>>> 3.1415 * 2      # герг: в виде кода
6.2830000000000004
>>> print 3.1415 * 2 # str: более удобочитаемо
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 восприятия человеком (как во втором результате).

Формально первая форма называется `repr` (объект в виде кода), а вторая, более дружелюбная к пользователю, — `str`. Различия между ними станут более понятны, когда мы приступим к изучению классов, а пока, если что-то выглядит непонятным, попробуйте вывести тот же результат с помощью инструкции `print`.

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, такие как комплексные числа, числа с фиксированной десятичной точкой и множества, а среди свободно распространяемых 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 произвольных последовательностей байтов. Это наш первый пример *последовательностей*, или упорядоченных коллекций других объектов по их позиции, в языке Python. Последовательности поддерживают порядок размещения элементов, которые они содержат, слева направо: элементы сохраняются и извлекаются, исходя из их позиции в последовательности. Строго говоря, строки являются последовательностями односимвольных строк. Другими типами последовательностей являются списки и кортежи (будут описаны ниже).

Операции над последовательностями

Будучи последовательностями, строки поддерживают операции, предполагающие определенный порядок позиционирования элементов.

Например, если имеется четырехсимвольная строка, то с помощью встроенной функции `len` можно проверить 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'
```

В языке Python индексы реализованы в виде смещений от начала, и потому индексация начинается с 0: первый элемент имеет индекс 0, второй – 1 и т. д. В языке Python предусмотрена возможность индексации в обратном порядке, от конца:

```
>>> 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 проще записывается и более понятна, чем вторая):

```
>>> 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 этому общему принципу.

В дополнение к простой возможности индексирования по номеру позиции последовательности поддерживают более общую форму индексирования, известную как *получение среза (slicing)*, которая обеспечивает возможность извлечения за одну операцию целого раздела (среза).

```
>>> 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 *столбца* из строки за один шаг. В общем виде синтаксис операции получения среза выглядит как `X[I:J]` и означает: «извлечь из X все, считая 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 штук) в виде новой строки. В результате получается срез, или выборка двух символов из середины.

При выполнении операции получения среза левая граница по умолчанию принимается равной нулю, а правая – длине последовательности, к которой применяется операция. В результате мы получаем следующие наиболее распространенные 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 отрицательные индексы и что последняя операция фактически создает копию всей строки. Как мы узнаем позднее, нет смысла копировать строки таким способом, но такая форма копирования очень удобна при работе с другими последовательностями, такими как списки.

Наконец, будучи последовательностями, строки поддерживают операцию *конкатенации*, которая записывается в виде знака плюс (объединение двух строк в одну строку), и операцию *повторения* (новая строка создается за счет многократного 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 универсальное свойство языка Python, которое далее в книге будет названо *полиморфизмом*, означает, что фактически выполняемая операция зависит от объектов, которые принимают в ней участие. Как будет показано, когда мы приступим к изучению динамической типизации, такой полиморфизм в значительной степени обеспечивает выразительность и гибкость программного кода на языке Python. Поскольку отсутствуют ограничения, связанные с типами, операции в языке Python обычно в состоянии автоматически обрабатывать объекты самых разных типов, при условии, что они поддерживают совместимый

интерфейс (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 операций не изменяла оригинальную строку. Все операции над строками в результате 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 не нужными больше объектами (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
'zspam'
```

Все объекты в языке Python либо относятся к классу неизменяемых, либо нет. Если говорить о базовых типах, то числа, строки и кортежи являются неизменяемыми, 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 действительности являются операциями над последовательностями, то есть эти операции 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, выполненные в виде *методов* (функции, присоединенные к объекту, которые запускаются вызовом).

Например, метод строк `find` выполняет поиск подстроки в строке (он возвращает смещение переданной ему подстроки или -1, если поиск не увенчался успехом), 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, мы не изменяем, применяя методы, оригинальную строку, а создаем новую, т. к. строки являются неизменяемыми, и это следует учитывать. Строковые методы — это первый уровень 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, алфавитные символы и т. д.) и отсекают пробельные символы с обоих концов строк.

```
>>> line = 'aaa,bbb,cccc,dd'
>>> line.split(',')         # Разбивает строку по разделителю и создает список строк
['aaa', 'bbb', 'cccc', 'dd']

>>> S = 'spam'
>>> S.upper()               # Преобразование символов в верхний и в нижний регистр
'SPAM'

>>> S.isalpha()             # Проверка содержимого: isalpha, isdigit и т. д.
True

>>> line = 'aaa,bbb,cccc,dd\n'
>>> line = line.rstrip()    # Удаляет завершающие пробельные символы
>>> line
'aaa,bbb,cccc,dd'
```

Следует заметить, что в отличие от универсальных операций, применяемых к последовательностям, строковые методы могут применяться только к строкам и ни к каким другим объектам. Следует понимать, что инструментальные средства языка Python делятся на несколько уровней: универсальные операции, которые 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.

ми. Вообще эта книга не является исчерпывающим источником информации о методах объектов. Чтобы получить дополнительную информацию, вы всегда можете воспользоваться функцией `dir`, которая возвращает список всех доступных атрибутов заданного объекта. Поскольку методы являются атрибутами-функциями, они также присутствуют в списке:

```
>>> dir(S)
['_add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
 '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__',
 '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__rmod__', '__rmul__', '__setattr__', '__str__', 'capitalize', 'center',
 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'index',
 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper',
 'join', 'ljust', 'lower', 'lstrip', '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 возможности перегрузки операторов в классах, — они представляют собой реализацию строкового объекта и доступны для поддержки настроек. Вообще ведущие и завершающие символы подчеркивания используются для обозначения механизмов внутренней реализации. Имена без символов подчеркивания в этом списке обозначают методы строковых объектов.

Функция `dir` возвращает лишь имена методов. Чтобы узнать назначение того или иного метода, можно передать его имя функции `help`:

```
>>> help(S.index)
Help on built-in function index:
(Справка о встроенной функции index:)

index(...)
    S.index(sub [,start [,end]]) -> int

    Like S.find() but raise ValueError when the substring is not found.
    (Сходна с функцией S.find(), но вызывает ошибку ValueError,
     если подстрока не найдена)
```

Функция `help` — один из немногих интерфейсов к системе программного кода Python, который поставляется в составе интерпретатора под названием *PyDoc* — инструмент, который позволяет извлекать описание из объектов. Далее в этой же книге вы узнаете, что PyDoc позволяет отображать информацию в формате HTML.

Можно запросить информацию и для самого строкового объекта (например, `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 продемонстрировано выше.

За дополнительной информацией всегда можно обратиться к справочному руководству по стандартной библиотеке или к печатным справочным изданиям, но функции `dir` и `help` в языке Python представляют собой самое первое средство 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 программирования Python предоставляет несколько различных способов представления строк в программном коде, которые мы будем исследовать позднее (включая служебные символы, которые представлены, например, в виде последовательностей, 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. Когда используется такая форма, все строки в программном коде объединяются в одну строку, а там, где в исходном тексте выполняется переход на новую строку, вставляется символ «конец строки». Это незначительное синтаксическое удобство весьма полезно для оформления в сценариях на языке Python крупных блоков текста, таких как разметка HTML или XML:

```
>>> msg = """
aaaaaaaaaaaaa
bbb'``'bbbbbbbbbb'``'bbbbbb'bbbb
cccccccccccccc"""

>>> msg
'\naaaaaaaaaaaa\nbbb'\``'bbbbbbbbbb'``'bbbbbb'\bbbb\ncccccccccccccc'
```

Кроме того, Python предоставляет поддержку «неформатированных» строковых литералов, в которых символ обратного слеша интерпретируется как обычный символ (они начинаются с символа `r`), а также поддержку строк с символами Юникода, обеспечивающих интернационализацию (такие строки начинаются с символа `u` и состоят из многобайтовых символов). С технической точки зрения строка с символами Юникода представляет собой иной тип данных, отличный от обычной строки, но он поддерживает те же самые строковые опера-

ции. В последующих главах мы встретимся с этими специальными 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, выходит за рамки этой книги, но для читателей, знакомых с другими языками сценариев, 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 замены, но за счет использования шаблонов мы можем использовать более общие варианты решения задач:

```
>>> 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 сохранены, 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 шаблон 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 обработки текста, но в языке Python имеется поддержка еще более сложных механизмов, включая возможность обработки естественного языка человеческого общения. Впрочем, для этого руководства я и так сказал уже достаточно об обработке 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 отличие от строк, списки являются изменяемыми – они могут модифицироваться 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, посвященном 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 т. д., 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]

>>> 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 программирования, 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 число с плавающей точкой). Кроме того, размер списков не ограничен, благодаря чему они могут увеличиваться и уменьшаться по мере необходимости 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 т. д. 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 элементы 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 все равно не допускает возможности обращаться к несуществующим элементам списка. Обращение к элементам списка по индексам, значения которых выходят за пределы списка, всегда является ошибкой:

```
>>> L
[123, 'spam', 'NI']

>>> L[99]
...текст сообщения об ошибке опущен...
IndexError: list index out of range

>>> 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, чтобы увеличить размер списка, интерпретатор Python сообщает 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 т. д.). Одно из очевидных применений этой особенности — представление матриц, или «многомерных массивов» в языке Python. Делается это с помощью списка, содержащего вложенные списки:

```
>>> M = [[1, 2, 3],      # Матрица 3x3 в виде вложенных списков
          [4, 5, 6],
```

```
[7, 8, 9]]
```

```
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Здесь мы реализовали список, состоящий из трех других списков. В результате была получена матрица чисел 3×3. 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 вторая – третий элемент в этой строке. Соединение операций индексирования позволяет все дальше и дальше погружаться вглубь вложенной структуры объектов.¹

Дополнительные операции над списками

Помимо обычных операций над последовательностями и методов списков, Python предоставляет возможность выполнять более сложные операции над списками, известные как *выражения генераторов списков* (*list comprehension expression*), которые представляют эффективный способ обработки таких структур, как наша матрица. Предположим, например, что нам требуется извлечь из нашей матрицы второй столбец. Строку легко можно получить, выполнив операцию индексирования, потому что матрица хранится в виде строк, однако, благодаря генераторам списков, получить столбец ничуть 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 элемента в последовательности, по одному за раз, слева

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 инструменты 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 информацию об этом расширении вы без труда найдете в Сети.

направо. Генераторы списков заключены в квадратные скобки (чтобы отразить тот факт, что они создают список) и составлены из выражения и конструкции цикла, которые используют одно и то же имя переменной (в данном случае `row`). В предыдущем примере генератор списков интерпретируется так: «Получить элементы `row[1]` из каждой 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 форму:

```
>>> [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 результата нечетных чисел с помощью операции деления по модулю — `%` (остаток от деления). Генераторы списков, применяемые к спискам, возвращают в качестве результатов новые списки, но могут использоваться и для любых других объектов, допускающих выполнение итераций, например, ниже показано использование 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, на мой взгляд, достаточно сложная тема, чтобы говорить о них здесь более подробно. Главная цель этого краткого обзора состоит в том, чтобы проиллюстрировать наличие как простых, так и очень сложных инструментов в арсенале Python. Генераторы списков можно и не использовать, но на практике они оказываются очень удобными и нередко обеспечивают более высокую производительность при работе со списками. Кроме того, их можно применять к любым другим последовательностям в языке Python, а также к некоторым другим типам, которые не являются последовательностями. Мы еще будем говорить об этих выражениях далее, в этой книге.

Словари

Словари в языке Python — это нечто совсем иное (по выражению Монтти Пайтона), они вообще не являются последовательностями, это то, что известно, как *отображения*. Отображения — это коллекции объектов,

но доступ к ним осуществляется не по определенным смещениям от начала коллекции, а по ключам. В действительности отображения вообще не подразумевают какого-либо упорядочения элементов по их позиции, они просто отображают ключи на связанные с ними значения. Словари – единственный тип отображения в наборе базовых объектов Python – также относятся к классу изменяемых объектов: они могут изменяться непосредственно, и в случае необходимости могут увеличиваться и уменьшаться в размерах, подобно спискам.

Операции отображения

Когда словарь определяется как литерал, программный код определения заключается в фигурные скобки и состоит из последовательности пар «ключ: значение». Словари удобно использовать всегда, когда возникает необходимость 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 значения, связанные с ключами. Для доступа к элементам словаря используется тот же синтаксис, который используется для обращения к элементам последовательностей, только в квадратных скобках указывается не смещение относительно 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, заключенного 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 отличие от списков, не допускающих присваивания значений отсутствующим элементам, присваивание значения по несуществующему ключу в словаре приводит 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 описывает некоторого человека. В других приложениях словари могут использоваться 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 имеет более сложную структуру. Возможно, придется записать имя и фамилию, а также несколько названий должностей, занимаемых одновременно. Это приводит к необходимости использования вложенных объектов Python. Словарь 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 (ключи «name» (имя), «job» (должность) и «age» (возраст)), однако значения имеют более сложную структуру: для описания имени человека используется вложенный 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 расширение 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 позже, в этой же книге).

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 потребовалось бы приложить больше усилий и написать больше программного кода: нам пришлось бы описать и объявить структуры и массивы, заполнить их значениями, связать их между собой и т. д. В языке Python все это делается 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 теряется последняя ссылка на объект, например в случае присваивания переменной какого-либо другого значения:

```
>>> res = 0      # Теперь память, занятая объектом, будет освобождена
```

С технической точки зрения, интерпретатор Python обладает такой особенностью, как *сборка мусора*, благодаря которой в ходе выполнения программы производится освобождение неиспользуемой памяти, что освобождает нас от необходимости предусматривать специальные действия в программном коде. Интерпретатор освобождает память сразу же, как только будет ликвидирована последняя ссылка на объект. С работой этого механизма мы познакомимся далее, в этой же книге, а пока достаточно знать, что вы можете работать с объектами, 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 последовательностями, они не предусматривают какой-либо надежный способ упорядочения позиций элементов. Это означает, что если мы создадим словарь и попытаемся вывести его, порядок следования ключей при выводе может не совпадать с порядком, в каком они определялись:

¹ В качестве примечания: имейте в виду, что запись `res`, 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 данных, обеспечивающих доступ по ключу. Мы не будем здесь углубляться в подробности, а за дополнительной информацией по этому вопросу обращайтесь к модулям Python – `pickle` и `shelve`.

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

Как же быть, если нам действительно потребуется упорядочить элементы словаря? 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`:

```
>>> Ks = D.keys()      # Неупорядоченный список ключей
>>> Ks
['a', 'c', 'b']

>>> Ks.sort()          # Сортировка списка ключей
>>> Ks
['a', 'b', 'c']

>>> for key in Ks:      # Обход отсортированного списка ключей
    print key, '=>', D[key]

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 главах, с помощью новой встроенной функции `sorted` (функция `sorted` сортирует объекты разных типов и возвращает их):

```
>>> 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 программного 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 отсортированном по ключам виде.

Цикл `for` и родственный ему цикл `while` – это основные способы реализации повторяющихся 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 некоторыми объектами, которые последовательностями не являются. Ниже приводится пример обхода всех символов в строке и вывод их в верхнем регистре:

```
>>> for c in 'spam':  
    print c.upper()  
  
S  
P  
A  
M
```

Инструкции циклов мы рассмотрим позднее, в этой же книге.

Итерации и оптимизация

Если цикл `for` выглядит похожим на генераторы списков, введенные ранее, следовательно, оба эти инструмента должны представлять собой универсальные средства выполнения итераций. Фактически обе конструкции способны работать с любыми объектами, которые поддерживают *протокол итераций* – идею, недавно появившуюся в Python, которая по сути подразумевает наличие в памяти последовательности или объекта, который генерирует по одному элементу в контексте выполнения итерации. Именно поэтому функция `sorted`, которая использовалась в предыдущем разделе, способна работать со словарем непосредственно – нам не требуется вызывать метод `keys` для получения последовательности, потому что словари поддерживают выполнение итераций.

О протоколе итераций я расскажу позднее, в этой же книге. А пока просто запомните, что любой генератор списков, такой, как показано ниже, который вычисляет квадраты чисел в списке:

```
>>> squares = [x ** 2 for x in [1, 2, 3, 4, 5]]  
>>> squares  
[1, 4, 9, 16, 25]
```

всегда можно запрограммировать 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]: # Это список, который участвует в вычислениях  
    squares.append(x ** 2)  
  
>>> squares  
[1, 4, 9, 16, 25]
```

Однако генераторы списков обычно выполняются быстрее (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 производительность 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 производительности следует рассматривать во вторую очередь, уже после того, как будет создана работоспособная программа

и когда проблема производительности программы действительно заслуживает того, чтобы на нее обратили внимание. Но чаще всего ваш программный код будет обладать достаточно 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`

Еще одно замечание о словарях, прежде чем двинуться дальше. Несмотря на то что операция присваивания значений элементам с несуществующими 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']      # При попытке обращения к несуществующему ключу возникает ошибка
...текст сообщения об ошибке опущен...
KeyError: 'f'
```

Программная ошибка при попытке получить значение несуществующего элемента — это именно то, что нам хотелось бы получать. Но в некоторых универсальных программах, когда создается программный код, 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. Метод словаря `has_key` позволяет определить наличие ключа и с помощью условного оператора `if` выполнить тот или иной программный код:

```
>>> D.has_key('f')
False

>>> if not D.has_key('f'):
    print 'missing'

missing
```

Далее в этой книге я расскажу подробнее об инструкции `if` и о 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 программного 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 более предложения `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 ошибок обращения к несуществующим элементам словаря (включая метод `get`, выражение `in` и инструкцию `try`, с которой мы познакомимся в главе 10, которая позволяет перехватывать и обрабатывать исключения), но подробнее об этом мы поговорим в одной из следующих глав. А сейчас рассмотрим кортежи.

Кортежи

Объект-кортеж (`tuple` – произносится как «тьюпл» или «тьюпел», в зависимости от того, у кого вы спрашиваете) в общих чертах напоминает список, который невозможно изменить, – кортежи являются последовательностями, как списки, но они являются неизменяемыми, как строки. Синтаксически, определение кортежа заключается в круглые, а не в квадратные скобки. Они также поддерживают включение объектов различных типов, вложение и операции, типичные 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 т. д.
1
```

Единственное реальное отличие кортежей – это невозможность их изменения после создания. 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
(TypeError: объект 'tuple' не поддерживает операцию присваивания элементам)
```

Для чего нужны кортежи?

Зачем нужен тип, который напоминает список, но поддерживает меньшее число операций? Откровенно говоря, на практике кортежи используются не так часто, как списки, но главное их достоинство – неизменяемость. Если коллекция объектов передается между компонентами программы в виде списка, он может быть изменен любым из компонентов. Если используются кортежи, такие изменения становятся невозможны. То есть кортежи обеспечивают своего рода ограничение целостности, что может оказаться полезным в крупных программах. Далее в книге мы еще вернемся к кортежам. А сейчас перейдем к последнему базовому типу данных – к файлам.

Файлы

Объекты-файлы – это основной интерфейс между программным кодом на языке 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. Например, чтобы создать файл для вывода данных, вместе с именем файла функции необходимо передать строку режима `'w'`:

```
>>> f = open('data.txt', 'w') # Создается новый файл для вывода
>>> f.write('Hello\n')        # Запись строки байтов в файл
>>> f.write('world\n')
>>> 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, файл нужно открыть в режиме `'r'` (этот режим используется по умолчанию, если строка режима в вызове функции опущена), затем прочитать содержимое файла в строку байтов и отобразить ее. Содержимое файла для сценария всегда является строкой байтов, независимо от типов данных, фактически хранящихся в файле:

```
>>> f = open('data.txt') # 'r' - это режим доступа к файлу по умолчанию
>>> bytes = f.read()     # Файл читается целиком в строку
>>> bytes
'Hello\nworld\n'

>>> print bytes          # Вывод, с попутной интерпретацией служебных символов
Hello
world

>>> bytes.split()        # Содержимое файла всегда является строкой
['Hello', 'world']
```

Объекты-файлы имеют и другие методы, поддерживающие дополнительные возможности, рассматривать которые у нас нет времени. 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 т. д.) и другие методы (`seek` – перемещает позицию чтения/записи в файле). В этой книге мы будем изучать их позднее, 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(file)
['_ _class_', '_ _delattr_', '_ _doc_', '_ _enter_', '_ _exit_',
```

```
'_getattribute_', '_hash_', '_init_', '_iter_', '_new_',
'_reduce_', '_reduce_ex_', '_repr_', '_setattr_', '_str_',
'close', 'closed', 'encoding', 'fileno', 'flush', 'isatty', 'mode',
'name', 'newlines', 'next', 'read', 'readinto', 'readline', 'readlines',
'seek', 'softspace', 'tell', 'truncate', 'write', 'writelines', 'xreadlines']

>>> help(file.seek)
...попробуйте и увидите...
```

Другие средства, напоминающие файлы

Функция `open` — это рабочая лошадка в большинстве операций с файлами, которые можно выполнять в языке Python. Для решения более специфичных задач Python поддерживает и другие инструментальные средства, напоминающие файлы: каналы, очереди, сокет, файлы с доступом по ключу, хранилища объектов, файлы с доступом по дескриптору, интерфейсы к реляционным и объектно-ориентированным базам данных и многие другие. Файлы 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 определения этой категории. Например, множества совсем недавно появились в языке. Множества — это контейнеры для других объектов. Множества создаются встроенной функцией `set` и поддерживают типичные математические операции над множествами:

```
>>> X = set('spam')
>>> Y = set(['h', 'a', 'm']) # Создаются два множества последовательностей
>>> X, Y
(set(['a', 'p', 's', 'm']), set(['a', 'h', 'm']))

>>> X & Y                                # Пересечение
set(['a', 'm'])

>>> X | Y                                # Объединение
set(['a', 'p', 's', 'h', 'm'])

>>> X - Y                                # Разность
set(['p', 's'])
```

Кроме того, недавно в Python появились вещественные числа с фиксированной точностью и тип `Boolean` (с предопределенными объектами `True` и `False`, которые по сути являются целыми числами 0 и 1 с некоторыми особенностями отображения на экране), а кроме того, давно уже существует специальный объект `None`:

```
>>> import decimal      # Вещественные числа с фиксированной точностью
>>> d = decimal.Decimal('3.141')
>>> d + 1
Decimal("4.141")

>>> 1 > 2, 1 < 2        # Логические значения
(False, True)
>>> bool('spam')
True

>>> X = None             # Специальный объект None
>>> print X
None

>>> L = [None] * 100     # Инициализация списка из ста объектов None
>>> L
[None, None, None, None, None, None, None, None, None, None, None, None,
...список из 100 объектов None...]

>>> type(L)              # Типы
<type 'list'>

>>> type(type(L))        # Даже типы являются объектами
<type 'type'>
```

Как можно нарушить гибкость программного кода

Мы еще будем много говорить обо всех этих типах данных далее в книге, но сначала я хочу сделать важный акцент. Объект `type` позволяет выполнять проверку типов объектов 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

>>> 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 признаком бывшего программиста на языке C, приступившего к программированию на языке 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 проверок ваш

программный код может оказаться в состоянии работать с более широким диапазоном типов.

Это связано с идеей полиморфизма, о которой упоминалось ранее, и это основная причина отсутствия необходимости описывать типы переменных в языке Python. Как будет говориться далее, программный код на языке Python ориентируется на использование *интерфейсов* объектов (набор поддерживаемых операций), а не их типов. Отсутствие заботы об определенных типах означает, 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 это обычно не способ думать на языке Python. Вы сами убедитесь, что ключевой идеей, обеспечивающей успех использования Python, является полиморфизм.

Классы, определяемые пользователем

Мы рассмотрим объектно-ориентированный стиль программирования на языке Python (необязательную к применению, но весьма мощную особенность языка, которая позволяет сократить время разработки за счет многократного использования программного кода) подробно далее в этой книге. Тем не менее, говоря абстрактными терминами, классы определяют новые типы объектов, которые расширяют базовый набор, и потому они заслуживают упоминания здесь. Например, вам мог бы потребоваться такой тип объектов, который моделировал бы сотрудников. В языке 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
```

Данный класс определяет новый тип объектов, который обладает атрибутами `name` и `pay` (иногда атрибуты называют *информацией о состоянии*), а также два описания поведения, оформленных в виде функций (которые обычно называют *методами*). Обращение к имени класса как к функции приводит 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 присутствует подразумеваемый объект «self»: внутри функций, определяемых в классах, всегда присутствует подразумеваемый объект. В некотором смысле типы, основанные 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`), плюс функции, выполняющие обработку этих двух встроенных объектов.

Кроме того, механизм наследования классов поддерживает программные иерархии, которые допускают возможность их расширения. Возможности программного обеспечения расширяются за счет создания новых классов, но при этом не изменяется программный код, который уже работает. Вместе с тем, 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, определяемых пользователем. Однако все это выходит далеко за рамки ознакомительной главы, поэтому, чтобы продолжить знакомство, вам придется прочитать следующие главы.

И все остальное

Как уже упоминалось ранее, все данные, которые обрабатываются сценариями на языке Python, являются объектами, поэтому наш краткий обзор типов объектов никак нельзя назвать исчерпывающим. Однако, даже при том, что все сущее в языке Python является «объектом», только рассмотренные типы образуют базовый набор. Другие типы объектов, как правило, реализуются в виде модулей и не являются синтаксическими элементами языка. Кроме того, они обычно имеют узкий круг применения – текстовые шаблоны, интерфейсы доступа к базам данных, сетевые взаимодействия и т. д.

Более того, имейте в виду, что объекты, с которыми мы здесь познакомились, действительно являются объектами, но для работы с ними не требуется использовать *объектно-ориентированный* подход – концепция, которая обычно подразумевает использование механизма наследования и оператора `class`, с которыми 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 вашему 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, специфичные 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, последовательности и полиморфизм.

Наряду с этим мы узнали, что базовые типы данных в языке Python обладают большей гибкостью и более широкими возможностями, чем типы данных, доступные в низкоуровневых языках программирования, таких как C. Например, списки и словари избавляют нас от необходимости реализовать программный 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 полную картину. В следующей главе мы приступим к всестороннему изучению чисел 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?
5. Что означает термин «отображение», и какой базовый тип является отображением?
6. Что означает термин «полиморфизм», и почему он имеет такое важное значение?

Ответы

1. К базовым типам объектов (данных) относятся числа, строки, списки, словари, кортежи и файлы. Множества, сами типы, None и логические значения также иногда относят к базовым типам. Существует несколько типов чисел (целые, длинные целые, с плавающей точкой и фиксированной точности) и два типа строк (обычные и в кодировке Юникод).
2. Базовыми 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 типов предусмотрен специальный синтаксис создания объектов, например, 'spam' – это выражение, которое создает строку и определяет 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, строки и кортежи. Но даже при том, что вы не можете изменить «неизменяемый» 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 последовательностей, такие как обращение к элементам по их индексам, конкатенация и получение срезов. Но помимо этого каждый из этих типов имеет ряд специфичных методов.
5. Под термином «отображение» подразумевается объект, который отображает 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.

5

Числа

С этой главы, мы начинаем погружаться в детали реализации языка Python. Данные в этом языке имеют форму объектов – это либо встроенные объекты, входящие в состав самого языка, либо объекты, создаваемые с помощью языковых конструкций. Фактически объекты являются основой любой программы на языке Python. Поскольку объекты представляют самое фундаментальное понятие для программирования на языке Python, все наше внимание в первую очередь будет сосредоточено на объектах.

В предыдущей главе мы коротко познакомились с базовыми типами объектов языка Python. И хотя в этой главе основные понятия были даны, мы старались избегать слишком специфичных подробностей в целях экономии книжного пространства. В этой главе мы приступаем к более внимательному изучению концепции типов данных, чтобы восполнить детали, о которых раньше умалчивалось. Кроме того, здесь мы рассмотрим некоторые типы, которые так или иначе связаны с числами, такие как множества и логические значения (Boolean). Итак, приступим к изучению чисел – первой категории типов данных в языке Python.

Числовые типы в Python

Числовые типы в языке Python не выделяются ничем необычным и наверняка покажутся вам знакомыми, если в прошлом вам уже приходилось использовать какой-либо язык программирования. Числа могут использоваться 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 категорией родственных типов. Язык Python поддерживает обычные

Числовые литералы

Таблица 5.1. Числовые литералы

Литерал	Интерпретация
1234, -24, 0	Обычные целые числа (<code>long</code> в языке C)
999999999999999999L	Длинные целые числа (с неограниченной точностью представления)
1.23, 3.14e-10, 4E210, 4.0e+210	Числа с плавающей точкой (<code>double</code> в языке C)
0177, 0x9ff, 0xFF	Восьмеричные и шестнадцатеричные литералы целых чисел
3+4j, 3.0+4.0j, 3J	Литералы комплексных чисел

Литералы целых чисел и чисел с плавающей точкой

Целые числа записываются как строки, состоящие из десятичных цифр. Числа с плавающей точкой могут содержать символ точки и/или необязательную экспоненту со знаком, которая начинается с символа `e` или `E`. Если в записи числа обнаруживается точка или экспонента, интерпретатор Python создает объект числа с плавающей точкой и использует вещественную (не целочисленную) математику, когда такой объект участвует в выражении. Правила записи

вещественных чисел в языке Python ничем не отличаются от правил, используемых в языке C.

Точность представления целых чисел и длинные целые

Простые целые числа в языке Python (первая строка в табл. 5.1) реализованы как тип `long` в языке C (то есть имеют размер как минимум 32 бита), а вещественные числа в языке Python реализованы как тип `double` в языке C. Таким образом, точность представления чисел этих типов соответствует точности представления, которую обеспечивает компилятор языка C, использованный для сборки интерпретатора Python.¹

Литералы длинных целых чисел

Если числовой литерал заканчивается символом `L` или `l`, он рассматривается интерпретатором Python как длинное целое число (не надо путать с типом `long` в языке C) и может использоваться для представления сколь угодно больших чисел. В Python 2.2 и более поздних версиях 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 потребуются увеличить точность представления.

Шестнадцатеричные и восьмеричные литералы

Правила записи шестнадцатеричных (по основанию 16) и восьмеричных (по основанию 8) чисел в языке Python ничем не отличаются от правил, используемых в языке C. Литералы восьмеричных чисел начинаются с нуля (0), вслед за которым следуют восьмеричные цифры 0–7. Шестнадцатеричные литералы начинаются с комбинации символов `0x` или `0X`, вслед за которыми следуют шестнадцатеричные цифры 0–9 и A–F. Шестнадцатеричные цифры в шестнадцатеричных литералах могут вводиться как в нижнем, так и в верхнем регистре. И восьмеричные, и шестнадцатеричные литералы создают объекты целых чисел – это всего лишь альтернативный синтаксис определения значений этого типа.

Комплексные числа

Литералы комплексных чисел в языке Python записываются в формате `вещественная_часть+мнимая_часть`, где `вещественная_часть` является необязательной и `мнимая_часть` может указываться без вещественной составляющей. Во внутреннем представлении комплексное число реализовано в виде двух чисел с плавающей точкой, но при обработке чисел этого типа используется математика комплексных чисел.

¹ Это замечание относится к стандартной реализации CPython. В Jython числовые типы Python в действительности соответствуют классам Java.

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` и другие.

Вспомогательные модули

`random`, `math` и другие.

По мере продвижения мы встретимся со всеми этими компонентами.

Наконец, если вам предстоит заниматься сложными математическими вычислениями, вы можете использовать специализированное расширение *NumPy* (Numeric Python), которое обеспечивает расширенные возможности при программировании математических вычислений, такие как матричный тип, обработка векторов и библиотеки для выполнения сложнейших математических вычислений. Группы программирования научных вычислений в таких организациях, как Lawrence Livermore и NASA, используют Python с расширением NumPy для реализации разнообразных задач, которые раньше решались с помощью языков C++, FORTRAN или Matlab.

Из-за высокой сложности NumPy мы больше не будем говорить об этом расширении в данной книге. Дополнительные средства поддержки математических вычислений в языке Python вы найдете на сайте Vaults of Parnassus или выполнив поиск в Сети. Следует также заметить, что в настоящее время расширение NumPy является дополнительным — оно не входит в состав пакета Python и его необходимо устанавливать отдельно.



В Python 3.0 типы целых чисел и длинных целых будут унифицированы. Таким образом, интерпретатор будет поддерживать единственный целочисленный тип — тип `int`. Он будет поддерживать неограниченную точность представления, как это сейчас реализовано в длинных целых числах. В связи с этим большинство программистов заметят лишь незначительные изменения или вообще ничего не заметят. Дополнительную информацию вы найдете в примечаниях к выпуску Python 3.0.

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 нотации и символов операторов. Например,

сложение двух чисел X и Y производится с помощью выражения $X + Y$, которое предписывает интерпретатору Python применить оператор $+$ к значениям с именами X и Y . Результатом выражения $X + Y$ будет другой числовой объект.

В табл. 5.2 приводится перечень всех операторов, имеющихся в языке Python. Многие из них достаточно понятны сами по себе, например обычные математические операторы ($+$, $-$, $*$, $/$ и т. д.). Некоторые будут знакомы тем, кто в прошлом использовал язык C: оператор $\%$ вычисляет остаток от деления, оператор $\ll$ производит побитовый сдвиг влево, оператор $\&$ выполняет побитовую операцию и т. д. Другие операторы более характерны для языка Python, и не все имеют математическую природу. Например, оператор is проверяет идентичность объектов (это более строгая форма проверки на равенство), оператор lambda создает неименованные функции и т. д. О большинстве из этих операторов мы еще будем говорить более подробно.

Таблица 5.2. Операторы выражений в языке Python и правила определения старшинства

Операторы	Описание
<code>yield x</code>	Генератор протокола передачи функции (появился в версии 2.5)
<code>lambda args: expression</code>	Создает анонимную функцию
<code>x if y else z</code>	Трехместный оператор выбора (появился в версии 2.5)
<code>x or y</code>	Логическая операция ИЛИ (значение y вычисляется, только если значение x ложно)
<code>x and y</code>	Логический оператор И (значение y вычисляется, только если значение x истинно)
<code>not x</code>	Логическое отрицание
<code>x < y, x <= y, x > y, x >= y, x == y, x <> y, x != y, x is y, x is not y, x in y, x not in y</code>	Операторы сравнения, операторы равенства ^a , операторы проверки идентичности объектов, операторы проверки вхождения в последовательность
<code>x y</code>	Битовая операция ИЛИ
<code>x ^ y</code>	Битовая операция «исключающее ИЛИ» (XOR)
<code>x & y</code>	Битовая операция И
<code>x << y, x >> y</code>	Сдвиг значения x влево или вправо на y битов
<code>-x + y, x - y</code>	Сложение/конкатенация, вычитание

^a В версии Python 2.5 неравенство значений можно проверить двумя способами, как $X \neq Y$ или как $X <> Y$. В Python 3.0 последний вариант будет убран, как избыточный – используйте выражение $X \neq Y$ для проверки на неравенство.

Операторы	Описание
<code>x * y, x % y, x / y, x // y</code>	Умножение/повторение, остаток/формат, деление ^a
<code>-x, +x, ~x, x ** y</code>	Унарный знак «минус», тождественность, битовое дополнение, возведение в степень
<code>x[i], x[i:j], x.attr, x(...)</code>	Индексация, извлечение среза, уточнение имени, вызов функции
<code>(...), [...], {...}, {...}</code>	Кортеж, список ^b , словарь, преобразование в строку ^c

^a Операция деления с округлением вниз (`X // Y`), впервые появившаяся в Python 2.2, всегда усекает дробную часть. Она будет описана в разделе «Деление: классическое, с округлением вниз и истинное».

^b Начиная с версии Python 2.0, с помощью синтаксиса списка (`[...]`) можно определить либо литерал списка, либо генератор списков. Последняя форма представления подразумевает выполнение цикла и сбор результатов в виде нового списка.

^c Преобразование объектов в соответствующие им строки для вывода на экран также может быть выполнено с помощью более удобочитаемых функций `str` и `repr`, которые будут описаны в разделе «Форматы отображения чисел» ниже, в этой же главе. Из-за неочевидности выражение `'X'` планируется исключить из Python 3.0, используйте вместо него `repr(X)`.

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 выполняться в первую очередь? Ответ на этот вопрос заключается в *старшинстве операторов*. Когда интерпретатор Python встречает выражение, содержащее более одного оператора, он делит его на отдельные части в соответствии с *правилами старшинства* и определяет порядок вычисления этих частей выражения. В табл. 5.2 чем выше приоритет оператора, тем ниже он находится в таблице и тем раньше он выполняется в смешанных выражениях.

Например, если вы запишете выражение `X + Y * Z`, интерпретатор Python сначала 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 группировать части выражений с помощью круглых скобок. Когда часть выражения заключается 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 способов, 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 отсутствовали). Вообще говоря, использование круглых скобок в сложных выражениях 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 числовые типы. Например, вы хотите найти сумму целого числа и числа с плавающей точкой:

```
40 + 3.14
```

Но это влечет за собой другой вопрос: какого типа будет результат – целое число или число с плавающей точкой? 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 преобразование типов операндов к типу самого сложного операнда, а потом применяет 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 образом: целые числа проще, чем длинные целые числа, которые проще чисел с плавающей точкой, которые в свою очередь проще комплексных чисел. Поэтому, когда в выражении участвуют целое число и число с плавающей точкой, как в предыдущем примере, то целое число сначала преобразуется в число с плавающей точкой и затем выполняется операция из математики чисел с плавающей точкой, что дает в результате число с плавающей точкой. Точно так же, когда один из операндов в выражении является комплексным числом, другой

операнд преобразуется в комплексное число, и выражение дает в результате также комплексное число. Как будет показано далее в этом разделе, начиная с версии 2.2, интерпретатор Python также автоматически преобразует целые числа в длинные целые, если их значения оказываются слишком большими, чтобы уместиться в формат простого целого числа.

Существует возможность принудительного преобразования типов с помощью встроенных функций:

```
>>> int(3.1415)
3
>>> float(3)
3.0
>>> long(4)
4L
```

Однако в обычных ситуациях делать это не приходится, потому что Python автоматически 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 чисел и чисел с плавающей точкой) в математических операциях или в операциях сравнения. Вообще, Python не вовлекает в преобразование другие типы. 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.

Обзор: перегрузка операторов

Несмотря на то что сейчас в центре нашего внимания находятся встроенные числа, вы должны знать, что в языке Python существует возможность выполнить (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 выражения `[i]` и т. д.

Кроме того, Python сам автоматически перегружает некоторые операторы, чтобы с их помощью можно было выполнять различные действия, в зависимости от типа обрабатываемых встроенных объектов. Например, оператор `+` выполняет операцию сложения, когда применяется к числам, но когда он применяется к последовательностям, таким как строки или списки, он выполняет операцию конкатенации. В действительности оператор `+` может выполнять любые действия, когда применяется к объектам, которые вы определяете с помощью классов.

Как было показано в предыдущей главе, эта особенность обычно называется *полиморфизмом*; термин означает, что выполняемая операция зависит от типов объектов-операндов, над которыми она выполняется.

Мы еще вернемся к этому понятию в главе 15, когда будем рассматривать функции, потому что в этом контексте суть полиморфизма становится более очевидной.

Числа в действии

Самый лучший способ понять числа и выражения состоит в том, чтобы увидеть их в действии. Давайте запустим 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 основные выражения

Прежде всего мы рассмотрим основные арифметические операции. В следующем упражнении выполняется присваивание целочисленных значений двум *переменным* (a и b), чтобы потом использовать их в более сложных выражениях. Переменные – это всего лишь имена, создаваемые в языке Python, которые используются для обозначения информации, сохраняемой в программах. Об этом мы будем говорить более подробно в следующей главе, а пока вы должны знать, что в языке Python:

- Переменные создаются при выполнении операции присваивания значения.
- При вычислении выражений имена переменных замещаются их значениями.
- Прежде чем переменная сможет участвовать в выражениях, ей должно быть присвоено значение.
- Переменные именуют объекты и никогда не объявляются заранее.

Другими словами, следующие операции присваивания автоматически приводят 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, чтобы объяснять работу программного кода.¹ В следующей части книги мы познакомимся

¹ Если вы прогоняете примеры на практике, вам не нужно вводить комментарии, потому что они просто игнорируются интерпретатором Python и не являются необходимой частью инструкций, которые мы выполняем.

с еще одной похожей особенностью – со строками описания, которые включают текст комментариев в объекты.

А теперь попробуем использовать наши первые целочисленные объекты в выражениях. 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, и при работе в интерактивном режиме результат вычисления выражения тут же выводится на экран:

```
>>> a + 1, a - 1      # Сложение (3 + 1), вычитание (3 - 1)
(4, 2)
>>> b * 3, b / 2      # Умножение (4 * 3), деление (4 / 2)
(12, 2)
>>> 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 выражения, разделенные запятыми. Именно по этой причине результаты отображаются в круглых скобках (подробнее о кортежах будет рассказываться позднее). Следует заметить, что эти выражения выполняются без ошибок потому, что ранее 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 значения. 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 т. д.

Ниже приводятся два немного более сложных выражения, чтобы проиллюстрировать порядок выполнения операторов и производимые преобразования:

```
>>> b / 2 + a          # То же, что и ((4 / 2) + 3)
5
>>> 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 первым. Результат вычисляется так, как если бы выражение включало скобки, как показано в комментарии. Кроме того, обратите внимание на то, что в первом выражении все числа являются целыми, поэтому интерпретатор выполняет целочисленные операции деления и сложения.

Во втором выражении круглые скобки окружают операцию сложения, чтобы вынудить интерпретатор выполнить ее в первую очередь (то есть перед оператором `/`). Кроме того, один из операндов является числом с плавающей точкой, т. к. в нем присутствует десятичная точка: `2.0`. Вследствие такого смещения типов перед выполнением операции сложения Python преобразует целое число, на которое ссылается имя `a`, в число с плавающей точкой (`3.0`). Кроме того, он также преобразует значение переменной `b` в число с плавающей точкой (`4.0`) и выполняет вещественное деление (`4.0 / 5.0`), что в результате также дает число с плавающей точкой `0.8`. Если бы в этом выражении участвовали только целые числа, была бы выполнена операция целочисленного деления (`4 / 5`) и результат был бы усечен до нуля (по крайней мере, в Python 2.5; смотрите обсуждение истинного деления, которое следует далее).

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
0.800000000000000000000000

>>> print b / (2.0 + a)    # Инструкция print отбрасывает лишние цифры
0.8
```

Причина появления такого, немного странного, результата кроется в ограничениях аппаратных средств, реализующих вещественную математику, и в невозможности 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 присутствуют в аппаратной части компьютера, выполняющей операции над числами 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`.

Однако надо заметить, что не всегда значения отображаются с таким большим числом цифр:

```
>>> 1 / 2.0
0.5
```

Форматы представления repr и str

С технической точки зрения различия между функцией автоматического вывода в интерактивной оболочке и инструкцией `print` заключаются в различиях между встроенными функциями `repr` и `str`:

```
>>> repr(num) # Используется для автоматического вывода:
                # в форме как есть
'0.33333333333333331'
>>> str(num)  # Используется инструкцией print: дружественная форма
'0.333333333333'
```

Обе функции преобразуют произвольные объекты в их строковое представление: `repr` (и функция автоматического вывода в интерактивной оболочке) выводит результаты в том виде, в каком они были получены; `str` (и инструкция `print`) обычно выполняет преобразование значения в более дружественное представление. Мы еще будем говорить об этом при обсуждении строк далее в книге, где вы найдете больше информации об этих встроенных функциях.

и что кроме применения инструкции `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 строки форматирования
'3.333333e-001'
>>> "%2.2f" % num      # Вывод с использованием строки форматирования
'0.33'
```

В последних двух случаях была использована *строка форматирования* результата выражения, которая позволяет гибко определять формат представления, но об этом мы поговорим в главе 7, когда займемся исследованием строк.

Деление: классическое, с округлением вниз и истинное

Теперь, когда вы увидели, как работает операция деления, вы должны знать, что она претерпит небольшие изменения в будущей версии Python (версия 3.0, которая должна появиться после выпуска этого издания книги). В Python 2.5 она работает так, как только что было описано, но фактически существуют два оператора деления (один из которых претерпит изменения):

X / Y

Классическое деление. В Python 2.5 и в более ранних версиях результат усекается при делении целых чисел и сохраняется дробная часть для чисел с плавающей точкой. Этот оператор станет оператором *истинного* деления – в будущей версии Python 3.0 он будет всегда сохранять дробную часть, независимо от типов операндов.

$X // Y$

Деление с округлением вниз. Этот оператор впервые появился в Python 2.2. Он всегда отсекает дробную часть, округляя результат до ближайшего наименьшего целого, независимо от типов операндов.

Деление с округлением вниз было добавлено по той причине, что в текущей модели классического деления тип результата зависит от типов операндов, который трудно предсказать заранее в языках программирования с динамической типизацией, таких как Python.

Из-за возможных проблем с обратной совместимостью операция деления в языке Python в настоящее время находится в состоянии постоянного изменения. Подытоживая вышесказанное: в версии 2.5 оператор деления `/` по умолчанию работает, как было описано выше, а оператор деления с округлением вниз `//` может использоваться для округления результата деления до ближайшего наименьшего целого, независимо от типов операндов:

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)
(2, 2.5, -2.5, -3)

>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)
(2, 2.0, -3.0, -3)

>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)
(3, 3.0, 3, 3.0)
```

В будущей версии Python оператор `/` изменится так, что будет возвращать *истинный* результат деления, который всегда будет содержать дробную часть, даже в случае деления целых чисел, например, результатом выражения `1 / 2` будет значение `0.5`, а не `0`, в то время как результатом выражения `1 // 2` будет значение `0`.

А сейчас, пока эти изменения еще не включены полностью, ознакомиться с будущим принципом действия оператора `/` можно при использовании специальной формы импорта: `from __future__ import division`. В этом случае оператор `/` превращается в оператор истинного деления (с сохранением дробной части в результате), а принцип действия оператора `//` остается неизменным. Ниже показано, как эти операторы будут действовать в будущем:

```
>>> from __future__ import division

>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)
(2.5, 2.5, -2.5, -2.5)
```

```
>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)
(2, 2.0, -3.0, -3)

>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)
(3.0, 3.0, 3, 3.0)
```

Рассмотрите простой пример цикла `while` в главе 13 и соответствующее упражнение в конце четвертой части книги, которые иллюстрируют программный код, на котором может сказаться это изменение в поведении оператора `/`. Вообще, воздействию этого изменения может быть подвержен любой программный код, который подразумевает усечение дробной части в целочисленном результате (в этих случаях вместо него следует использовать оператор `//`). К моменту написания этих строк данное изменение планируется включить в версию Python 3.0, но вам обязательно следует опробовать эти выражения в своей версии интерпретатора, чтобы увидеть, как этот оператор действует. Кроме того, запомните специальную команду `from`, использованную здесь, — она еще будет обсуждаться в главе 21.

Битовые операции

Помимо обычных числовых операций (сложение, вычитание и т. д.) язык Python поддерживает большую часть видов числовых выражений, доступных в языке C. Например, ниже приводится пример выполнения операций поразрядного 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 двоичная операция **И** ($0001\&0001 = 0001$). Такого рода операции позволяют хранить сразу несколько флагов и других значений в единственном целом числе.

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 возникает желание использовать битовые операции в программах на языке Python, вам необходимо вспомнить, на каком языке вы программируете. В Python

Комплексные числа

Комплексные числа – это особый базовый тип данных в Python. Если вы знаете о существовании комплексных чисел, вы должны знать, когда они используются. В противном случае этот раздел можно пропустить. Комплексные числа состоят из двух чисел с плавающей точкой, представляющих вещественную и мнимую части, и в тексте программы отличаются наличием суффикса `J`, или `j`, после мнимой части. Если вещественная часть не равна нулю, комплексное число записывается как сумма двух частей, с помощью символа `+`. Например, комплексное число, вещественная часть которого равна 2, а мнимая часть `-3`, записывается как `2 + -3j`. Ниже приводятся примеры некоторых 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`, предназначенная для работы с комплексными числами). Комплексные числа обычно используются в инженерных программах. Поскольку это инструмент повышенной сложности, ищите подробности в справочном руководстве к языку Python.

Шестнадцатеричная и восьмеричная формы записи чисел

Как уже говорилось ранее, целые числа в языке Python могут записываться в шестнадцатеричной (по основанию 16) и восьмеричной (по основанию 8) форме:

- Восьмеричные литералы должны начинаться с символа `0`, вслед за которым следует последовательность восьмеричных цифр `0-7`, каждая из которых представляет три бита.
- Шестнадцатеричные литералы должны начинаться с комбинации символов `0x` или `0X`, вслед за которой следует последовательность шестнадцатеричных цифр `0-9` и `A-F` (в верхнем или в нижнем регистре), каждая из которых представляет четыре бита.

Имейте в виду, что это всего лишь альтернативный синтаксис задания значений целочисленных объектов. Например, следующие восьмеричные и шестнадцатеричные литералы создают обычные целые числа с заданными значениями:

```
>>> 01, 010, 0100    # Восьмеричные литералы
(1, 8, 64)
```

```
>>> 0x01, 0x10, 0xFF # Шестнадцатеричные литералы
(1, 16, 255)
```

Здесь восьмеричное значение 0100 соответствует десятичному значению 64, а шестнадцатеричное 0xFF – десятичному 255. По умолчанию интерпретатор Python выводит числа в десятичной системе счисления (по основанию 10), но предоставляет встроенные функции, которые позволяют преобразовывать целые числа в последовательности восьмеричных и шестнадцатеричных цифр:

```
>>> oct(64), hex(64), hex(255)
('0100', '0x40', '0xff')
```

Функция `oct` выводит целое число в восьмеричном представлении, а функция `hex` – в шестнадцатеричном. Кроме того, существует возможность обратного преобразования – встроенная функция `int`, которая преобразует строку цифр в целое число. Во втором необязательном аргументе она может принимать основание системы счисления:

```
>>> int('0100'), int('0100', 8), int('0x40', 16)
(100, 64, 64)
```

Функция `eval`, с которой мы встретимся далее в этой книге, интерпретирует строку во входном аргументе, как программный код на языке Python. Поэтому она имеет похожий эффект (хотя обычно она работает заметно медленнее, потому что ей приходится компилировать и выполнять строку как часть программы, а это предполагает, что вы должны иметь безграничное доверие к источнику запускаемой строки – достаточно грамотный пользователь мог бы подсунуть вашей программе строку, которая при выполнении в функции `eval` удалит все файлы на вашем компьютере!):

```
>>> eval('100'), eval('0100'), eval('0x40')
(100, 64, 64)
```

Наконец, целые числа могут быть преобразованы в восьмеричное и шестнадцатеричное представления с помощью строки выражения форматирования:

```
>>> "%o %x %X" % (64, 64, 255)
'100 40 FF'
```

Строки форматирования будут рассматриваться в главе 7.

Прежде чем двинуться дальше, считаю своим долгом предупредить вас: будьте осторожны при использовании ведущего нуля в языке Python, если вы не предполагаете использование восьмеричных литералов. Интерпретатор Python будет воспринимать их как числа в восьмеричной системе счисления, что может не соответствовать вашим ожиданиям – число 010 всегда соответствует десятичному числу 8, а не десятичному 10 (независимо от того, что вы имели в виду!).

Другие встроенные средства для работы с числами

В дополнение к этим базовым типам объектов Python предоставляет встроенные функции и модули для работы с числами. Встроенные функции `int` и `round`, например, усекают и округляют числа с плавающей точкой, соответственно. В следующем примере демонстрируется использование модуля `math` (который содержит реализацию множества функций из библиотеки `math` языка C) и нескольких 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)

>>> abs(-42), 2**4, pow(2, 4)
(42, 16, 16)

>>> int(2.567), round(2.567), round(2.567, 2)      # Усечение, округление
(2, 3.0, 2.5699999999999998)
```

Как уже указывалось ранее, если в последнем случае использовать инструкцию `print`, мы получим результат (2, 3.0, 2.57).

Обратите внимание, что встроенные модули, такие как `math`, необходимо импортировать, а встроенные функции, такие как `abs`, доступны всегда, без выполнения операции импорта. Говоря другими словами, модули – это внешние компоненты, а встроенные функции постоянно располагаются в пространстве имен, которое используется интерпретатором Python по умолчанию для поиска имен, используемых программой. Это пространство имен соответствует модулю с именем `__builtin__`. В четвертой части книги мы подробнее поговорим о разрешении имен, а пока всякий раз, когда слышите слово «модуль», думайте: «импорт».

Модуль `random` из стандартной библиотеки также необходимо импортировать. Этот модуль предоставляет возможность получения случайных чисел с плавающей точкой в диапазоне от 0 до 1, случайных целых чисел в заданном диапазоне, случайного выбора элементов последовательности и многое другое:

```
>>> import random
>>> random.random()
0.49741978338014803
>>> random.random()
0.49354866439625611

>>> 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 слайдов, в программах статистического моделирования и т. д. За дополнительной информацией обращайтесь к руководству по стандартной библиотеке языка Python.

Другие числовые типы

В этой главе мы рассматривали базовые числовые типы языка Python – целые числа, длинные целые, числа с плавающей точкой и комплексные числа. Их вполне достаточно для решения математических задач, с которыми придется столкнуться большинству программистов. Однако язык Python содержит несколько более экзотических числовых типов, которые заслуживают того, чтобы коротко познакомиться с ними.

Числа с фиксированной точностью

В версии Python 2.4 появился новый базовый числовой тип: числа с фиксированной точностью представления. Синтаксически такие числа создаются вызовом функции из импортируемого модуля и не имеют литерального представления. Функционально числа с фиксированной точностью напоминают числа с плавающей точкой, с фиксированным числом знаков после запятой, отсюда и название «числа с фиксированной точностью». Например, с помощью таких чисел можно хранить значение, которое всегда будет иметь два знака после запятой. Кроме того, можно указать, как должны обрабатываться лишние десятичные цифры – усекаться или округляться. И хотя скорость работы с такими числами несколько ниже, чем с обычными числами с плавающей точкой, 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.

Точность при работе с обычными вещественными числами несколько ниже. Например, результат следующего выражения должен быть равен нулю, но из-за недостаточного числа битов в представлении вещественных чисел страдает точность вычислений:

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.5511151231257827e-017
```

Попытка вывести результат в более дружелюбной форме мало помогает, потому что проблема связана с ограниченной точностью представления вещественных чисел:

```
>>> print 0.1 + 0.1 + 0.1 - 0.3
5.55111512313e-017
```

Однако при использовании чисел с фиксированной точностью результат получается 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 представления создаются вызовом функции конструктора `Decimal` из модуля `decimal`, которому передается строка, содержащая желаемое число знаков после запятой. Когда в выражении участвуют числа с различной точностью представления, Python автоматически выбирает наибольшую точность для представления результата:

```
>>> Decimal('0.1') + Decimal('0.10') + Decimal('0.10') - Decimal('0.30')
Decimal('0.00')
```

Другие инструменты модуля `decimal` позволяют задать точность представления всех таких чисел и многое другое. Например, объект `context` из этого модуля позволяет задавать точность (число знаков после запятой) и режим округления (вниз, вверх и т. д.):

```
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1428571428571428571428571429')

>>> decimal.getcontext().prec = 4
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
```

Поскольку тип чисел с фиксированной точностью достаточно редко используется на практике, за дополнительными подробностями я отсылаю вас к руководствам по стандартной библиотеке Python.

Множества

В версии Python 2.4 также появился новый тип коллекций – *множества*. Поскольку этот тип является коллекцией других объектов, возможно, его обсуждение выходит за рамки этой главы. Но так как он поддерживает набор математических операций, мы познакомимся с основами его использования. Чтобы создать объект-множество, нужно передать последовательность или другой объект, поддерживающий возможность итераций по его содержимому, встроенной функции `set` (похожая функция существовала в Python до версии 2.4 во внешнем модуле, но начиная с версии Python 2.4 эта функция стала встроенной):

```
>>> x = set('abcde')
>>> y = set('bdxyz')
```

Обратно функция возвращает объект множества, который содержит все элементы объекта, переданного функции (примечательно, что множеств

ва не предусматривают возможность определения позиций элементов, а кроме того, они не являются последовательностями):

```
>>> x
set(['a', 'c', 'b', 'e', 'd'])
```

Множества, созданные таким способом, поддерживают обычные математические 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         # Разница 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'])
```

Такие операции удобны при работе с большими множествами данных – пересечение двух множеств содержит объекты, которые присутствуют в обоих множествах, а объединение содержит все объекты из обоих множеств. Ниже приводится более реальный пример применения операций над множествами – при работе со списками служащих гипотетической компании:

```
>>> engineers = set(['bob', 'sue', 'ann', 'vic'])
>>> managers = set(['tom', 'sue'])
>>>
>>> engineers & managers # Кто является и инженером, и менеджером одновременно?
set(['sue'])
>>>
>>> engineers | managers # Все, кто принадлежит к той или иной категории
set(['vic', 'sue', 'tom', 'bob', 'ann'])
>>>
>>> engineers - managers # Инженеры, которые не являются менеджерами
set(['vic', 'bob', 'ann'])
```

Кроме того, объект множества предоставляет метод, который реализует более экзотический набор операций. Хотя операции над множествами в языке Python можно реализовать вручную (часто так и делалось в прошлом), встроенные операции над множествами обеспечивают высокую скорость выполнения стандартных операций, опираясь на эффективные алгоритмы и приемы реализации.

За дополнительной информацией по множествам обращайтесь к руководству по стандартной библиотеке языка Python.



В версии Python 3.0 предлагается форма записи литералов `{1, 2, 3}`, которая дает тот же эффект, который имеет вызов функции `set([1, 2, 3])` и тем самым предоставляет еще один способ создания объекта множества. Однако это – будущее расширение языка, поэтому за более подробной информацией обращайтесь к примечаниям к выпуску 3.0.

Логические значения

Иногда утверждается, что логический тип `bool` в языке Python по своей природе 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.

В версии Python 2.3 появился новый логический тип с именем `bool`, доступный в виде двух новых встроенных предопределенных имен `True` и `False`, которые в свою очередь являются всего лишь подклассом (в объектно-ориентированном смысле) встроенного целочисленного 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, класс `bool` переопределяет функции `str` и `repr`).

В соответствии с интерпретацией этих значений, начиная с версии Python 2.3, значения выражений логического типа выводятся 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 становится инициализация флагов, например `flag = False`.

Во всех остальных практических применениях значения `True` и `False` можно интерпретировать, как предопределенные переменные с целочисленными значениями `1` и `0`. В любом случае раньше большинство программистов создавали переменные `True` и `False`, которым присваивали значения `1` и `0`, таким образом данный новый тип 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 + 3` даст в языке Python результат `4`!

Мы еще вернемся к логическому типу в главе 9 (где будет дано определение истины в языке Python) и в главе 12 (где познакомимся с такими логическими операторами, как `and` и `or`).

Расширения сторонних разработчиков

Помимо встроенных числовых типов собственно языка Python вы можете встретить различные свободно распространяемые расширения сторонних разработчиков, реализующие еще более экзотические числовые типы. К таким типам можно отнести рациональные числа, векторы и матрицы. Модули, реализующие эти типы, можно загрузить из Сети, там же вы найдете более подробную информацию о них.

В заключение

В этой главе были рассмотрены типы числовых объектов языка Python и операции, применяемые к ним. Здесь мы познакомились со стандартными целочисленными типами и с числами с плавающей точкой, а также с некоторыми экзотическими, нечасто используемыми типами, такими как комплексные числа и числа с фиксированной точностью. Мы также исследовали синтаксис выражений в языке Python, порядок преобразования типов, битовые операции и различные литературные формы представления чисел в сценариях.

Далее, в этой же части книги, мы рассмотрим следующий тип объектов – строки. Однако в следующей главе мы потратим некоторое время на более подробное ознакомление с механизмом операции присваивания значений переменным. Это, пожалуй, самая фундаментальная идея в языке Python, поэтому вам обязательно 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. Каким образом можно выполнить усечение и округление числа с плавающей точкой?
7. Каким образом можно преобразовать целое число в число с плавающей точкой?
8. Каким образом можно вывести целое число в восьмеричном и шестнадцатеричном представлениях?
9. Каким образом можно преобразовать строковое представление восьмеричного или шестнадцатеричного 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 результат выражения $6 + 4$. При отсутствии круглых скобок в языке Python используются правила старшинства операторов, согласно которым оператор умножения имеет более 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 *pi*, тангенса и многие другие доступны в импортируемом модуле `math`. Чтобы найти квадратный корень числа, необходимо импортировать модуль `math` и вызвать функцию `math.sqrt(N)`. Чтобы найти квадрат числа, можно воспользоваться либо оператором возведения в степень $X ** 2$, либо встроенной функцией `pow(X, 2)`.
5. Результатом будет число с плавающей точкой: целые числа будут преобразованы в числа с плавающей точкой, как к наиболее сложному числовому типу в выражении, и для вычисления результата будет использоваться математика вещественных чисел.
6. Функция `int(N)` выполняет отсечение дробной части, а функция `round(N, digits?)` выполняет округление.
7. Функция `float(N)` выполняет преобразование целых чисел в числа с плавающей точкой. Смешивание целых чисел и чисел с плавающей точкой в одном выражении также приводит к выполнению преобразования.
8. Встроенные функции `oct(I)` и `hex(I)` возвращают строковые представления целых чисел в восьмеричном и в шестнадцатеричном форматах. Для этих же целей может использоваться строка формата `%`.
9. Для преобразования строковых представлений восьмеричных и шестнадцатеричных чисел в обычные целые числа может использоваться функция `int(S, base?)`. Для этих целей может также использоваться функция `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 вывода которых просто используются разные форматы представления.

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 исследования 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 действительности нам даже не приходится беспокоиться о конкретных типах; более того, они могут применяться в более широком диапазоне ситуаций, чем можно было предусмотреть заранее. Поскольку динамическая типизация составляет основу этой гибкости, давайте коротко ознакомимся с этой моделью.

Отсутствие инструкций объявления

Если у вас имеется опыт работы с компилирующими языками или с языками, обладающими статической типизацией, такими как C, C++ или Java, вероятно, эта тема в книге вызывает у вас недоумение. Мы все время использовали переменные, не объявляя ни их самих, ни их типы, и все как-то работало. Например, когда вводится инструкция `a = 3` в интерактивной оболочке интерпретатора 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 действия модели *динамической типизации*, используемой в языке Python. В языке Python типы 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).

Переменные, объекты и ссылки

Как было видно из примеров, приводившихся до сих пор в книге, когда запускается 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 решается весьма естественным способом, как описано ниже:

Создание переменной

Переменная (то есть имя), такая как `a`, создается автоматически, когда в программном коде ей впервые присваивается некоторое значение. Все последующие операции присваивания просто изменяют значение, ассоциированное с уже созданным именем. Строго говоря, интерпретатор Python определяет некоторые имена до запуска программного кода, но вполне допустимо думать, что переменные создаются первой операцией присваивания.

Типы переменных

Переменные не имеют никакой информации о типе или ограничениях, связанных с ним. Понятие типа присуще объектам, а не именам. Переменные универсальны по своей природе – они всегда являются всего лишь ссылками 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.

Эта модель существенно отличается от традиционных языков программирования. Динамическая типизация обычно проще поддается пониманию начинающих, особенно если они четко осознают разницу между именами и объектами. Например, если ввести такую инструкцию:

```
>>> 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. Переменная связывается с новым объектом, представляющим число 3.

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 показана 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).

Эти ссылки на объекты в языке Python так и называются *ссылки*, то есть ссылки – это своего рода ассоциативная связь, реализованная в виде указателя на область памяти.¹ Когда бы ни использовалась переменная (то есть ссылка), интерпретатор Python автоматически переходит по ссылке от переменной на объект. 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 выполнения выражения создается новое значение, интерпретатор Python создает новый объект (то есть выделяет область памяти),

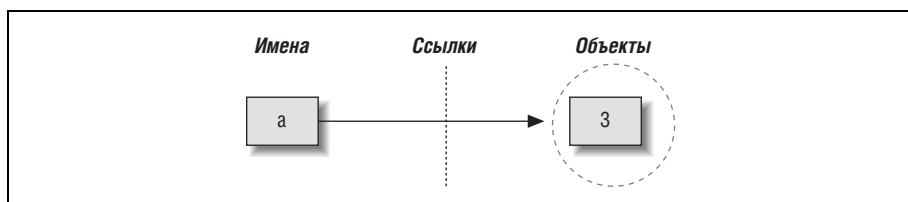


Рис. 6.1. Имена и переменные после выполнения операции присваивания $a = 3$. Переменная превращается в ссылку на объект 3. Во внутреннем представлении переменная в действительности является указателем на пространство памяти с объектом, созданным в результате интерпретации литерального выражения 3

¹ Читатели, знакомые с языком C, сочтут, что между ссылками в языке Python и указателями в языке C много общего (адреса в памяти). И действительно, ссылки реализованы как указатели и они часто играют ту же роль, особенно когда объект относится 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 огромный пласт ошибок, которые можно допустить в языке C. Вы можете считать ссылки указателями типа «void» в языке C, которые автоматически разыменовываются при любом использовании.

представляющий это значение. Внутренняя реализация интерпретатора, с целью оптимизации, кэширует и повторно использует некоторые типы неизменяемых объектов, такие как малые целые числа и строки (каждый 0 в действительности не является новой областью в памяти, но подробнее о механизме кэширования мы поговорим позже). 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: *указатель типа*, используемый для хранения информации о типе объекта, и *счетчик ссылок*, используемый для определения момента, когда память, занимаемая объектом, может быть освобождена. 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 число с плавающей точкой
```

Это не совсем типичный программный код на языке Python, но он работает – сначала создается целое число, потом строка и, наконец, число с плавающей точкой. 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 строковый.

Однако в действительности этого не происходит. В языке Python все реализовано 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 конкретные моменты времени.

С другой стороны, объекты знают, к какому типу они относятся, – каждый объект содержит поле, в котором хранится информация о его типе. Целочисленный 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). Указатель типа для строки `'spam'` указывает на строковый тип (с именем `str`). Поскольку информация о типе хранится в объектах, ее не нужно хранить в переменных.

Итак, типы в языке Python – это свойства объектов, а не переменных. В типичном программном коде переменная обычно ссылается на объекты только одного типа. Так как для языка Python это не является обязательным требованием, программный код на языке Python обладает более высокой гибкостью, чем вам может показаться, – при умелом использовании особенностей языка программный код смог бы работать со многими типами автоматически.

Я уже упоминал, что каждый объект имеет два поля – указатель типа и счетчик 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 переменной объекты различных типов. Но что происходит с прежним значением, когда выполняется новое присваивание? Например, что произойдет с объектом 3 после выполнения следующих инструкций:

```
>>> a = 3
>>> a = 'spam'
```

Дело в том, что всякий раз, когда имя ассоциируется с новым объектом, интерпретатор Python освобождает память, занимаемую предыдущим объектом (если на него не ссылается какое-либо другое имя или объект). Такое автоматическое 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 операции 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 ощущение изменения типа переменной `x` с течением времени. Напомним, 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 Python 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 повторного использования в будущем.

Достигается это за счет того, что в каждом объекте имеется счетчик ссылок, с помощью которого интерпретатор следит за количеством ссылок, указывающих на объект в настоящий момент времени. Как только (и именно в этот момент) значение счетчика достигает нуля, память, занимаемая объектом, автоматически освобождается. В предыдущем листинге мы исходили из предположения, что всякий раз, когда имя x ассоциируется с новым объектом, счетчик предыдущего объекта уменьшается до нуля, заставляя интерпретатор освобождать память.

Основная выгода от сборки мусора состоит в том, что программист может свободно распоряжаться объектами, не будучи обязан освобождать память в своем сценарии. Интерпретатор сам будет выполнять очистку неиспользуемой памяти в ходе выполнения программы. На практике эта особенность позволяет существенно уменьшить объем программного кода по сравнению с низкоуровневыми языками программирования, такими как C и C++.

Разделяемые ссылки

До сих пор мы рассматривали вариант, когда ссылка на объект присваивается 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 взаимоотношений, отраженная на рис. 6.2. Вторая инструкция вынуждает интерпретатор создать переменную b и использовать для инициализации переменную a , при этом она замещается объектом, на который

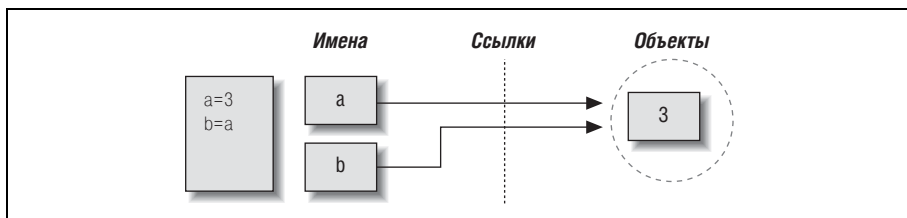


Рис. 6.2. Имена и объекты после выполнения инструкции присваивания $b = a$. Переменная b превращается в ссылку на объект 3. С технической точки зрения переменная в действительности является указателем на область памяти объекта, созданного в результате выполнения литерального выражения 3

ссылается (3), и *b* превращается 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.

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 оказывают влияния на переменную *b* – она по-прежнему ссылается на первый объект, целое число 3. В результате схема ссылок приобретает вид, показанный на рис. 6.3.

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. Например, рассмотрим следующие три инструкции:

```
>>> 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 этого он создает переменную *b* и записывает в нее ту же ссылку, что хранится в переменной *a*, как показано на рис. 6.2. Наконец, последняя инструкция создает совершенно новый объект (в данном

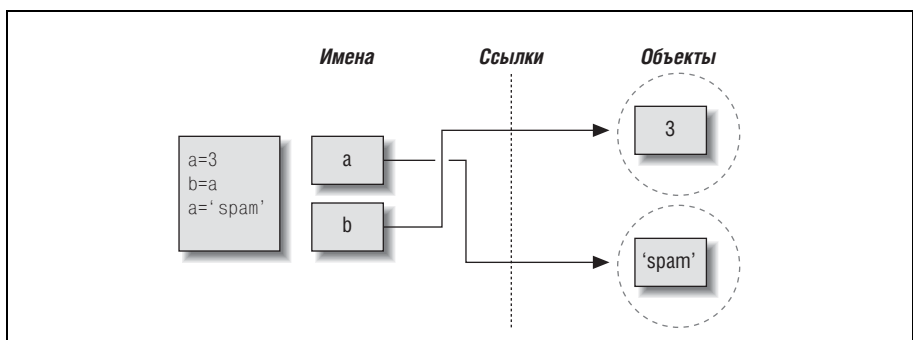


Рис. 6.3. Имена и объекты после выполнения инструкции присваивания *a = 'spam'*. Переменная *a* ссылается на новый объект (то есть на область памяти), созданный в результате выполнения литерального выражения 'spam', но переменная *b* по-прежнему ссылается на первый объект 3. Так как эта операция присваивания никак не изменяет объект 3, она изменяет только переменную *a*, но не *b*

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 неизменяемых 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, доступных 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 когда 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, операция присваивания значения элементу списка фактически изменяет сам список вместо того, чтобы создавать совершенно новый объект списка. При работе 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 коллекция объектов, которая в программном коде записывается как литерал в квадратных скобках:

```
>>> L1 = [2, 3, 4]
>>> L2 = L1
```

В данном случае `L1` – это список, содержащий объекты 2, 3 и 4. Доступ к элементам списка осуществляется по их индексам; так, `L1[0]` ссылается на объект 2, первый элемент в списке `L1`. Безусловно, списки являются полноценными 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 ссылаться на первоначальный список. Однако если синтаксис последней инструкции чуть-чуть изменить, эффект получится радикально другим:

```
>>> 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 также изменилась и переменная L2!
[24, 3, 4]
```

Здесь мы не изменяем сам объект `L1`, изменяется компонент *объекта*, на который 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 сказываться 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 изменилось.

Как правило, это именно то, что требовалось, но вы должны понимать, как это происходит. Это – поведение по умолчанию: если вас оно не устраивает, можно 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 главах 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: способ, основанный на получении среза, неприменим в случае с другим изменяемым базовым типом – со словарями, потому что словари не являются последовательностями. Чтобы скопировать словарь, необходимо воспользоваться методом `D.copy()`. Следует также отметить, что модуль `copy` из стандартной библиотеки имеет в своем составе универсальную функцию, 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 изменяемые объекты), всегда подвержены описанным эффектам. В число таких объектов в языке Python попадают списки, словари и некоторые объекты, объявленные с помощью инструкции `class`. Если такое поведение является нежелательным, 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 принципиальным понятием, чем литералы для объектов некоторых типов. Рассмотрим следующие 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 всего не будет уничтожен. Он, скорее всего, останется в системной таблице для повторного использования, когда вы вновь сгенерируете число 42 в программном коде. Однако большинство объектов уничтожаются немедленно, как только будет потеряна последняя ссылка, особенно те, к которым применение механизма кэширования не имеет смысла.

Например, согласно модели ссылок в языке Python, существует два разных способа выполнить проверку равенства. Давайте создадим разделяемую ссылку 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 практически всегда используется именно этот способ. Второй способ, основанный на использовании оператора `is`, проверяет 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 ссылок в программном коде. Он возвращает значение `False`, даже если имена ссылаются 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 малыми целыми числами:

```
>>> 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 значение), но не эквивалентны (*is*, один и тот же объект), потому что было выполнено 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 значение 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 самой IDLE, а не мною):

```
>>> import sys
>>> sys.getrefcount(1)      # 837 указателей на этот участок памяти
837
```

Такое кэширование объектов и повторное их использование будет малоприменимо к вашему программному коду (если вы не используете оператор *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, объектами, 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 возвращается измененным, значит, вы стали свидетелем того, о чем рассказывалось 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 ссылках, 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 импортировании модулей и т. д. 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, которая была введена в главе 4 и к которой мы еще вернемся далее в этой книге – в языке Python. Поскольку мы не ограничены применением типов в программном коде на языке Python, он обладает очень высокой гибкостью. Как будет показано далее, при правильном использовании динамической типизации и полиморфизма можно создавать такой программный код, который автоматически будет адаптироваться под новые требования в ходе развития вашей системы.

В заключение

В этой главе мы подробно рассмотрели модель динамической типизации в языке Python, то есть способ, который используется интерпретатором для автоматического хранения информации о типах объектов и избавляет нас от необходимости вставлять код объявлений в наши сценарии. Попутно мы узнали, как реализована связь между переменными и объектами. Кроме того, мы исследовали понятие сборки мусора, узнали, как разделяемые ссылки на объекты могут 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 используется повсюду в этом языке, очень важно, чтобы вы разобрались в ней, прежде чем двигаться дальше. Контрольные вопросы к этой главе должны помочь 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 переменной `A`?

```
A = "spam"
B = A
B = "shrubbery"
```

2. Взгляните на следующие три инструкции. Изменится ли значение переменной 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 переменной B присваивается строка 'shrubbery', она просто начинает указывать на другой строковый объект. Первоначально переменные A и B разделяют (то есть ссылаются или указывают) один и тот же строковый объект со значением 'spam', но в языке Python два имени никак не связаны между собой. Таким образом, назначение переменной B ссылки на другой объект не оказывает 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 объекты (или числа, или кортежи), потому что они относятся к категории неизменяемых 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 строка 'spam'. Изменение самого объекта с помощью ссылки B не оказывает влияние на переменную A по той причине, что выражение получения среза создает копию объекта списка, прежде чем присвоить ссылку переменной B. После второго оператора присваивания в программе будет существовать два разных объекта списков 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 переменная A, остается неизменным.

7

Строки

Следующий основной тип на пути нашего исследования встроенных объектов языка Python – это *строки*, упорядоченные последовательности символов, используемые для хранения и представления текстовой информации. Мы коротко 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 представления 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 т. д.

Возможно, вам уже приходилось использовать строки в других языках программирования. 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 обладают мощным набором средств для их обработки. Кроме того, в отличие от таких языков, как C, в языке Python отсутствует специальный тип для представления единственного символа (как тип `char` в языке C), поэтому в случае необходимости используются односимвольные строки.

Строго говоря, строки в языке Python относятся к категории неизменяемых последовательностей, в том смысле, что символы, которые они содержат, имеют определенный порядок следования слева направо и что сами строки невозможно изменить. Фактически, строки – это первый представитель большого класса объектов, называемых последовательностями, которые мы будем здесь изучать. Обратите особое внимание на операции над последовательностями, представленные в этой главе, так как они похожим образом работают и с другими типами

последовательностей, такими как списки и кортежи, которые мы будем рассматривать позже.

В табл. 7.1 приводятся наиболее типичные литералы строк и операций, которые обсуждаются в этой главе. Пустые строки записываются как пара кавычек (или апострофов), между которыми ничего нет; и существуют различные способы записи строк в программном коде. Для работы со строками поддерживаются операции *выражений*, такие как конкатенация (объединение строк), выделение подстроки, выборка символов по индексам (по смещению от начала строки) и т. д. Помимо выражений язык Python предоставляет ряд строковых *методов*, которые реализуют обычные задачи работы со строками, а также *модули* для решения более сложных задач обработки текста, таких как поиск по шаблону. Все эти инструменты будут рассматриваться далее в этой же главе.

Таблица 7.1. Наиболее типичные литералы строк и операции

Операция	Интерпретация
s1 = ''	Пустая строка
s2 = "spam's"	Строка в кавычках
block = """..."""	Блоки в тройных кавычках
s3 = r'\temp\spam'	Неформатированные строки
s4 = u'spam'	Строки с символами Юникода
s1 + s2	Конкатенация, повторение
s2 * 3	
s2[i]	Обращение к символу по индексу, извлечение подстроки (среза), длина
s2[i:j]	
len(s2)	
"a %s parrot" % type	Форматирование строки
s2.find('pa')	Методы строк: поиск, удаление пробелов, поиск с заменой, разбиение на подстроки по символу-разделителю, проверка содержимого, преобразование регистра символов и т. д.
s2.rstrip()	
s2.replace('pa', 'xx')	
s1.split(',')	
s1.isdigit()	
s1.lower()	
for x in s2	
'spam' in s2	
	Обход в цикле, проверка на вхождение

Помимо базового набора инструментов для работы со строками, Python также поддерживает более сложные приемы обработки строк, основанные на применении шаблонов, с помощью библиотечного модуля *re* (*regular expression* – регулярные выражения), о котором говорилось в главе 4. Данная глава начинается с краткого обзора форм строковых литералов и базовых строковых выражений, затем мы перейдем к более сложным инструментам, таким как методы строк и форматирование.

Литералы строк

Вообще говоря, работать со строками в языке Python достаточно удобно. Самое сложное, пожалуй, – это наличие множества способов записи строк в программном коде:

- Строки в апострофах: `'spam'`
- Строки в кавычках: `"spam"`
- Строки в тройных кавычках: `'''... spam ...'''`, `"""... spam ..."""`
- Экранированные последовательности: `"s\tp\na\0m"`
- Неформатированные строки: `r"C:\new\test.spm"`
- Строки символов Юникода: `u'eggs\u0020spam'`

Варианты представления строк в апострофах и кавычках являются наиболее типичными, остальные играют особую роль. 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 взаимозаменяемыми. То есть строковые литералы можно заключать как в апострофы, так и в кавычки – эти две формы представления строк ничем не отличаются 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 обратного слепа. Вы можете вставлять апострофы в строки, заключенные в кавычки, и наоборот:

```
>>> 'knight"s', "knight's"  
( 'knight"s', "knight's" )
```

Между прочим, Python автоматически объединяет последовательно строковых литералов внутри выражения, хотя нет ничего сложного в том, чтобы добавить 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 получен кортеж, а не строка. Кроме того, заметьте, что во всех этих примерах интерпретатор предпочитает выводить строки в апострофах, если их нет внутри строки. Кавычки и апострофы можно вставлять в строки, экранируя их символом обратного слепа:

```
>>> 'knight\'s', "knight's"  
('knight's', 'knight's')
```

Чтобы понять, зачем, вам необходимо узнать, как работает механизм экранирования вообще.

Экранированные последовательности представляют служебные символы

В последнем примере вставляемые в строку кавычка и апостроф превращаются символом обратного слеша. Это частный случай более общей формы: символы обратного 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 символов (обычно ASCII-код 10). Аналогично последовательность `\t` замещается символом 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 последовательности, 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 число байтов в строке независимо от того, как строка отображается на экране:

```
>>> len(s)  
5
```

Длина этой строки составляет пять байтов: она содержит байт ASCII-символа *a*, байт символа новой строки, байт ASCII-символа *b* и т. д. Обратите внимание, что символы обратного слеша не сохраняются в памяти строкового объекта. Для представления служебных символов язык Python обеспечивает полный набор экранированных последовательностей, перечисленных в табл. 7.2.

Таблица 7.2. Экранированные последовательности

Последовательность	Назначение
<code>\newline</code>	Игнорируется (продолжение на новой строке)
<code>\\</code>	Сам символ обратного следа (остается один символ <code>\</code>)
<code>\'</code>	Апостроф (остается один символ <code>'</code>)
<code>\"</code>	Кавычка (остается один символ <code>"</code>)
<code>\a</code>	Звонок
<code>\b</code>	Забой
<code>\f</code>	Перевод формата
<code>\n</code>	Новая строка (перевод строки)
<code>\r</code>	Возврат каретки
<code>\t</code>	Горизонтальная табуляция
<code>\v</code>	Вертикальная табуляция
<code>\N{id}</code>	Идентификатор ID базы данных Юникода
<code>\uhhhh</code>	16-битовый символ Юникода в шестнадцатеричном представлении
<code>\Uhhhh...</code>	32-битовый символ Юникода в шестнадцатеричном представлении ^a
<code>\xhh</code>	Шестнадцатеричное значение
<code>\ooo</code>	Восьмеричное значение
<code>\0</code>	Символ Null (не признак конца строки)
<code>\другое</code>	Не является экранированной последовательностью (символ обратного следа сохраняется)

^a Экранированная последовательность `\Uhhhh...` состоит ровно из восьми шестнадцатеричных цифр (h). Последовательности `\u` и `\U` могут использоваться только в литералах строк символов Юникода.

Некоторые экранированные последовательности позволяют указывать абсолютные двоичные значения в байтах строк. Например, ниже приводится пример строки из пяти символов, содержащей два нулевых байта:

```
>>> 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 (записано в восьмеричной форме), за которыми следует двоичное значение 3 (записано в шестнадцатеричной форме):

```
>>> s = '\001\002\x03'
>>> s
'\x01\x02\x03'
>>> len(s)
3
```

Это обстоятельство приобретает особую важность, когда возникает необходимость обрабатывать на языке Python файлы с двоичными данными. Поскольку 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

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 последней строкой таблицы поведение.² Чтобы явно добавить символ обратного слеша в строку, нужно указать два символа обратного слеша, идущие подряд (\\ – экранированный вариант представления символа \), или использовать неформатированные 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 поро-

¹ Если вам требуется работать с файлами, содержащими двоичные данные, главное отличие в работе с ними заключается в том, что открывать их нужно в режиме двоичного доступа (добавляя к флагу режима открытия флаг b, например "rb", "wb" и т. д.). Кроме того, обратите внимание на модуль struct, который будет описан в главе 9, с помощью которого можно выполнять интерпретацию двоичных данных, загруженных из файла.

² Мне доводилось встречать людей, которые помнили всю таблицу или большую ее часть. Я бы счел их ненормальными, но пришлось бы себя включить в их число.

ждать 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 символ 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 интерпретатор Python будет воспринимать символы обратного слеша в строке как обычные символы. Таким образом, чтобы ликвидировать проблему, связанную с именами файлов в Windows, не забывайте добавлять символ *r*.

```
myfile = open(r'C:\new\text.dat', 'w')
```

Как вариант, учитывая, что два идущих подряд символа обратного слеша интерпретируются как один символ, можно просто продублировать символы обратного слеша:

```
myfile = open('C:\\new\\text.dat', 'w')
```

Сам интерпретатор Python в определенных случаях использует удвоение обратного слеша при выводе строк, содержащих обратный слеш:

```
>>> path = r'C:\new\text.dat'
>>> path          # Показать, как интерпретатор представляет эту строку
'C:\new\text.dat'
>>> print path    # Более дружелюбный формат представления
C:\new\text.dat
>>> len(path)     # Длина строки
15
```

Так же как и в случае с числами, при выводе результатов в интерактивной оболочке по умолчанию используется такой формат представления, как если бы это был программный код, отсюда и экранирование символов обратного слеша. Чтобы проверить, что дело обстоит именно так, можно проверить результат с помощью встроенной функции `len`, которая возвращает число байтов в строке независимо от формата отображения. Если посчитать символы в выводе инструкции `print path`, можно увидеть, что каждому символу обратного слеша соответствует один байт, а всего строка содержит 15 символов.

Помимо хранения имен каталогов в Windows 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 Windows и UNIX можно использовать *простые* символы слеша, потому что Python старается поддерживать переносимость для путей к файлам. И все же, когда для кодирования имен каталогов в Windows используется традиционная нотация с обратными слешами, удобно использовать неформатированные строки.

Тройные кавычки, многострочные блоки текста

К настоящему моменту мы познакомились с кавычками, апострофами, экранированными последовательностями и неформатированными строками. Кроме этого в арсенале языка Python имеется формат представления строковых литералов, в котором используются тройные кавычки. Этот формат иногда называют *блочной строкой*, которую удобно использовать для определения многострочных блоков текста в программном коде. Литералы в этой форме начинаются с трех идущих подряд кавычек (или апострофов), за которыми может 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 завершенной, 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 (в некоторых системах строка приглашения к вводу изменяется на `...`, когда ввод продолжается на следующей линии; среда IDLE просто переводит курсор на следующую линию). Интерпретатор 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.

Строки в тройных кавычках удобно использовать, когда в программе требуется ввести многострочный текст, например, чтобы определить многострочный текст сообщения об ошибке или код разметки на языке HTML или XML. Вы можете встраивать такие блоки текста прямо в свои сценарии, не используя для этого внешние текстовые файлы или явную операцию конкатенации и символы новой строки.

Часто строки в тройных кавычках используются для создания строк документирования, которые являются литералами строк, воспринимаемыми как комментарии при появлении их в определенных местах

сценария (подробнее о них будет рассказываться позже в этой книге). Комментарии не обязательно (но часто!) представляют собой многострочный текст, и данный формат дает возможность вводить многострочные комментарии.

Наконец, иногда тройные кавычки являются ужасающим, хакерским способом временного отключения строк программного кода во время разработки (Хорошо, хорошо! На самом деле это не так ужасно, и в действительности довольно 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
```

Я назвал этот прием ужасным, потому что при работе интерпретатор вынужден создавать строку из таких линий программного кода, 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 поддерживает возможность редактирования исходных текстов на языке Python. При программировании на этом языке практичность часто берет верх над эстетичностью.

Строки символов Юникода позволяют использовать крупные наборы символов

Последний способ создания строк в сценариях является, пожалуй, самым специфичным, и он редко используется за пределами задач обработки кода разметки XML и веб-программирования. *Строки символов Юникода* иногда называют строками «широких» символов. Так как символы Юникода могут быть представлены более чем одним байтом, такие строки могут обеспечить представление более широкого набора символов, чем стандартные строки.

Строки символов Юникода обычно используются с целью поддержки *интернационализации* приложений (иногда называется как «i18n», чтобы сжать 18 символов, расположенные между первым и последним символом в слове «internationalization»). Например, они позволяют программистам непосредственно использовать в сценариях символы из европейских и азиатских алфавитов. Поскольку такие наборы могут содержать больше символов, чем можно представить с помощью однобайтовой кодировки, то для обработки таких видов текста обычно используется Юникод.

В языке Python строки символов Юникода можно записывать, добавляя символ *U* (в верхнем или нижнем регистре) перед открывающей кавычкой:

```
>>> u'spam'
u'spam'
```

Формально с помощью этого синтаксиса создается объект строки символов Юникода, который является самостоятельным типом данных, отличным от обычного строкового типа. Однако Python позволяет легко смешивать в выражениях строки символов Юникода и обычные строки, преобразуя последние до строк символов Юникода при вычислении результата (подробнее об операции конкатенации рассказывает в следующем разделе):

```
>>> 'ni' + u'spam'      # Смешивание строк различных типов
u'nispam'
```

На практике строки символов Юникода поддерживают все обычные операции над строками, которые встретятся вам в следующем разделе, поэтому для вашего программного кода различия в типах играют несущественную роль. Подобно обычным строкам, строки символов Юникода могут объединяться, из них можно извлекать отдельные символы и подстроки, производить поиск по шаблону с помощью модуля `re` и т. д., и эти строки также являются неизменяемыми.

Даже если у вас возникнет необходимость явно выполнить преобразование между этими двумя типами, вы можете использовать встроенную функцию `unicode`:

```
>>> str(u'spam')        # Из Юникода в обычную строку
'spam'
>>> unicode('spam')     # Из обычной строки в Юникод
u'spam'
```

Поскольку кодировка Юникод предназначена для работы с многобайтовыми символами, вы также можете использовать экранированные последовательности `\u` и `\U` для представления символов Юникода в виде числовых значений, размер которых превышает 8 битов:

```
>>> u'ab\x20cd'         # 8-битовые/1-байтовые символы
u'ab cd'
>>> u'ab\u0020cd'       # 2-байтовые символы
u'ab cd'
>>> u'ab\U00000020cd'    # 4-байтовые символы
u'ab cd'
```

Первая из этих инструкций встраивает числовой код символа пробела, который в шестнадцатеричной записи имеет вид `x20`. Вторая и третья инструкции делают то же самое, но используют 2-байтовое и 4-байтовое представление в виде экранированных последовательностей.

Даже если вы полагаете, что вам не понадобятся символы Юникода, вам, возможно, хотя бы неявно придется использовать их. Поскольку

некоторые программные интерфейсы (например, COM API в Windows и некоторые синтаксические анализаторы XML) представляют текст в кодировке Юникод, это может повлечь за собой необходимость выполнять преобразование между обычной кодировкой и кодировкой Юникод при передаче текста таким API или при получении результатов.

В большинстве случаев интерпретатор Python рассматривает эти два типа строк как взаимозаменяемые, поэтому присутствие строк символов Юникода часто проходит незамеченным для вашего программного кода – в большинстве случаев вы можете просто игнорировать тот факт, что текст передается между программными компонентами в виде объектов строк символов Юникода, и использовать обычные строковые операции.

Поддержка Юникода – это полезное встроенное дополнение в языке Python, поэтому данные в кодировке Юникод легко обрабатываются в сценариях. Безусловно, о Юникоде можно рассказать больше. Например:

- Объекты строк Юникода предоставляют метод, выполняющий преобразования строки символов Юникода в обычную строку 8-битовых символов с использованием указанной кодировки.
- Встроенная функция `unicode` и модуль `codecs` поддерживают зарегистрированные «кодеки» («codec», от слов «COders» и «DECoders») Юникода.
- Функция `open` из модуля `codecs` позволяет обрабатывать текстовые файлы в кодировке Юникод, где каждый символ может быть представлен более чем одним байтом.
- Модуль `unicodedata` предоставляет доступ к базе данных символов Юникода.
- Модуль `sys` включает функции для получения и изменения схемы кодирования Юникод (по умолчанию обычно используется ASCII).
- Допускается комбинировать форматы представления неформатированных строк и строк символов Юникода (например, `ur'a\b\c'`).

Поскольку тема использования кодировки Юникод является достаточно сложной, и это не самый распространенный инструмент, мы не будем обсуждать ее далее в этой вводной книге. Дополнительные сведения о Юникоде вы найдете в стандартном руководстве по языку Python.



В Python 3.0 строковый тип претерпит некоторые изменения: в версии 3.0 текущий тип `str` всегда будет представлять строки символов Юникода, а также появится новый тип – «байты», изменяемая последовательность малых целых чисел, удобная для представления коротких строк символов. Некоторые операции чтения из файла будут возвращать байты вместо типа `str` (например, при чтении двоичных файлов). Однако все это пока только в планах, поэтому подробности вам следует искать в примечаниях к выпуску 3.0.

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 демонстрируются основы работы со строками, форматирование и методы – первая 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 перечислены в табл. 7.1. Строки можно объединять с помощью оператора конкатенации + 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 строковый объект, объединяющий содержимое операндов. Операция повторения напоминает многократное сложение строки с самой собой. В обоих случаях Python позволяет создавать строки произвольного размера – нет никакой необходимости предварительно объявлять что бы то ни было, включая размеры структур данных.¹ Встроенная функция `len` возвращает длину строки (или любого другого объекта, который имеет длину).

Операция повторения на первый взгляд выглядит несколько странно, но она очень удобна в очень широком диапазоне применений. Напри-

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 объекты строк, когда в этом возникает необходимость, и позволяете интерпретатору самому управлять памятью, выделяемой для этих объектов. Как уже говорилось в предыдущей главе, 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 компонентами, которые ссылаются на него. Когда количество ссылок уменьшается до нуля, интерпретатор освобождает память, занятую объектом. Это означает, что интерпретатору не требуется останавливаться и просматривать всю память в поисках неиспользуемого пространства (дополнительный компонент сборки мусора собирает также циклические объекты).

мер, чтобы вывести 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 используем те же самые операторы `+` и `*`, которые используются для выполнения операций сложения и умножения с числами. Интерпретатор выполняет требуемую операцию, потому что ему известны типы объектов, к которым применяются операции сложения и умножения. Но будьте внимательны: правила не так либеральны, как может показаться. Например, интерпретатор не позволяет 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 элементов строки в цикле, используя инструкцию `for`, и проверять вхождение подстроки в строку с помощью оператора выражения `in`, который, по сути, выполняет операцию поиска:

```
>>> myjob = "hacker"
>>> for c in myjob: print c,      # Обход элементов строки в цикле
...
h a c k e r
>>> "k" in myjob                 # Найдено
True
>>> "z" in myjob                 # Не найдено
False
```

Оператор цикла `for` присваивает переменной очередной элемент последовательности (в данном случае — строки) и для каждого элемента выполняет одну или более инструкций. В результате переменная `c` превращается в своего рода курсор, который постепенно перемещается по строке. Далее в этой книге мы подробнее рассмотрим средства выполнения итераций, подобных этим.

Доступ по индексам и извлечение подстроки

Так как строки определены как упорядоченные коллекции символов, мы можем обращаться к элементам строк по номерам позиций. В языке Python символы извлекаются из строк с помощью операции *индексирования* — указанием числового смещения требуемого компонента в квадратных скобках после имени 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 исчисляться с 0, а последний символ в строке имеет смещение на единицу меньше 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 строки, чтобы получить положительное смещение. Отрицательные смещения можно 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 именем `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 эквивалентно смещению $(4 + -2)$ от начала). Смещения и срезы отображаются на элементы последовательности, как показано на рис. 7.1.¹

Последняя строка в предыдущем примере демонстрирует операцию *извлечения среза* (*slicing*). Получение среза можно считать некоторой формой синтаксического анализа, особенно при работе со строками – она позволяет извлекать 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 синтаксического анализа строки.

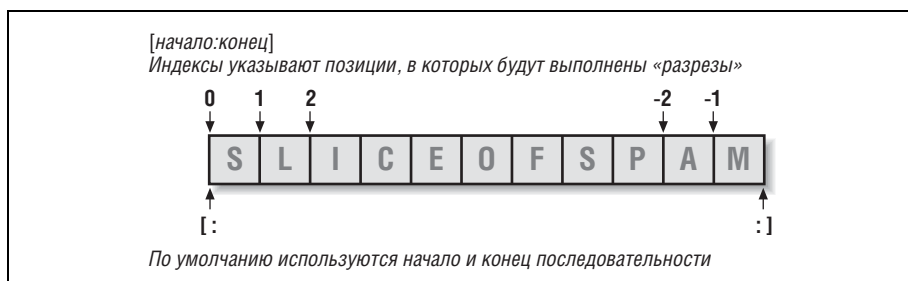


Рис. 7.1. Смещения и срезы: положительные смещения отсчитываются от левого конца (первый элемент имеет смещение 0), а отрицательные отсчитываются от правого конца (последний элемент имеет смещение -1). При выполнении операций индексации и получения среза можно использовать любые виды смещений

¹ Математически более подготовленные читатели (и студенты в моих классах) иногда обнаруживают здесь небольшую асимметрию: самый левый элемент имеет смещение 0, а самый правый имеет смещение -1. Увы, в языке Python отсутствует такая вещь, как значение -0.

Как же выполняется операция получения среза? Когда производится индексирование объекта последовательности, такого как строка, парой смещений, разделенных двоеточием, интерпретатор Python возвращает новый объект, содержащий непрерывную область, определяемую парой смещений. Значение смещения слева от двоеточия обозначает левую границу (включительно), а справа – верхнюю границу (она не входит в срез). Интерпретатор извлекает все элементы от нижней границы до верхней, но верхняя граница в срез не включается. Если левая и правая граница опущены, по умолчанию принимаются значения, равные 0 и длине объекта, из которого извлекается срез, соответственно.

Например, для только что рассмотренного примера выражение `S[1:3]` вернет элементы 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 последнего*, – 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 операции извлечения отдельных элементов и срезов превратятся в простые и мощные инструменты, как только вы ими овладеете. Если вы забыли, как выполняется срез, попробуйте получить его в интерактивном сеансе. В следующей главе вы увидите, что существует возможность изменить целый раздел определенного 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, равное 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 выражение `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 конца (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 последнего (не включая его).
- Выражение `S[:]` извлекает элементы, начиная со смещения 0 и до конца – это эффективный способ создать поверхностную копию последовательности `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 копировании 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 поддержка необязательного третьего индекса, используемого как *шаг* (иногда называется как *шаг по индексу*). Величина шага добавляется к индексу каждого извлекаемого элемента. Полная форма записи операции извлечения подстроки теперь выглядит так: `X[I:J:K]`. Она означает: «Извлечь все элементы последовательности `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 последовательности `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 пределы принимают значения 0 и длину последовательности соответственно, поэтому выражение `X[:2]` вернет каждый второй элемент от начала и до конца последовательности:

```
>>> S = 'abcdefghijklmnp'
>>> S[1:10:2]
'bdfhj'
>>> S[:2]
'acegikmo'
```

Можно также использовать отрицательное значение шага. Например, выражение `"hello"[::-1]` вернет новую строку `"olleh"`. Здесь первые две границы получают значения по умолчанию – 0 и длина последовательности, а величина шага, равная -1, указывает, что срез должен быть выбран в обратном порядке – 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 последовательности со смещениями 5, 4, 3 и 2):

```
>>> S = 'abcdefg'
>>> S[5:1:-1]
'fdec'
```

Пропуск элементов и изменение порядка их следования – это наиболее типичные случаи использования операции получения среза с тремя пределами. За более подробной информацией вам следует обратиться к руководству по стандартной библиотеке языка Python или поэкспериментировать в интерактивной оболочке – это расширит ваши представления о срезах по сравнению с объемом, который мы можем охватить здесь. Мы еще вернемся к операции получения среза с тремя пределами далее в этой книге, когда будем рассматривать ее в соединении с оператором цикла `for`.

Инструменты преобразования строк

Один из девизов языка Python – не поддаваться искушению предполагать, 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
(TypeError: невозможно объединить объекты 'str' и 'int')
```

В соответствии с архитектурой языка оператор `+` может означать и операцию сложения, и операцию конкатенации, вследствие чего выбор типа преобразования становится неочевидным. Поэтому интерпретатор воспринимает такую инструкцию как ошибочную. Вообще в языке Python отвергается любая магия, 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 инструменты преобразования, чтобы можно было интерпретировать строку как число или наоборот. Например:

```
>>> int("42"), str(42)  # Преобразование в/из строки
(42, '42')
>>> repr(42), '42'     # Преобразование в строку, как если бы она была
('42', '42')           # программным кодом
```

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, также преобразуют объект в строковое представление, но они возвращают объект в виде строки программного кода, который можно выполнить, чтобы воссоздать объект (если объект – строка, то результат, выведенный инструкцией `print`, будет включать кавычки, окружающие строку). Подробнее о различиях между функциями `str` и `repr` можно прочитать во врезке «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'
```

Придется держать в уме: срезы

Повсюду в этой книге я буду включать аналогичные врезки с описанием наиболее типичных случаев использования рассматриваемых особенностей языка на практике. Поскольку у вас нет возможности осознать реальный потенциал языка, пока вы не увидите большую часть картины, эти врезки будут содержать множество упоминаний тем, незнакомых для вас. Поэтому вам следует воспринимать эти сведения как предварительное знакомство со способами, которые связывают абстрактные концепции языка с решением наиболее распространенных задач программирования.

Например, далее вы увидите, что аргументы командной строки, переданные сценарию на языке Python при запуске, доступны в виде атрибута `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 операции получения среза (или, что то же самое, извлечения подстроки): мы можем с помощью единственной инструкции 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 заботясь о присутствии имени программы в начале.

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 извлечения подстроки обеспечивает логику выполнения, которую в низкоуровневых языках программирования пришлось бы реализовывать явно.

Обратите внимание: для удаления символа новой строки часто предпочтительнее использовать функцию `line.rstrip`, потому что она не повреждает строки, в которых отсутствует символ новой строки в конце, — типичный случай при создании текстовых файлов некоторыми текстовыми редакторами. Операция извлечения подстроки применима, только если вы полностью уверены, что строки завершаются корректным образом.

Существуют похожие встроенные функции для преобразования чисел с плавающей точкой 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 строку, содержащую программный код на языке Python, и потому может выполнять преобразование строки в объект любого вида.

Функции `int` и `float` преобразуют 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 форматирования строки также обеспечивает возможность преобразования чисел в строки. Форматирование будет обсуждаться ниже, в этой же главе.

Преобразование кодов символов

Возможно также выполнить преобразование одиночного символа в его целочисленный код ASCII, для чего нужно передать этот символ функции `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 соответствующий 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. Например, чтобы получить следующий по алфавиту символ, его можно преобразовать в число и выполнить математическое действие над ним:

```
>>> S = '5'
>>> S = chr(ord(S) + 1)
>>> S
'6'
>>> S = chr(ord(S) + 1)
>>> S
'7'
```

Следующий пример преобразования представляет собой альтернативу встроенной функции `int`, по крайней мере для односимвольных строк, для преобразования строки в целое число:

```
>>> int('5')
5
>>> ord('5') - ord('0')
5
```

Такие преобразования могут использоваться в сочетании с оператором цикла — для получения целочисленных значений из строковых представлений двоичных чисел — в каждой итерации текущее значение умножается на 2 и затем к нему прибавляется числовое значение следующей цифры:

```
>>> B = '1101'
>>> I = 0
>>> while B:
```



```
...     I = I * 2 + (ord(B[0]) - ord('0'))
...     B = B[1:]
...
>>> I
13
```

Оператор побитового сдвига влево ($I \ll 1$) мог бы дать тот же эффект, что и операция умножения на 2. Но, пока мы еще не познакомились с циклами, оставим реализацию этого упражнения как есть.

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!
```

Тогда каким образом в языке Python производить изменение текстовой информации? 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 затем присваивается имени `S`, но вы можете представить себе это действие как «изменение» первоначальной строки. Второй пример замещает четыре символа шестью новыми – с помощью операций извлечения подстроки и конкатенации. 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 каждого изменения – операция не столь неэффек-

тивная, 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 случаях работает гораздо быстрее, чем можно было бы ожидать.

Наконец, дополнительно существует возможность сборки текстовых значений с помощью выражений форматирования строк:

```
>>> 'That is %d %s bird!' % (1, 'dead') # Напоминает функцию sprintf в языке C
That is 1 dead bird!
```

Это очень мощная операция! В следующем разделе рассказывается о том, как она работает.

Форматирование строки

В языке Python имеется двухместный оператор `%`, предназначенный для работы со строками (вы можете вспомнить, что для чисел он еще является оператором деления по модулю или получения остатка от деления). Когда этот оператор применяется к строкам, он играет роль функции `sprintf` в языке C и обеспечивает простой способ форматирования значений в строке согласно заданной строке формата. Проще говоря, оператор `%` обеспечивает возможность компактной записи программного кода, выполняющего множественную подстановку строк.

Чтобы отформатировать строку, требуется:

1. Слева от оператора `%` указать строку формата, содержащую один или более спецификатор формата, каждый из которых начинается с символа `%` (например, `%d`).
2. Справа от оператора `%` указать объект (или объекты, в круглых скобках), которые будут подставляться на место спецификатора (или спецификаторов) в левой части выражения.

Например, в последнем примере предыдущего раздела мы видели, что целое число `1` подставляется на место спецификатора `%d` в строке формата, расположенной в левой части выражения, а строка `"dead"` подставляется на место спецификатора `%s`. В результате получается новая строка, которая содержит эти две подстановки.

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" подключается к целевой строке слева, замещая спецификатор %s. Во втором примере в целевую строку вставляются три значения. Обратите внимание: когда вставляется более одного значения, в правой части выражения их необходимо сгруппировать с помощью круглых скобок (то есть создать из них кортеж).

В третьем примере также вставляются три значения — целое число, объект числа с плавающей точкой и объект списка, но, обратите внимание, что в левой части выражения всем значениям соответствует спецификатор %s, который соответствует операции преобразования в строку. Объекты любого типа могут быть преобразованы в строку (это происходит, например, при выводе на экран), поэтому для любого объекта может быть указан спецификатор %s. Вследствие этого вам не придется выполнять специальное форматирование, в большинстве случаев вам достаточно 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 являются неизменяемыми, этот оператор вынужден работать именно таким способом. Как уже говорилось ранее, если вам требуется сохранить полученный результат, присвойте его переменной.

Дополнительные возможности форматирования строки

Для реализации более сложного форматирования в выражениях форматирования можно использовать любые спецификаторы формата, представленные в табл. 7.3. Большинство из них окажутся знакомы программистам, использовавшим язык C, потому что операция форматирования строк в языке Python поддерживает все наиболее типичные спецификаторы формата, которые допускается использовать в функции printf языка C (в отличие от которой выражение в языке Python возвращает результат, а не выводит его на экран). Некоторые спецификаторы из табл. 7.3 предоставляют альтернативные способы форматирования данных одного и того же типа, например, %e, %f и %g обеспечивают альтернативные способы форматирования чисел с плавающей точкой.

Фактически спецификаторы формата в левой части выражения поддерживают целый набор операций преобразования с достаточно сложным собственным синтаксисом. В общем виде синтаксис использования спецификатора формата выглядит следующим образом:

```
%(name)[flags][width][.precision]code
```

Таблица 7.3. Спецификаторы формата

Спецификатор	Назначение
%s	Строка (или любой объект)
%r	s, но использует функцию repr, а не str
%c	Символ
%d	Десятичное (целое) число
%i	Целое число
%u	Целое число без знака
%o	Восьмеричное целое число
%x	Шестнадцатеричное целое число
%X	x, но шестнадцатеричные цифры возвращаются в верхнем регистре
%e	Число с плавающей точкой в экспоненциальной форме
%E	e, но алфавитные символы возвращаются в верхнем регистре
%f	Число с плавающей точкой в десятичном представлении
%g	Число с плавающей точкой e или f
%G	Число с плавающей точкой E или f
%%	Символ %

Символ спецификатора формата (*code*) из табл. 7.3 располагается в самом конце. Между символом % и символом спецификатора можно добавлять следующую информацию: ключ в словаре (*name*); список флагов (*flags*), которые могут определять, например, признак выравнивания (-), знак числа (+), наличие ведущих нулей (0); общую ширину поля и число знаков после десятичной точки и многое другое.

Полное описание синтаксиса спецификаторов формата вы найдете в стандартном руководстве по языку Python, а сейчас для демонстрации наиболее типичных случаев их использования приведем несколько примеров. В первом примере 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 числа с плавающей точкой разными способами, как демонстрируется в следующем примере:

```
>>> x = 1.23456789
>>> x
```

```
1.23456788999999999
>>> '%e | %f | %g' % (x, x, x)
'1.234568e+000 | 1.234568 | 1.23457'
```

Для чисел с плавающей точкой можно реализовать дополнительные эффекты форматирования, указав необходимость выравнивания по левому краю, знак числа, ширину поля и число знаков после десятичной точки. Для простых задач можно было бы использовать простые функции преобразования 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'

>>> "%s" % x, str(x)
('1.23456789', '1.23456789')
```

Форматирование строк из словаря

Операция форматирования позволяет также использовать в спецификаторах формата ключи словаря, который указывается в правой части выражения, для извлечения соответствующих значений. Мы пока немного говорили 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 правой части выражения и служат для извлечения соответствующих им значений. 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 = ""
Greetings...
Hello %(name)s!
Your age squared is %(age)s
""

>>> values = {'name': 'Bob', 'age': 40}
>>> print reply % values

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, существующие на момент ее вызова:

```
>>> food = 'spam'
>>> age = 40
```

```
>>> vars()
{'food': 'spam', 'age': 40, ...и еще множество других... }
```

Если применить эту функцию в правой части оператора форматирования, можно отформатировать значения, обращаясь к ним по именам переменных (то есть по ключам словаря):

```
>>> "%(age)d %(food)s" % vars()
'40 spam'
```

Словари во всех подробностях мы будем изучать в главе 8. А в главе 5 вы найдете примеры использования спецификаторов `%x` и `%o` для преобразования значений в шестнадцатеричное и восьмеричное строковое представление.

Строковые методы

В дополнение к операторам выражений строки предоставляют набор *методов*, реализующих более сложные операции обработки текста. Методы – это простые функции, которые связаны с определенными объектами. Формально они являются атрибутами, присоединенными к объектам, которые ссылаются на функции. В языке Python методы являются специфичными для типов объектов: строковые методы, например, работают только с строковыми объектами.

Если говорить более точно, функции – это пакеты программного кода, а вызовы методов объединяют в себе выполнение двух операций (извлечение атрибута и вызов функции).

Извлечение атрибута

Выражение в форме *object.attribute* означает: «извлечь значение атрибута *attribute* из объекта *object*».

Вызов функции

Выражение в форме *function(arguments)* означает: «вызвать программный код функции *function*, передав ему ноль или более объектов-аргументов, разделенных запятыми, и вернуть значение функции».

Объединение этих двух действий позволяет вызвать метод объекта. Выражение вызова метода *object.method(arguments)* вычисляется слева направо, то есть интерпретатор сначала извлекает метод объекта, а затем вызывает его, передавая 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 обладает 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 некоторых примеров.

Примеры использования строковых методов: изменение строк

В табл. 7.4 приводятся шаблоны вызова встроенных строковых методов (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). Строковые методы, представленные в этой таблице, реализуют высокоуровневые операции, такие как разбиение и слияние, преобразование регистра символов, проверка типа содержимого и поиск подстроки.

Таблица 7.4. Строковые методы

<code>S.capitalize()</code>	<code>S.ljust(width)</code>
<code>S.center(width)</code>	<code>S.lower()</code>
<code>S.count(sub [, start [, end]])</code>	<code>S.lstrip()</code>
<code>S.encode([encoding [,errors]])</code>	<code>S.replace(old, new [, maxsplit])</code>
<code>S.endswith(suffix [, start [, end]])</code>	<code>S.rfind(sub [,start [,end]])</code>
<code>S.expandtabs([tabsize])</code>	<code>S.rindex(sub [, start [, end]])</code>
<code>S.find(sub [, start [, end]])</code>	<code>S.rjust(width)</code>
<code>S.index(sub [, start [, end]])</code>	<code>S.rstrip()</code>
<code>S.isalnum()</code>	<code>S.split([sep [,maxsplit]])</code>
<code>S.isalpha()</code>	<code>S.splitlines([keepends])</code>
<code>S.isdigit()</code>	<code>S.startswith(prefix [, start [, end]])</code>
<code>S.islower()</code>	<code>S.strip()</code>
<code>S.isspace()</code>	<code>S.swapcase()</code>
<code>S.istitle()</code>	<code>S.title()</code>
<code>S.isupper()</code>	<code>S.translate(table [, delchars])</code>
<code>S.join(seq)</code>	<code>S.upper()</code>

Теперь давайте поработаем над программным кодом, который демонстрирует некоторые наиболее часто используемые методы в действии и попутно иллюстрирует основные приемы обработки текста, применяемые в языке Python. 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 использовать 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 программный 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 присваиваем его переменной – присваивать результат переменной необходимо только в том случае, если потребуется сохранить результат дальнейшего использования.

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 операциями 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'
```

Метод `find` возвращает смещение, по которому найдена подстрока (по умолчанию поиск начинается с начала строки), или значение `-1`, если искомая подстрока не найдена. Другой вариант использования метода `replace` заключается в передаче третьего аргумента, который определяет число производимых замен:

```
>>> 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 действительно-

сти не изменяют оригинальную строку, даже если они называются «replace» (заменить)!

Тот факт, что операция конкатенации и метод `replace` всякий раз создают новые строковые объекты, может оказаться недостатком их использования для изменения строк. Если в сценарии производится множество изменений длинных 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 представлении, можно производить необходимые изменения, не вызывая создания 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 преобразование (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 указание 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'
```

Примеры методов строк: разбор текста

Еще одна распространенная роль, которую играют методы строк, – это простейший *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 потому могут быть легко извлечены из оригинальной строки. Этот прием может использоваться, только если анализируемые компоненты располагаются в известных фиксированных позициях. Если для разделения полей используется некоторый разделитель, можно воспользоваться методом разбиения строки на компоненты. Этот прием используется, когда искомые данные могут располагаться в произвольных позициях внутри строки:

```
>>> line = 'aaa bbb ccc'
>>> cols = line.split()
>>> cols
['aaa', 'bbb', 'ccc']
```

Строковый метод `split` преобразует строку в список подстрок, окружающих строку-разделитель. В предыдущем примере мы не указали строку-разделитель, поэтому в данном случае по умолчанию в качестве разделителей принимаются 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 следующем примере производится разбиение (и, следовательно, разбор) строки по символу запятой, который обычно используется для отделения данных, извлеченных 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']
```

Хотя оба способа, основанные на извлечении подстрок и разбиении строк, имеют определенные ограничения, они работают достаточно быстро и могут использоваться для разбора текстовой информации в простых случаях.

Другие часто используемые методы строк в действии

Другие строковые методы имеют более специфическое предназначение, например, удаляют пробельные символы в конце текстовой строки, вы-

полняют преобразование регистра символов, проверяют характер содержимого строки и проверяют наличие подстроки в конце строки:

```
>>> line = "The knights who sy Ni!\n"
>>> line.rstrip()
'The knights who sy Ni!'
>>> line.upper()
'THE KNIGHTS WHO SY NI!\n'
>>> line.isalpha()
False
>>> line.endswith('Ni!\n')
True
```

Для достижения тех же результатов в некоторых случаях могут использоваться альтернативные приемы – с использованием оператора проверки вхождения `in` можно проверить присутствие подстроки, например, а функция получения длины строки и операция извлечения подстроки могут использоваться для имитации действия функции `endswith`:

```
>>> line
'The knights who sy 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
```

Для работы со строками существует достаточно много методов, однако мы не будем рассматривать их все. Некоторые методы вы увидите далее в этой книге, а за дополнительной информацией вы можете обратиться к руководству по библиотеке языка Python и другим источникам информации или просто поэкспериментировать с ними в интерактивном режиме.

Обратите внимание: ни один из строковых методов не поддерживает шаблоны, для обработки текста с использованием шаблонов необходимо использовать модуль `re`, входящий в состав стандартной библиотеки языка Python, – дополнительный инструмент, начальные сведения о котором приводились в главе 4, а полное его обсуждение выходит далеко за рамки этой книги. Тем не менее, несмотря на это ограничение, строковые методы иногда оказываются эффективнее, чем функции модуля `re`.

Оригинальный модуль `string`

История развития строковых методов достаточно запутанна. В течение первого десятилетия существования Python в состав стандартной

библиотеки входил модуль `string`, который содержал функции, во многом напоминающие современные 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 строковые методы. Фактически, как предполагается, оригинальные строковые функции, соответствующие методам, будут убраны из состава стандартной библиотеки языка Python в версии 3.0, которая должна быть выпущена после выхода в свет этого издания. Однако, поскольку использование модуля `string` еще можно встретить в старом программном коде, мы коротко рассмотрим его.

В версии Python 2.5 по-прежнему существует два способа использования расширенных операций над строками: посредством вызова методов и вызовом функций модуля `string`, которым в качестве аргумента передается объект строки. 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 протяжении многих лет, а строки являются одним из центральных компонентов большинства программ, при анализе существующих программ вам в программном коде наверняка встретятся оба варианта работы со строками.

И, тем не менее, при создании новых программ вместо функций устаревшего модуля вы должны использовать строковые методы. Для это-

го есть и другие серьезные основания помимо того, что многие функции модуля `string` будут исключены из состава стандартной библиотеки с выходом Python 3.0. Одна из таких 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 модуль 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` скорее всего останется в составе Python 3.0, потому что в нем присутствуют дополнительные средства работы со строками, включая предопределенные строковые константы и систему шаблонных объектов (которая не описывается в этой книге; за дополнительной информацией обращайтесь к руководству по библиотеке языка Python). Если вы не хотите модифицировать свой программный код после выхода версии 3.0, вы должны смотреть на базовые строковые функции как на призраки прошлого.

Общие категории типов

Теперь, когда мы исследовали строки, первый объект из коллекции языка Python, сделаем паузу, чтобы дать определение некоторым общим концепциям, применимым к большинству типов, которые будут рассматриваться дальше. 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 только 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 подумать.

Типы одной категории имеют общий набор операций

Как вы уже знаете, строки представляют собой неизменяемые последовательности: они не могут быть изменены непосредственно (*неизменяемые*) и являются упорядоченными коллекциями элементов, доступ к которым может осуществляться 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 цикле и т. д. Формально в языке Python существует три категории типов (и операций):

Числа

Поддерживают операции сложения, умножения и т. д.

Последовательности

Поддерживают операции индексации, извлечения среза, конкатенации и т. д.

Отображения

Поддерживают операцию индексации по ключу и т. д.

Мы еще не рассматривали подробно отображения (словари обсуждаются в следующей главе), но к другим типам, которые нам встретятся, в основном будут применимы одни и те же операции. Например, для любых объектов последовательностей 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 последовательностей, включая строки, списки, кортежи и некоторые типы, определяемые пользователем. Единственное отличие состоит в том, что результат, возвращаемый 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 существенное ограничение, чтобы не забыть о нем, и все же она часто сбивает с толку начинающих программистов. Если объект является неизменяемым типом, вы не сможете изменить его значение непосредственно – в этом случае интерпретатор будет выдавать сообщение об ошибке. Вместо этого необходимо, чтобы программный код создавал новый объект, содержащий новое значение. Вообще неизменяемые типы обеспечивают определенный уровень поддержки целостности, гарантируя, что объект не подвергнется изменениям в другой части программы. Чтобы вспомнить, почему это имеет такое большое значение, вернитесь к дискуссии о разделяемых ссылках на объекты в главе 6.

В заключение

В этой главе мы подробно рассмотрели строковый тип объектов. Мы узнали о строковых литералах и исследовали операции над строками, включая операции над последовательностями, форматирование и методы. Попутно мы подробно изучили различные концепции, такие как извлечение среза, вызовы методов и блоки строк в тройных кавычках.

Кроме того, мы определили некоторые идеи, общие для различных типов: последовательности, к примеру, поддерживают общий набор операций. В следующей главе мы продолжим наши исследования и рассмотрим наиболее 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. 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 строковых методов?

Ответы

1. Нет, потому что методы всегда зависят от типа объекта. То есть, они могут применяться только к одному определенному типу данных. Но выражения являются более универсальными и могут использоваться для выполнения операций над различными типами. В этом случае, к примеру, оператор проверки вхождения `in` имеет похожий эффект и может использоваться для поиска как в строках, так и в списках.
2. Да. В отличие от методов, выражения универсальны и могут использоваться для выполнения операций над различными типами. В данном случае операция извлечения среза в действительности является операцией над последовательностями – она может применяться к любому типу последовательностей, включая строки, списки и кортежи. Различие состоит лишь в том, что при выполнении операции над списком в результате будет получен новый список.
3. Встроенная функция `ord(S)` преобразует односимвольную строку в целочисленный код. Функция `chr(I)` преобразует целочисленный код в символ.

4. Строки невозможно изменить – они являются неизменяемыми. Однако подобного эффекта можно добиться, создав новую строку, – выполнив операцию конкатенации, извлечения среза, форматирования или вызвав метод, такой как `replace`, – и затем присвоив результат первоначальной переменной.
5. Можно извлечь подстроку с помощью выражения `S[2:4]` или, разбив строку по запятым, извлечь строку с помощью выражения `S.split(',')[1]`. Попробуйте 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 новой строки (`\n`), `b`, символ с десятичным кодом 31 (`\x1f` – в шестнадцатеричном представлении), символ с кодом 0 (`\000` – в восьмеричном представлении) и `d`. Передайте строку встроенной функции `len`, чтобы проверить этот ответ, и выведите результаты преобразования каждого символа с помощью функции `ord`, чтобы увидеть фактические значения кодов символов. За более подробной информацией обращайтесь к табл. 7.2.
7. Вам не следует использовать модуль `string`. При создании новых программ следует использовать строковые методы – функции этого модуля считаются устаревшими и скорее всего будут исключены из Python 3.0. Единственная причина использования модуля `string` – это применение других средств из этого модуля, таких как предопределенные константы и шаблонные объекты.

8

Списки и словари

В этой главе вашему вниманию будут представлены такие типы объектов, как списки и словари, каждый из которых является коллекцией других объектов. Эти два типа являются основными рабочими лошадками практически во всех сценариях на языке Python. Как вы увидите далее, оба типа обладают 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 самый гибкий тип упорядоченных коллекций в языке Python. В отличие от строк списки могут содержать объекты любых типов: числа, строки и даже другие списки. Кроме того, в отличие от строк, списки могут изменяться непосредственно, с помощью операции присваивания по смещениям и срезам, с помощью методов списков, с использованием инструкций удаления и другими способами – списки являются изменяемыми объектами.

Списки в языке Python реализуют практически все необходимое для работы с коллекциями данных, что вам пришлось бы писать вручную при использовании 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 собраны другие объекты, поэтому их можно также рассматривать

как группы. Кроме того, списки обеспечивают позиционное упорядочение элементов слева направо (то есть они являются последовательностями).

Доступ к элементам по смещению

Так же, как и в случае со строками, вы можете использовать операцию индексирования для извлечения отдельных объектов из списка по их смещениям. Поскольку элементы в списках упорядочены по их местоположению, можно также выполнять такие действия, как извлечение срезов и конкатенация.

Переменная длина, гетерогенность и произвольное число уровней вложенности

В отличие от строк списки могут увеличиваться и уменьшаться непосредственно (их длина может изменяться) и могут содержать не только односимвольные строки, но и любые другие объекты (списки гетерогенны). Списки могут содержать другие сложные объекты и поддерживают возможность создания произвольного числа уровней вложенности, поэтому имеется возможность создавать списки из списков из списков и т. д.

Относятся к категории изменяемых объектов

В терминах категорий типов списки могут изменяться непосредственно (являются изменяемыми объектами) и поддерживают все операции над последовательностями, которые поддерживаются и строками, такие как индексирование, извлечение срезов и конкатенация. Операции над последовательностями одинаковым образом работают как в случае списков, так и в случае строк, единственное отличие – при применении операций над последовательностями (таких как конкатенация и извлечение среза) к спискам операций возвращают новый список, а не новую строку. Кроме того, т. к. списки являются изменяемыми объектами, они поддерживают также операции, которые не поддерживаются строками (такие как операции удаления и присваивания по индексам, которые изменяют 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 (адресов). 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 интерпретаторе Python, а не связанные структуры данных. Как мы узнали в главе 6, всякий раз, когда используется ссылка на объект, интерпретатор разыменовывает ее, поэтому ваши программы всегда будут иметь дело только с объектами. Всякий раз, когда выполняется присваивание объекта элементу какой-либо структуры или имени переменной, интерпретатор Python

сохраняет 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 обратиться к руководству по стандартной библиотеке языка Python или запустить `help(list)` или `dir(list)` в интерактивной оболочке, чтобы получить полный список методов списков – этим функциям можно передать существующий список или слово `list`, которое является именем типа данных ‘список’.

Таблица 8.1. Литералы списков и операции

Операция	Интерпретация
<code>L1 = []</code>	Пустой список
<code>L2 = [0, 1, 2, 3]</code>	Четыре элемента с индексами 0..3
<code>L3 = ['abc', ['def', 'ghi']]</code>	Вложенные списки
<code>L2[i]</code> <code>L3[i][j]</code> <code>L2[i:j]</code> <code>len(L2)</code>	Индекс, индекс индекса, срез, длина
<code>L1 + L2</code> <code>L2 * 3</code>	Конкатенация, дублирование
<code>for x in L2</code> <code>3 in L2</code>	Обход в цикле, проверка вхождения
<code>L2.append(4)</code> <code>L2.extend([5, 6, 7])</code> <code>L2.sort()</code> <code>L2.index(1)</code> <code>L2.insert(1, X)</code> <code>L2.reverse()</code>	Методы: дополнение списка, сортировка, поиск, вставка, изменение порядка следования элементов на обратный и т. д.
<code>del L2[k]</code> <code>del L2[i:j]</code> <code>L2.pop()</code> <code>L2.remove(2)</code> <code>L2[i:j] = []</code>	Уменьшение списка
<code>L2[i] = 1</code> <code>L2[i:j] = [4, 5, 6]</code>	Присваивание по индексу, присваивание срезу
<code>range(4)</code> <code>xrange(0, 4)</code>	Создание списков/кортежей целых чисел
<code>L4 = [x**2 for x in range(5)]</code>	Дополнительные операции над списками (главы 13 и 17)

Когда список определяется литеральным выражением, он записывается как последовательность объектов (точнее, как последовательность

выражений, создающих объекты) в квадратных скобках, разделенных запятыми. Например, вторая строка в табл. 8.1 присваивает переменной `l2` список из четырех элементов. Вложенные списки описываются как вложенные последовательности 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, которые мы применяли к строкам, – индексирование, конкатенация, обход элементов в цикле и т. д. Кроме того, списки поддерживают специфические для своего типа методы (такие как сортировка, перестановка элементов в обратном порядке, добавление элементов в конец списка и т. д.), а также операции непосредственного изменения списка (удаление элементов, присваивание по индексам и срезам и т. д.). 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 посмотреть на них в действии. Давайте еще раз вернемся к интерактивному сеансу работы с интерпретатором и проиллюстрируем операции из табл. 8.1.

Базовые операции над списками

Списки, как и строки, поддерживают операторы `+` и `*` – для списков они так же соответствуют операции конкатенации и повторения, но в результате получается новый список, а не строка. Фактически списки поддерживают все общие операции над последовательностями, которые мы рассматривали применительно к строкам в предыдущей главе.

```
% 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!']
>>> 3 in [1, 2, 3]           # Проверка на вхождение
True
>>> for x in [1, 2, 3]: print x, # Обход в цикле
```

¹ На практике в текстах программ обработки списков вы нечасто встретите списки, заданные подобным образом. Чаще вам будет встречаться программный код, обрабатывающий списки, которые создаются динамически (во время выполнения). Несмотря на всю важность владения синтаксисом литералов, следует иметь в виду, что большая часть структур данных в языке Python конструируется во время выполнения программы.

```
...
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 отношение к синтаксису инструкций. Говоря вкратце, оператор цикла `for` выбирает элементы последовательности в порядке слева направо и выполняет одну или более инструкций для каждого из них. Последняя строка в табл. 8.1 – это одна из дополнительных операций над списками, которые описываются в главе 13 и расширенное обсуждение которых приводится в главе 17. Как говорилось в главе 4, они представляют собой способ построить список, применяя выражение к каждому элементу последовательности, единственной инструкцией.

Несмотря на то, что оператор `+` со списками работает точно так же, как и со строками, очень важно знать, что с обеих сторон оператора должны находиться последовательности одного и того же типа, в противном случае во время работы программного кода вы получите сообщение об ошибке. Например, нельзя выполнить операцию конкатенации для списка и строки, если предварительно не преобразовать список в строку (используя, например, обратные апострофы (backquotes), функцию `str` или `%`-форматирование) или строку в список (с помощью встроенной функции `list`):

```
>>> 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']
```

Индексы, срезы и матрицы

Так как списки являются последовательностями, операции доступа к элементам по индексам и извлечения срезов работают точно так же, как и в случае со строками. Однако, в результате обращения к элементу по индексу возвращается объект, который расположен по указанному смещению, а в результате операции извлечения среза 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 типов) могут быть вложены в другие списки, иногда бывает необходимо объединять 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 3×3 , 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
```

Обратите внимание на то, что, как показано в предыдущем примере, список может быть при необходимости растянут на несколько строк, ограниченных квадратными скобками (подробнее о синтаксисе будет говориться в следующей части книги). Далее в этой главе вы также увидите матрицы, реализованные на базе словарей. Модуль NumPy, упомянутый в главе 5, для высокопроизводительной работы с числовыми данными предоставляет другие способы организации матриц.

Изменение списка

Списки относятся к категории изменяемых объектов, поэтому они поддерживают 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 строками. В языке Python приходится иметь дело только со ссылками на объекты, что обуславливает 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 содержимое, 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 – производятся 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 программирования: интерпретатор замещает старую ссылку на объект в указанном смещении на новую.

Присваивание срезу, последняя операция в предыдущем примере, замещает целый раздел списка за один прием. Поскольку это довольно сложная операция, проще будет представить ее как последовательное выполнение двух действий:

1. *Удаление*. Раздел списка, определяемый слева от оператора =, удаляется.
2. *Вставка*. Новые элементы, содержащиеся в объекте, расположенном справа от оператора =, вставляются в список, начиная с левого края, где находился прежний удаленный срез.

В действительности это не совсем то, что происходит на самом деле, но это достаточно точно объясняет, почему число вставляемых элементов не должно соответствовать 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, состоящий 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 действительности является операцией удаления, – интерпретатор удалит срез (элемент 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 выполняется присваивание, эта операция может использоваться для замены (посредством перезаписи), расширения (посредством вставки) или сжатия (посредством удаления) требуемого списка. Это довольно мощная операция, но, честно говоря, она достаточно редко используется на практике. Обычно используются более простые способы замены, вставки и удаления (например, операция конкатенация и методы списков `insert`, `pop` и `remove`), которые программисты предпочитают использовать на практике.

Методы списков

Как и строки, объекты списков в языке Python поддерживают специфичные методы:

```
>>> 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 напомним, 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`, который просто добавляет единственный элемент (ссылку на объект) в конец списка. В отличие от операции конкатенации, метод `append` принимает единственный 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 – создается новый список.¹ Другой часто используемый метод – метод `sort` – выполняет переупорядочивание элементов в самом списке. По умолчанию он использует стандартные операторы сравнения языка Python (в данном случае выполняется сравнение строк) и выполняет сортировку в порядке возрастания значений. Однако существует возможность передать методу `sort` свою собственную функцию сравнения.



В Python версии 2.5 и более ранних сравнение выполняется поразному для объектов разных типов (например, списков и строк) – язык определяет способ упорядочения различных типов, который является детерминистским и не всегда может выглядеть правильным. Этот способ упорядочения основан на именах типов, вовлеченных в операцию 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 сравнения объектов числовых типов.

В Python 3.0 такой порядок, возможно, будет изменен: планируется, что в случае попытки сравнения объектов различных типов вместо прежнего способа упорядочения будет возбуждаться исключение. Так как метод сортировки использует операцию сравнения, это означает, что инструкция `[1, 2, 'spam'].sort()` будет успешно выполнена в Python 2.5, но будет возбуждать исключение в Python 3.0. За дополнительной информацией обращайтесь к примечаниям к выпуску Python 3.0.

¹ В отличие от операции конкатенации (+), метод `append` не создает новый объект, поэтому обычно он выполняется быстрее. Существует возможность имитировать работу метода `append` с помощью операции присваивания срезу: выражение `L[len(L):]=[X]` соответствует вызову `L.append(X)`, а выражение `L[:0]=[X]` соответствует операции добавления в начало списка. В обоих случаях удаляется пустой сегмент списка и вставляется элемент `X`, при этом изменяется сам список `L`, так же быстро, как при использовании метода `append`.

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 получите измененное значение `L` (в действительности вы совсем потеряете ссылку на список) – использование таких атрибутов, как `append` и `sort` приводит к изменению самого объекта, поэтому нет никаких причин выполнять повторное присваивание.

Как и строки, списки обладают рядом других методов, выполняющих специализированные операции. Например, метод `reverse` изменяет порядок следования элементов в списке на обратный, а методы `extend` и `pop` вставляют несколько элементов в конец списка и удаляют элементы из конца списка, соответственно:

```
>>> 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]
```

В некоторых типах программ метод `pop`, показанный здесь, часто используется в паре с методом `append` для реализации структур данных типа *стек* – «последний пришел, первый ушел» (Last-In-First-Out, LIFO). Конец списка служит вершиной стека:

```
>>> 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 необязательное 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 заданному значению (`index`) и т. д. Чтобы поближе познакомиться с этими методами, обратитесь к имеющимся источникам документации или поэкспериментируйте с этими методами в интерактивной оболочке интерпретатора.

Прочие часто используемые операции над списками

Так как списки относятся к категории изменяемых объектов, вы можете использовать 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]=[]`) – интерпретатор сначала удалит срез, определяемый слева от оператора `=`, а затем вставит 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:

```
>>> L = ['Already', 'got', 'one']
>>> L[1:] = []
>>> L
['Already']
>>> L[0] = []
>>> L
[[]]
```

Все только что рассмотренные операции используются достаточно часто, однако существуют и другие дополнительные методы и операции для списков, которые не были показаны здесь (включая методы вставки и поиска). Чтобы получить полный список существующих дополнительных операций, всегда следует обращаться к руководствам по языку Python, к функциям `dir` и `help` (с которыми мы впервые познакомились в главе 4) или к книге «Python Pocket Reference» (O'Reilly) и другим справочным руководствам, упоминавшимся в предисловии.

Кроме того, я хотел бы напомнить, что все операции непосредственного изменения объектов, обсуждавшиеся здесь, применимы только к изменяемым объектам: они не будут работать со строками (или с кортежами, которые рассматриваются в следующей главе) независимо от прикладываемых вами усилий. Изменяемость или неизменяемость – это исходное свойство, присущее каждому типу объектов.

Словари

После списков *словари* являются, пожалуй, самым гибким из встроенных типов данных в языке Python. Если считать списки упорядоченными коллекциями объектов, то в отличие от них элементы в слова-

рях сохраняются и извлекаются с помощью *ключа*, а не с помощью смещения, определяющего их позицию.

Будучи встроенным типом данных, словари могут заменить множество алгоритмов поиска и структур данных, которые приходится реализовывать вручную в низкоуровневых языках программирования, – доступ к элементам словаря по их индексам представляет собой быструю операцию поиска. Кроме того, иногда словари могут играть роль записей и таблиц символов, используемых в других языках, и способны служить для представления разреженных (по большей части пустых) структур данных. Ниже приводятся основные характеристики словарей в языке Python:

Доступ к элементам по ключу, а не по индексу

Иногда словари называют *ассоциативными массивами*, или *хеши-ми*. Они определяют взаимосвязь между значениями и ключами, поэтому для извлечения элементов словаря можно использовать ключи, под которыми эти элементы были сохранены в словаре. Для получения элементов словаря используется та же самая операция доступа по индексу, как и в списке, только индекс приобретает форму ключа, а не смещения относительно начала.

Неупорядоченные коллекции произвольных объектов

В отличие от списков, элементы словарей хранятся в неопределенном порядке. В действительности, интерпретатор вносит элемент случайности в порядок следования элементов для обеспечения более быстрого поиска. Ключи описывают символическое (не физическое) местоположение элементов в словаре.

Переменная длина, гетерогенность и произвольное число уровней вложенности

Подобно спискам словари могут увеличиваться и уменьшаться непосредственно (то есть при этом не создаются новые копии). Они могут содержать объекты любых типов и поддерживают возможность создания произвольного числа уровней вложенности (они могут содержать списки, другие словари и т. д.).

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 среза), не имеют смысла для словарей. Словари – это единственный встроенный 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 неупорядоченные таблицы ссылок на объекты, которые поддерживают доступ к элементам по ключу. Внутри словари реализованы как хеш-таблицы (структуры данных, которые обеспечивают очень высокую скорость поиска), изначально небольшого размера и увеличивающиеся по мере необходимости. 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, обращайтесь 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 пустая пара скобок. Словари могут вкладываться в литеральном представлении в виде значений внутри других словарей, списков или кортежей.

Таблица 8.2. Литералы словарей и операции

Операция	Интерпретация
<code>D1 = {}</code>	Пустой словарь
<code>D2 = {'spam': 2, 'eggs': 3}</code>	Словарь из двух элементов
<code>D3 = {'food': {'ham': 1, 'egg': 2}}</code>	Вложение
<code>D2['eggs']</code> <code>D3['food']['ham']</code>	Доступ к элементу по ключу
<code>D2.has_key('eggs')</code> <code>'eggs' in D2</code> <code>D2.keys()</code> <code>D2.values()</code> <code>D2.copy()</code> <code>D2.get(key, default)</code> <code>D2.update(D1)</code> <code>D2.pop(key)</code>	Методы: проверка на вхождение, список ключей, список значений, копирование, получение значения по умолчанию, слияние, удаление и т. д.
<code>len(D1)</code>	Длина (количество элементов)

¹ Как и в случае со списками, вам нечасто придется встречать словари, сконструированные с использованием литералов. Однако списки и словари увеличиваются в размерах по-разному. Как будет показано в следующем разделе, словари обычно дополняются с помощью операции присваивания по новым ключам во время выполнения программы – такой подход совершенно не годится для списков (списки обычно расширяются с помощью метода `append`).

Операция	Интерпретация
D2[key] = 42 del D2[key]	Добавление/изменение ключей, удаление ключей
D4 = dict.fromvalues(['a', 'b']) D5 = dict(zip(keylist, valslist)) D6 = dict(name='Bob', age=42)	Альтернативные способы создания словарей

Словари в действии

Согласно табл. 8.2 доступ к элементам словарей осуществляется по ключу, а для доступа к элементам вложенных словарей осуществляется путем объединения серии индексов (ключей в квадратных скобках) в цепочку. Когда интерпретатор создает словарь, он сохраняет элементы в произвольном порядке – 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.

Базовые операции над словарями

В обычном случае сначала создается словарь, а затем выполняются операции сохранения новых ключей и обращения к элементам по ключу:

```
% python
>>> d2 = {'spam': 2, 'ham': 1, 'eggs': 3} # Создание словаря
>>> d2['spam']                               # Извлечение значения по ключу
2
>>> d2                                         # Случайный порядок следования
{'eggs': 3, 'ham': 1, 'spam': 2}
```

Здесь переменной d2 присваивается словарь, в котором ключу 'spam' соответствует целочисленное значение 2, и т. д. Для доступа к элементам словаря 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 следования ключей в словари практически всегда отличается от первоначального. Дело в том, что для обеспечения максимально высокой скорости поиска по ключу (для хеширования) необходимо случайное расположение ключей в памяти. Именно поэтому операции, которые предполагают наличие установленного 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 число элементов в словаре или, что то же самое, длину списка ключей. Метод словаря has_key и оператор проверки вхождения in

позволяют проверить наличие ключа, а метод `keys` возвращает все ключи, имеющиеся в словаре, в виде списка. Последний удобно использовать для последовательной обработки словарей, но при этом вы не должны делать какие-либо предположения о порядке следования ключей в списке. Поскольку результатом вызова метода `keys` является список, он всегда может быть отсортирован:

```
>>> len(d2)           # Число элементов словаря
3
>>> d2.has_key('ham') # Проверка на вхождение
True
>>> 'ham' in d2        # Проверка на вхождение, альтернативный способ
True
>>> d2.keys()          # Создает новый список ключей
['eggs', 'ham', 'spam']
```

Обратите внимание на третье выражение в этом листинге. Как уже упоминалось ранее, оператор проверки на вхождение `in` может использоваться для работы со строками и списками, но точно так же он может использоваться и для работы со словарями. Он проверяет наличие указанного ключа в словаре точно так же, как и метод `has_key` строкой выше. Это возможно благодаря тому, что словари определяют *итераторы*, которые обеспечивают пошаговый обход списков ключей. Существуют и другие типы, которые поддерживают итераторы, отвечающие обычному использованию типа; например, файлы имеют итераторы, которые позволяют выполнять построчное чтение данных. Итераторы будут рассматриваться в главах 17 и 24.

Далее в этой главе и в книге вы узнаете о двух альтернативных способах создания словарей, которые демонстрируются в конце табл. 8.2: можно передать функции `dict` (которая фактически является конструктором типа) упакованные списки ключей/значений в виде кортежей или значения в виде именованных аргументов. Передача значений в виде именованных аргументов будет рассматриваться в главе 16. Использование функции `zip`, которая представляет собой способ конструирования словарей из списков ключей и значений, будет рассматриваться в главе 13. Если заранее невозможно предугадать, какие ключи и значения будут получаться во время работы программы, вы всегда сможете создавать их динамически в виде списков и затем упаковывать функцией `zip`.

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. Здесь также применима инструкция `del` — она удаляет значение, связанное с ключом, который играет роль индекса. Обратите также внимание на

наличие вложенного списка в следующем примере (значение для ключа 'ham'). Все типы-коллекции в языке Python могут вкладываться друг в друга в произвольном порядке:

```
>>> d2['ham'] = ['grill', 'bake', 'fry'] # Изменение элемента
>>> d2
{'eggs': 3, 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
>>> del d2['eggs'] # Удаление элемента
>>> d2
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}
>>> d2['brunch'] = 'Bacon' # Добавление нового элемента
>>> d2
{'brunch': 'Bacon', 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```

Как и в случае со списками, в словаре операция присваивания по существующему индексу приводит к изменению ассоциированного с ним значения. Однако в отличие от списков, словари допускают выполнение присваивания по *новому* ключу (который ранее отсутствовал), в результате создается новый элемент словаря, как показано в предыдущем примере для ключа 'brunch'. Этот прием 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 средства 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:

```
>>> d2 = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> d2.values()
[3, 1, 2]
>>> d2.items()
[('eggs', 3), ('ham', 1), ('spam', 2)]
```

Такие списки удобно использовать в циклах, когда необходимо выполнить обход элементов словаря. Попытка извлечения несуществующего элемента словаря 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 избежать появления ошибки обращения к несуществующему ключу:

```
>>> d2.get('spam') # Ключ присутствует в словаре
2
>>> d2.get('toast') # Ключ отсутствует в словаре
None
>>> d2.get('toast', 88)
88
```

Метод `update` реализует своего рода операцию конкатенации для словарей. Он объединяет ключи и значения одного словаря с ключами и значениями другого, просто перезаписывая значения с одинаковыми ключами:

```
>>> d2
{'eggs': 3, 'ham': 1, 'spam': 2}
>>> d3 = {'toast': 4, 'muffin': 5}
>>> d2.update(d3)
>>> d2
{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}
```

Наконец, метод `pop` удаляет ключ из словаря и возвращает его значение. Он напоминает метод `pop` списков, только вместо необязательного индекса элемента принимает ключ:

```
# удаление элементов словаря по ключу

>>> d2
{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}
>>> d2.pop('muffin')
5
>>> d2.pop('toast')    # Удаляет и возвращает значение заданного ключа
4
>>> d2
{'eggs': 3, 'ham': 1, 'spam': 2}

# удаление элементов списка по 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 следующей главе, как один из способов избежать побочных эффектов, связанных с наличием нескольких ссылок на один и тот же словарь. В действительности словари обладают гораздо большим числом методов, чем перечислено в табл. 8.2. Чтобы получить 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 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.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 подробно не рассматривали. Для тех, кто не знаком с циклами `for`, замечу, что приведенная 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.

Так как словари не являются последовательностями, их нельзя напрямую использовать в операторе цикла `for`, как это допускают строки и списки. Но, если необходимо выполнить обход элементов словаря, это легко сделать с помощью метода `keys`, возвращающего список всех ключей словаря, который может использоваться для выполнения итераций в цикле `for`. В случае необходимости внутри цикла можно получать значение элемента по его ключу, как это реализовано в данном примере.

Кроме того, Python в действительности позволяет выполнять обход списка 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, упоминавшихся ранее, которая позволяет использовать оператор проверки вхождения `in` со словарями (подробнее об итераторах далее в книге).

Замечания по использованию словарей

Словари окажутся достаточно просты в использовании, когда вы освоите работу 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 сделать нечто подобное, интерпретатор выдает сообщение об ошибке.

- *Присваивание по несуществующему индексу приводит к созданию нового элемента.* Ключи можно создавать при определении словаря в виде литерала (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 качестве 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 можно 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 методы, 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 примерах. Списки не допускают возможность присваивания по индексам, находящимся за пределами списков:

```
>>> L = []
>>> L[99] = 'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range

(Перевод: Диагностика (самого последнего вызова):
  Файл "<stdin>", строка 1, позиция ?
IndexError: присваивание по индексу за пределами списка)
```

Можно, конечно, с помощью операции повторения создать список достаточно большой длины (например, `[0]*100`), но можно создать нечто, что напоминает словарь, который не требует такого выделения пространства. При использовании 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 является строка 'spam'. В такой структуре можно обращаться по смещениям, как и в списке, но вам не требуется выделять пространство для всех позиций, которые могут когда-либо потребоваться при выполнении программы. При использовании подобным образом словари представляют собой более гибкие эквиваленты списков.

Использование словарей для структур разреженных данных

Похожим образом словари могут использоваться для реализации структур *разреженных* данных, таких как многомерные массивы, где всего несколько элементов имеют определенные значения:

```
>>> Matrix = {}
>>> Matrix[(2, 3, 4)] = 88
>>> Matrix[(7, 8, 9)] = 99
>>>
>>> X = 2; Y = 3; Z = 4      # ; 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 возбуждению 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)

(Перевод: Диагностика (самого последнего вызова):
  Файл "<stdin>", строка 1, позиция ?
KeyError: (2, 3, 6))
```

Как избежать появления ошибок обращения к несуществующему ключу

Ошибки обращения к несуществующему ключу являются обычными при работе с разреженными матрицами, но едва ли кто-то захочет, чтобы они приводили к преждевременному завершению программы. Существует, по крайней мере, три способа получить значение по умолчанию вместо возбуждения исключения – можно предварительно проверить наличие ключа с помощью условного оператора `if`, воспользоваться конструкцией `try`, чтобы перехватить и явно обработать исключение, или просто использовать показанный ранее метод словаря `get`, способный возвращать значение по умолчанию для несуществующих ключей:

```

>>> if Matrix.has_key((2,3,6)): # Проверить наличие ключа перед обращением
...     print Matrix[(2,3,6)]
... 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. Вообще говоря, они способны заменить реализацию алгоритмов поиска (потому что операция индексирования по ключу уже является операцией поиска), могут представлять самые разные типы структурированной информации. 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 структурированную информацию; это особенно заметно, когда появляются уровни вложенности. 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'],
...     'web': 'www.rmi.net/~lutz',
...     'home': {'state': 'CO', 'zip': 80513}}
...
```

Чтобы извлечь компоненты вложенных объектов, достаточно просто объединить в цепочку операции индексирования:

```
>>> mel['name']
'Mark'
>>> mel['jobs']
['trainer', 'writer']
>>> mel['jobs'][1]
'writer'
>>> mel['home']['zip']
80513
```

Другие способы создания словарей

Наконец, обратите внимание, что благодаря практической ценности словарей с течением времени появились и другие способы их создания. В Python 2.3 и более поздних версиях, например, последние два вызова конструктора `dict` 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:

- Первый вариант удобен, если содержимое всего словаря известно заранее.
- Второй вариант удобно использовать, когда необходимо динамически создавать словарь по одному полю за раз.
- Третий вариант с использованием именованных аргументов даже компактнее, 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.

Как было показано в конце табл. 8.2, последний вариант также часто используется в соединении с функцией `zip`, которая позволяет объединить отдельные списки ключей и значений, создаваемые динамически во время выполнения (например, при извлечении данных из столбцов в файле).

Если значения всех ключей словаря остаются все время одними и теми же, существует специализированная форма инициализации словаря, при использовании которой достаточно просто передать список ключей и начальное значение (по умолчанию используется значение `None`):

```
>>> dict.fromkeys(['a', 'b'], 0)
{'a': 0, 'b': 0}
```

В настоящее время вы вполне можете обойтись простыми литералами и операцией присваивания по ключам, но вы наверняка найдете применение всем упомянутым вариантам создания словарей, как только приступите к созданию настоящих, гибких и динамических программ на языке Python.

Придется держать в уме: интерфейсы словарей

Помимо удобного способа хранения информации по ключам непосредственно в программе некоторые расширения для Python также предоставляют интерфейсы, которые выглядят и действуют как словари. Например, обращение к индексированным файлам данных в формате DBM во многом напоминает обращение к словарию, который сначала требуется 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
```

Позднее будет показано, как таким же способом можно сохранять целые объекты Python, достаточно лишь заменить имя модуля `anydbm` на `shelve` (`shelves` (хранилища) — это базы данных с доступом к информации по ключу, предназначенные для постоянного хранения объектов Python). Для работы 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, находящегося на клиентской веб-странице:

```
import cgi
form = cgi.FieldStorage( ) # Parse form data
if form.has_key('name'):
    showReply('Hello, ' + form['name'].value)
```

Все эти объекты (и словари в том числе) являются примерами отображений. Как только вы овладеете словарными интерфейсами, вы обнаружите, что они имеют отношение ко множеству встроенных инструментов языка Python.

В заключение

В этой главе мы исследовали списки и словари – два, пожалуй, наиболее часто 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 увеличиваться 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 различные операции непосредственного их изменения, недоступные для строк, например, списки можно увеличивать в размерах с помощью метода `append`, а словари – за счет выполнения операции присваивания по новому ключу.

В следующей главе мы закончим подробное исследование базовых типов объектов, рассмотрев кортежи и файлы. После этого мы перейдем к инструкциям, которые реализуют логику обработки этих объектов, что позволит нам еще больше приблизиться к написанию полноценных программ. Но прежде чем заняться 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 списка.
4. Назовите четыре операции, которые изменяют непосредственно объект словаря.

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.

2. Литеральное выражение, например, `{ 'a': 0, 'b': 0 }`, или последовательность операций присваивания, таких как `D = []; D['a'] = 0, D['b'] = 0`, создадут требуемый словарь. Можно также использовать более новый и более простой способ с использованием именованных аргументов `dict(a=0, b=0)` или более гибкий вариант с указанием последовательностей пар ключ/значение `dict([('a', 0), ('b', 0)])`. Или, поскольку все ключи известны заранее и не будут изменяться в дальнейшем, можно использовать специализированную форму `dict.fromkeys(['a', 'b'], 0)`.
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, методы `remove` и `pop` удаляют элементы списка по значениям и по смещению, инструкция `del` удаляет элемент списка или срез, а операции присваивания по индексу или срезу замещают элемент или целый сегмент списка. Для правильного 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 руководствам.

9

Кортежи, файлы и все остальное

Эта глава завершает наше детальное исследование базовых типов объектов языка 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, поддерживающие операции, которые по большей части вам уже знакомы по строкам и спискам. Объекты-файлы – это широко используемые многофункциональные инструменты для работы с файлами – краткий обзор файлов, что приводится 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 типов, с которыми мы познакомились, – понятия равенства, сравнения, копий объекта и т. д. Мы также кратко познакомимся и с другими типами объектов, присутствующими в арсенале Python, – несмотря на то, что мы рассмотрели все основные встроенные типы, спектр объектов в языке Python гораздо шире, чем я давал вам основания полагать к этому моменту. В заключение мы закроем эту часть книги изучением связанных с типами объектов ловушек, в которые часто попадаются программисты, и исследуем некоторые примеры, которые позволят вам поэкспериментировать с освоенными идеями.¹

¹ После детального изучения удобно пользоваться краткими справочниками, например, представленными в справочном разделе официального сайта проекта Python. Особо стоит отметить Мастерскую отца-основателя Python (Python Workshop by Guido van Rossum). Презентация доступна по адресу: <http://www.python.org/doc/essays/ppt/acm-ws/>. – Примеч. науч. ред.

Кортежи

Последний тип коллекций в нашем обзоре – это кортежи. Кортежи представляют собой простые группы объектов. Они работают точно так же, как списки, за исключением того, что не допускают непосредственного изменения (они являются неизменяемыми) и в литеральной форме записываются как последовательность элементов в круглых, а не в квадратных скобках. Кортежи не имеют методов, но 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 (т. е. они обеспечивают упорядочение своего содержимого слева направо). Подобно спискам они могут содержать объекты любого типа.

Обеспечивают доступ к элементам по смещению

Подобно строками и спискам доступ к элементам кортежей осуществляется по смещению (а не по ключу) – они поддерживают все операции, которые основаны на использовании смещения, такие как индексирование и извлечение среза.

Относятся к категории неизменяемых последовательностей

Подобно строкам кортежи являются неизменяемыми объектами – они не поддерживают никаких операций непосредственного изменения, которые применяются к спискам. Подобно строкам и спискам кортежи являются последовательностями и поддерживают многие операции над последовательностями.

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 стороны, кортежи могут хранить другие составные объекты (т. е. списки, словари и другие кортежи), а следовательно, поддерживают произвольное число уровней вложенности.

Массив ссылок на объекты

Подобно спискам кортежи проще представлять как массивы ссылок на объекты – кортежи хранят указатели (ссылки) на другие объекты, а операция 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, заключенная в круглые скобки. Пустые кортежи определяются как пара пустых круглых скобок.

Таблица 9.1. Литералы кортежей и операции

Операция	Интерпретация
()	Пустой кортеж
t1 = (0,)	Кортеж из одного элемента (не выражение)
t2 = (0, 'Ni', 1.2, 3)	Кортеж из четырех элементов
t2 = 0, 'Ni', 1.2, 3	Еще один кортеж из четырех элементов (тот же самый, что и строкой выше)
t3 = ('abc', ('def', 'ghi'))	Вложенные кортежи
t1[i]	Индекс, индекс индекса, срез, длина
t3[i][j]	
t1[i:j]	
len(t1)	
t1 + t2	Конкатенация, повторение
t2 * 3	
for x in t	Обход в цикле, проверка вхождения
'spam' in t2	

Кортежи в действии

Как обычно, запустите интерактивный сеанс работы с интерпретатором Python, чтобы приступить к исследованию кортежей в действии. Обратите внимание: в табл. 9.1 отсутствуют методы кортежей (т. е. метод `append` к кортежам неприменим). Это действительно так, зато кортежи поддерживают обычные операции над последовательностями, которые применяются 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 пояснений. Поскольку круглые скобки могут также окружать выражения (глава 5), необходимо что-то предпринять, чтобы дать интерпретатору понять, что единственный объект в круглых скобках – это кортеж, а не простое выражение. Если вам действительно необходимо получить кортеж с единственным элементом, нужно просто добавить запятую после этого элемента, перед закрывающей круглой скобкой:

```
>>> x = (40)           # Целое число
>>> x
40
>>> y = (40,)          # Кортеж, содержащий целое число
>>> y
(40,)
```

В виде исключения при определении кортежей интерпретатор позволяет опускать открывающую и закрывающую круглые скобки, если синтаксически конструкция интерпретируется однозначно. Например, в четвертой строке таблицы кортеж создается простым перечислением четырех элементов, разделенных запятыми. В контексте операции присваивания интерпретатор распознает, что это кортеж, даже при отсутствии круглых скобок.

Кто-то может посоветовать вам всегда использовать круглые скобки в кортежах, а кто-то – посоветовать никогда не использовать их (найдутся и те, кто не скажет вам, что с кортежами делать!). Единственное место, где круглые скобки являются *обязательными*, – при передаче кортежей функциям в виде литералов (где круглые скобки имеют важное значение) и при передаче их инструкции `print` (где важное значение имеют запяты).

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 удобочитаемость сценариев, делая кортежи в программном коде более заметными, но у вас может быть свое собственное мнение на этот счет.

Преобразования и неизменяемость

Хотя синтаксис литералов отличается, операции, выполняемые над кортежами (последние три строки в табл. 9.1), идентичны операциям, применяемым к строкам и спискам. Единственное отличие состоит в том, что операции `+`, `*` и извлечения среза при применении к кортежам возвращают новые *кортежи*, а также в том, что в отличие от строк, списков и словарей, кортежи не имеют методов. Если, к примеру, необходимо отсортировать содержимое кортежа, его сначала следует преобразовать в список, чтобы превратить его в изменяемый объект и получить доступ к методу сортировки:

```
>>> T = ('cc', 'aa', 'dd', 'bb')
>>> tmp = list(T)           # Создается список из элементов кортежа
>>> tmp.sort()              # Сортировка списка
>>> tmp
['aa', 'bb', 'cc', 'dd']
>>> T = tuple(tmp)          # Создается кортеж из элементов списка
>>> T
('aa', 'bb', 'cc', 'dd')
```

Здесь `list` и `tuple` – это встроенные функции, которые используются для преобразования 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 списков (`list comprehensions`). Например, ниже из кортежа создается список, причем попутно к каждому элементу прибавляется число 20:

```
>>> 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 последовательностями – они всегда создают новые списки, но они могут использоваться для обхода содержимого любых объектов последовательностей, включая кортежи, строки и другие списки. Как будет показано дальше, они могут применяться даже к программным компонентам, которые физически не являются последовательностями, – к любым объектам, поддерживающим возможность выполнения итераций, включая файлы, которые автоматически читаются строка за строкой.

Кроме того, следует заметить, что правило неизменяемости кортежа применимо 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
(Перевод: TypeError: объект не поддерживает операцию присваивания элементам)

>>> T[1][0] = 'spam'   # Допустимо: вложенный изменяемый объект можно изменить
>>> T
(1, ['spam', 3], 4)
```

Для большинства программ такой одноуровневой неизменяемости для обычного использования кортежей вполне достаточно. О чем, совершенно случайно, и рассказывается в следующем разделе.

Для чего нужны кортежи, когда уже имеются списки?

Похоже, что это самый первый вопрос, который задают начинающие программисты, узнав о кортежах: «Зачем нам нужны кортежи, когда у нас уже имеются списки?». Некоторые из причин корнями уходят в прошлое – создатель языка Python является математиком по образованию, который видел кортежи как простые ассоциации объектов, а списки – как структуры данных, допускающие изменения в течение всего времени своего существования.

Однако более правильным будет считать, что неизменяемость кортежей обеспечивает своего рода *поддержку целостности*, – вы можете

быть уверены, что кортеж не будет изменен посредством другой ссылки из другого места в программе, чего нельзя сказать о списках. Тем самым кортежи играют роль объявлений «констант», присутствующих в других языках программирования, несмотря на то, что в языке Python это понятие связано с объектами, а не с переменными.

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 коллекции, которые может потребоваться изменить. Кортежи могут использоваться и в остальных случаях, когда необходимы фиксированные ассоциации объектов.

Файлы

Возможно, вы уже знакомы с понятием файла – так называются области постоянной памяти в вашем компьютере, которыми управляет операционная 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 обеспечивает связь с файлом, размещенным в компьютере. После вызова функции `open` можно выполнять операции чтения и записи во внешний файл, используя методы полученного объекта. Встроенное имя `file` является синонимом имени `open`, поэтому файлы могут открываться обращением к функции как по имени `open`, так и по имени `file`. Обычно для выполнения операции открытия файла предпочтительнее использовать функцию `open`, тогда как имя `file` в основном предназначено для обеспечения соответствия требованиям объектно-ориентированного программирования (описывается далее в этой книге).

По сравнению с типами, с которыми вы уже знакомы, объекты файлов выглядят несколько необычно. Они не являются ни числами, ни последовательностями 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 т. д.

Открытие файлов

В табл. 9.2 приводятся наиболее часто используемые операции над файлами. Чтобы открыть файл, программа должна вызвать функцию `open`, передав ей имя внешнего файла и режим работы. Обычно в качестве режима используется строка `'r'`, когда файл открывается для чтения

(по умолчанию), 'w' – когда файл открывается для записи, или 'a' – когда файл открывается на запись в конец. Существуют и другие возможности, но мы не будем упоминать о них здесь. В некоторых случаях в конец строки режима может добавляться символ b, что позволяет работать с двоичными данными (интерпретация символов новой строки отключена), а добавление символов a+ означает, что файл открывается как для чтения, так и для записи (т. е. из файла можно читать и записывать в него с помощью одного и того же объекта).

Таблица 9.2. Часто используемые операции над файлами

Операция	Интерпретация
<code>output = open('/tmp/spam', 'w')</code>	Открывает файл для записи ('w' означает write – запись)
<code>input = open('data', 'r')</code>	Открывает файл для чтения ('r' означает read – чтение)
<code>input = open('data')</code>	То же самое, что и в предыдущей строке (режим 'r' используется по умолчанию)
<code>aString = input.read()</code>	Чтение файла целиком в единственную строку
<code>aString = input.read(N)</code>	Чтение следующих N байтов (одного или более) в строку
<code>aString = input.readline()</code>	Чтение следующей текстовой строки (включая символ конца строки) в строку
<code>aList = input.readlines()</code>	Чтение файла целиком в список строк
<code>output.write(aString)</code>	Запись строки в файл
<code>output.writelines(aList)</code>	Запись всех строк из списка в файл
<code>output.close()</code>	Заккрытие файла вручную (выполняется по окончании работы с файлом)
<code>output.flush()</code>	Выталкивает выходные буферы на диск, файл остается открытым
<code>anyFile.seek(N)</code>	Изменяет текущую позицию в файле для следующей операции, смещая ее на N байтов от начала файла.

Оба аргумента функции `open` должны быть строками. Кроме того, может использоваться третий необязательный аргумент, управляющий буферизацией выводимых данных, – значение ноль означает, что выходная информация не будет буферизироваться (т. е. она будет записываться во внешний файл сразу же, в момент вызова метода записи). Имя внешнего файла может включать платформозависимые префиксы абсолютного или относительного пути к файлу. Если путь к файлу не указан, предполагается, что он находится в текущем рабочем каталоге (т. е. в каталоге, где был запущен сценарий).

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 возвращает текст в строках, а 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.

Несмотря на то, что методы чтения и записи, перечисленные в таблице, являются наиболее часто используемыми, имейте в виду, что самый лучший, пожалуй, способ чтения строк из файла на сегодняшний день состоит в том, чтобы не читать файл вообще – как будет показано в главе 13, файлы имеют *итератор* (*iterator*), который автоматически читает информацию из файла строку за строкой в контексте цикла `for`, в генераторах списков и в других итерационных контекстах.

Обратите внимание: 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 выполнении операции записи данных в файл, в отличие от инструкции `print`, интерпретатор Python не выполняет автоматическое преобразование объектов 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` и выражения форматирования строк). Кроме того, в состав Python входят дополнительные 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` разрывает связь с внешним файлом. Как рассказывалось в главе 6, интерпретатор Python немедленно освобождает память, занятую объектом, как только в программе будет утеряна последняя ссылка на этот объект. Как только объект файла освобождается, интерпретатор автоматически закрывает ассоциированный с ним файл. Благодаря этому вам не требуется 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 является частью спецификации языка, и с течением времени такое поведение может измениться. Следовательно, привычку вызывать метод `close` вручную можно только приветствовать.

Файлы в действии

Давайте рассмотрим небольшие примеры, демонстрирующие основы работы с файлами. В первом примере выполняется открытие нового файла в режиме для записи, в него записывается строка (завершающаяся символом новой строки `\n`), после чего файл закрывается. Далее этот же файл открывается в режиме для чтения и выполняется чтение строки из него. Обратите внимание, что второй вызов метода `readline` возвращает пустую строку – таким способом методы файла в языке Python сообщают о том, что был достигнут конец файла (пустая строка в файле возвращается как строка, содержащая единственный символ новой строки, а не как действительно пустая строка). Ниже приводится полный листинг сеанса:

```
>>> myfile = open('myfile', 'w')      # Открывает файл для записи (создает файл)
>>> myfile.write('hello text file\n')  # Записывает строку текста
>>> myfile.close()                    # Выталкивает выходные буферы на диск
>>> myfile = open('myfile')            # Открывает файл для чтения: 'r' – по умолчанию
>>> myfile.readline()                  # Читает строку
'hello text file\n'
>>> myfile.readline()                  # Пустая строка: конец файла
..
```

Этот пример записывает единственную строку текста в файл, добавляя в нее символ конца строки `\n` – методы записи не добавляют символ конца строки, поэтому нам необходимо добавлять его в выводимые строки (в противном случае следующая операция записи дополнит текущую строку в файле).

Сохранение и интерпретация объектов Python в файлах

Теперь создадим большой файл. Следующий пример записывает различные объекты в текстовый файл. Обратите внимание на использование средств преобразования объектов в строки. Напомню, что данные всегда записываются в файл в виде строк, а методы записи не выполняют автоматического преобразования данных:

```
>>> 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 автоматического вывода в интерактивной оболочке дает точное побайтовое представление содержимого, а инструкция `print`

интерпретирует встроенные символы конца строки, чтобы обеспечить более удобочитаемое отображение:

```
>>> bytes = open('datafile.txt').read() # Отображение bytes
>>> bytes                               # в неформатированном виде
"Spam\n43,44,45\n[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> print bytes                          # Удобочитаемое представление
Spam
43,44,45
[1, 2, 3]${'a': 1, 'b': 2}
```

Теперь нам необходимо выполнить обратные преобразования, чтобы получить действительные объекты языка Python из строк в текстовом файле. Интерпретатор Python никогда автоматически не выполняет преобразование строк в числа или в объекты других типов, поэтому нам необходимо выполнить соответствующие преобразования, чтобы можно было использовать операции над этими объектами, такие как индексирование, сложение и т. д.:

```
>>> F = open('datafile.txt') # Открыть файл снова
>>> line = F.readline()      # Прочитать одну строку
>>> line
'Spam\n'
>>> line.rstrip()           # Удалить символ конца строки
'Spam'
```

Для этой первой строки был использован метод `rstrip`, чтобы удалить завершающий символ конца строки. Тот же эффект можно было бы получить с помощью извлечения среза `line[:-1]`, но такой подход можно использовать, только если мы уверены, 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 прочитаем следующий блок, в котором содержатся числа, и выполним разбор этого блока (т. е. извлечем объекты):

```
>>> line = F.readline()      # Следующая строка из файла
>>> line                      # Это - строка
'43,44,45\n'
>>> parts = line.split(',')   # Разбить на подстроки по запятым
>>> parts
['43', '44', '45\n']
```

Здесь был использован строковый метод `split`, чтобы разбить строку на части по запятым, которые играют роль символов-разделителей – в результате мы получили список строк, каждая из которых содержит отдельное число. Теперь нам необходимо 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 числа, а генератор списков, представленный в главе 4, позволяет применить функцию ко всем элементам списка в одной инструкции (подробнее о генераторах списков читайте далее в этой книге). Обратите внимание: для удаления завершающего символа `\n` в конце последней подстроки не был использован метод `rstrip`, потому что `int` и некоторые другие функции преобразования просто игнорируют символы-разделители, окружающие цифры.

Для преобразования списка и словаря в третьей строке файла воспользуемся встроенной функцией `eval`, интерпретирующей строку как код на Python (формально строка содержит выражение 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 строку, 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 *слишком* мощный. Функция `eval` без лишних вопросов выполнит любое выражение на языке Python, даже если в результате будут удалены все файлы в компьютере, если передать в выражение соответствующие права доступа! Если вам действительно 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 преобразования. Он напоминает суперуниверсальную утилиту форматирования и преобразования данных. Чтобы сохранить словарь в файле, например, мы передаем его непосредственно в функцию модуля `pickle`:

```
>>> F = open('datafile.txt', 'w')
>>> import pickle
```

```
>>> pickle.dump(D, F)           # Модуль pickle запишет в файл любой объект
>>> F.close()
```

Чтобы потом прочитать словарь обратно, можно просто еще раз воспользоваться возможностями модуля `pickle`:

```
>>> F = open('datafile.txt')
>>> 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 и из строки байтов, не требуя от нас почти никаких действий. В действительности, 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.txt').read()
"(dp0\nS'a'\np1\nI1\nsS'b'\np2\nI2\ns."
```

Поскольку модуль `pickle` умеет реконструировать объекты из строкового представления, нам не требуется самим возиться с этим. Дополнительную информацию 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 использует модуль `pickle` для сохранения объектов Python в файлах с доступом по ключу, описание которых далеко выходит за рамки этой книги.

Сохранение и интерпретация упакованных двоичных данных в файлах

Прежде чем двинуться дальше, необходимо рассмотреть еще один аспект работы с файлами: в некоторых приложениях приходится иметь дело с упакованными двоичными данными, которые создаются, например, программами на языке C. Для их сохранения и восстановления можно воспользоваться модулем `struct` из стандартной библиотеки языка Python. В некотором смысле это совершенно другой инструмент преобразования данных, интерпретирующий строки в файлах как двоичные данные.

Например, чтобы создать файл с упакованными двоичными данными, откройте его в режиме `'wb'` (write binary – запись двоичных данных) и передайте модулю `struct` строку форматирования и некоторый объект Python. В следующем примере используется строка форматирования, которая определяет пакет данных как содержащий 4-байтовое целое число, 4-символьную строку и 2-байтовое целое число, причем все представлены в формате `big-endian` – в порядке следования байтов «от стар-

шего к младшему» (существуют также спецификаторы форматов, которые поддерживают наличие символов дополнения слева, числа с плавающей точкой и многие другие):

```
>>> F = open('data.bin', 'wb') # Открыть файл для записи в двоичном режиме
>>> import struct
>>> bytes = struct.pack('>i4sh', 7, 'spam', 8) # Создать пакет двоичных данных
>>> bytes
'\x00\x00\x00\x07spam\x00\x08'
>>> F.write(bytes) # Записать строку байтов
>>> F.close()
```

Интерпретатор Python создает строку двоичных данных, которая затем записывается в файл обычным способом (строка состоит из экранированных последовательностей, представляющих шестнадцатеричные значения). Чтобы преобразовать эти значения в обычные объекты языка Python, достаточно просто 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
'\x00\x00\x00\x07spam\x00\x08'
>>> 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 информацией обращайтесь к руководству по библиотеке языка Python или импортируйте модуль `struct` в интерактивном сеансе и передайте имя `struct` функции `help`. Обратите также внимание на то, что режимы доступа к двоичным файлам `'wb'` и `'rb'` могут использоваться для обработки простейших двоичных файлов, таких как изображения или аудиофайлы, без необходимости выполнять распаковку их содержимого.

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 отсутствуют в таблице. Например, функция `seek` переустанавливает текущую позицию в файле (которая будет использована в следующей операции чтения или записи), функция `flush` принудительно выталкивает содержимое выходных буферов на диск (по умолчанию файлы всегда буферизуются) и т. д.

Руководство по стандартной библиотеке и книги, упомянутые в предисловии, содержат полный перечень методов для работы с файлами. Чтобы кратко ознакомиться с ним, можно также воспользоваться функцией `dir` или `help` в интерактивном сеансе, передав ей слово `file`

(имя типа объектов). Дополнительные примеры работы с файлами вы найдете во врезке «Придется держать в уме: сканирование файлов» в главе 13. В ней приводятся типичные примеры организации циклов для просмотра содержимого файлов и приводятся инструкции, которые мы еще не рассматривали.

Следует также отметить, что функция `open` и объекты файлов, которые она возвращает, являются в языке Python основным интерфейсом к внешним файлам, однако в арсенале Python существуют и другие инструменты, напоминающие файлы. Кроме того, имеется возможность взаимодействовать с дескрипторами файлов (целочисленные значения, которые поддерживаются такими низкоуровневыми средствами, как механизм блокировки файлов), реализованная в модуле `os`; сокетами, каналами и файлами `FIFO` (объекты, по своим характеристикам напоминающие файлы, используемые для синхронизации процессов или организации взаимодействий по сети); с файлами с доступом по ключу, известными как «хранилища» («shelves») и используемыми для хранения объектов языка Python по ключу; и многими другими. За получением дополнительной информации обо всех этих инструментах обращайтесь к специализированной литературе по языку Python.

Пересмотренный перечень категорий типов

Теперь, когда мы познакомились со всеми встроенными базовыми типами языка Python в действии, давайте завершим наше турне рассмотрением некоторых общих для них свойств.

В табл. 9.3 приводится классификация всех рассмотренных типов по категориям, которые были введены ранее. Над объектами одной категории могут выполняться одни и те же операции, например, над строками, списками и кортежами можно выполнять операции для последовательностей. Непосредственное изменение допускают только изменяемые объекты (списки и словари) – вы не сможете непосредственно изменить числа, строки и кортежи. Файлы только экспортируют 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 типов в языке Python.

Таблица 9.3. Классификация объектов

Тип объектов	Категория	Изменяемый?
Числа	Числа	Нет
Строки	Последовательности	Нет
Списки	Последовательности	Да
Словари	Отображения	Да
Кортежи	Последовательности	Нет
Файлы	Расширения	Понятие неприменимо

Придется держать в уме: перегрузка операторов

Позднее мы увидим, что объекты, реализованные с помощью классов, могут свободно попадать в любую из этих категорий. Например, если нам потребуется реализовать новый тип объектов последовательностей, 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 т. д. Точно так же можно было бы создать новый изменяемый или неизменяемый объект, выборочно реализовав методы, вызываемые для выполнения операций непосредственного изменения (например, для выполнения операций присваивания `self[index] = value` вызывается метод `__setitem__`). Существует также возможность выполнять реализацию новых объектов на других языках программирования, таких как С, в виде расширений. Чтобы определить выбор между множествами операций над числами, последовательностями и отображениями, необходимо будет подставить указатели на функции С 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 т. д. Вложенность может быть произвольной глубины, какая только потребуется, чтобы смоделировать структуру обрабатываемых данных.

Рассмотрим пример с вложенными компонентами. В следующем примере определяется три вложенных составных объекта последовательностей, как показано на рис. 9.1. Для обращения к отдельным элементам допускается использовать столько индексов, сколько потребуется. В языке Python разыменование индексов производится слева направо

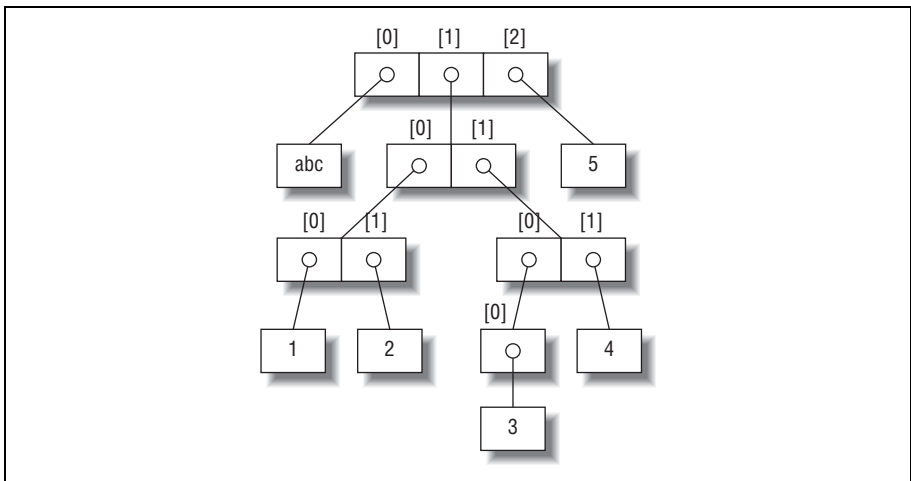


Рис. 9.1. Дерево вложенных объектов со смещениями их компонентов, созданное с помощью литерального выражения `['abc', [(1, 2), ([3], 4)], 5]`. Синтаксически вложенные объекты внутри реализованы в виде ссылок (т. е. в виде указателей) на отдельные участки памяти

и на каждом шаге извлекается объект на более глубоком уровне. Структура данных на рис. 9.1 может показаться излишне 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
```

Ссылки и копии

В главе 6 говорилось, что при выполнении операции присваивания всегда сохраняется ссылка на объект, а не копия самого объекта. В большинстве случаев именно это нам и требуется. Однако в процессе выполнения операций присваиваний может быть создано несколько ссылок на один и тот же объект, поэтому очень важно понимать, что изменения, произведенные в изменяемом объекте с помощью одной ссылки, отразятся и на других ссылках, указывающих на этот же объект. Если такой порядок вещей является нежелательным, вам 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 приме-

ре создается список, которому ставится в соответствие переменная *X*, затем создается другой список, которому ставится в соответствие переменная *L*, который включает ссылку на список *X*. 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 существует три ссылки на список, который был создан первым: из переменной *X*, из элемента списка, соответствующего переменной *L*, и из элемента словаря, соответствующего переменной *D*. Эта ситуация показана на рис. 9.2.

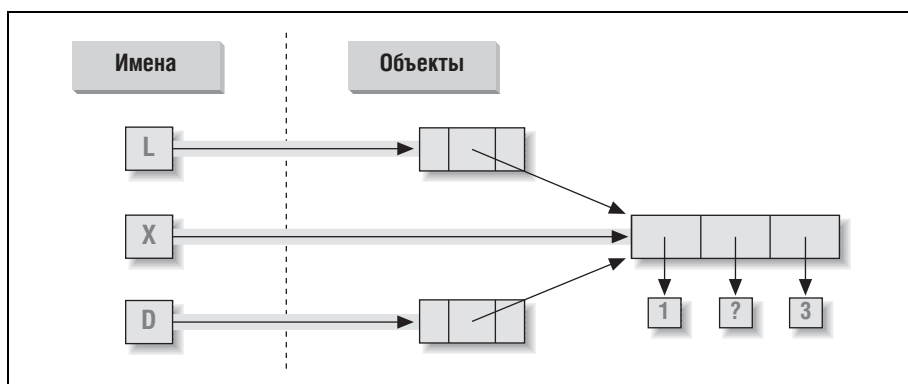


Рис. 9.2. Разделяемые ссылки на объект: поскольку переменная *X*, элемент списка *L* и элемент словаря *D* ссылаются на один и тот же список, то изменения, произведенные с помощью *X*, отразятся также на *L* и *D*

Так как списки относятся к категории изменяемых объектов, изменения в объекте 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 программирования. 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 т. д.). Это полезная особенность – вы можете передавать крупные объекты между компонентами программы без выполнения дорогостоящей операции копирования. Однако, когда действительно необходимо создать копию объекта, вы можете запросить их:

- Выражение извлечения среза с пустыми пределами (`L[:]`) создает копию последовательности.
- Метод словаря `copy` (`D.copy()`) создает копию словаря.
- Некоторые встроенные функции, такие как `list(list(L))`, создают копию списка.
- Модуль `copy`, входящий в состав стандартной библиотеки, создает полные копии объектов.

Например, предположим, что у нас имеются список и словарь, и для нас нежелательно изменение их значений посредством других переменных:

```
>>> L = [1, 2, 3]
>>> D = {'a': 1, 'b': 2}
```

Чтобы предотвратить такую возможность, достаточно связать с другими переменными копии объектов вместо того, чтобы связать с ними сами объекты:

```
>>> A = L[:]          # Вместо A = L (или list(L))
>>> B = D.copy()      # Вместо B = D
```

В этом случае изменения, произведенные с помощью других переменных, повлияют 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, используя 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 переменной `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 значениями пределов и метод словаря `copy` создают *поверхностные* копии — т. е. они не копируют вложенные структуры данных, даже если таковые присутствуют. Если необходима полная копия структуры произвольной глубины вложенности, следует использовать стандартный модуль `copy`: добавьте инструкцию `import copy` и вставьте выражение `X = copy.deepcopy(Y)`, которое создаст полную копию объекта

У 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 такие способы копирования, как операция извлечения среза и метод `copy`.

Сравнения, равенство и истина

Любые объекты языка Python поддерживают операции сравнения: проверка на равенство, относительная величина и т. д. Операции сравнения в языке Python всегда проверяют все части составных объектов, пока результат не станет определенным. В действительности, для вложенных объектов интерпретатор Python всегда автоматически выполняет обход структуры 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 компоненты:

```
>>> L1 = [1, ('a', 3)] # Разные объекты с одинаковыми значениями
>>> L2 = [1, ('a', 3)]
>>> L1 == L2, L1 is L2 # Эквивалентны? Это один и тот же объект?
(True, False)
```

В данном случае `L1` и `L2` ссылаются на совершенно разные списки, но с одинаковым содержимым. Из-за природы ссылок в языке Python 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 (т. е. расположены ли они по одному и тому же адресу в памяти).

В предыдущем примере `L1` и `L2` прошли проверку `==` на равенство (они имеют одинаковое значение, потому что все их компоненты равны), но не прошли проверку `is` на эквивалентность (они ссылаются на разные объекты и, соответственно, на разные области памяти). Однако, обратите внимание, что происходит при сравнении двух коротких строк:

```
>>> S1 = 'spam'
>>> S2 = 'spam'
>>> S1 == S2, S1 is S2
(True, True)
```

Здесь у нас также присутствуют два различных объекта, имеющие одинаковые значения: оператор `==` должна вернуть истину, а оператор

`is` – ложь. Но так как интерпретатор с целью оптимизации кэширует и повторно использует короткие строки, в действительности обе они являются одной и той же строкой `'spam'` в памяти, которая совместно используется переменными `S1` и `S2`, поэтому оператор `is` проверки идентичности возвращает истину. Чтобы получить нормальное поведение, потребуется использовать 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 механизма кэширования объектов не оказывает отрицательного влияния на ваш программный код – строки нельзя изменить непосредственно независимо от того, сколько переменных ссылаются на них. Если результат проверки на идентичность 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 рассмотрим случаи использования этих операторов далее в книге.

Операторы отношений к вложенным структурам также применяются рекурсивно:

```
>>> 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. Результат 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 направо.
- Словари сравниваются как отсортированные списки (ключ, значение).

Вообще сравнение 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 могут изменять способ сравнения.

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 формально описать смысл этих названий.

В языке Python, как в большинстве языков программирования, «ложь» представлена целочисленным значением 0, а «истина» – целочисленным значением 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.

В табл. 9.4 приводятся примеры истинных и ложных значений объектов в языке Python.

Таблица 9.4. Примеры значений истинности объектов

Объект	Значение
"spam"	True
""	False
[]	False
{}	False
1	True
0.0	False
None	False

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 указатель NULL в языке C.

Например, вспомните, что списки не допускают присваивание значений отсутствующим элементам (списки не вырастают по волшебству, когда вы выполняете присваивание некоторого значения несуществующему элементу). Чтобы можно было выполнять присваивание любому из 100 элементов списка, обычно создается список из 100 элементов, который заполняется объектами None:

```
>>> L = [None] * 100
>>>
>>> L
[None, None, None, None, None, None, None, ... ]
```

Логический тип `bool` в языке Python (представленный в главе 5) просто усиливает понятия «истина» и «ложь» в языке Python. Как вы могли заметить, результаты операций проверок в этой главе выводятся в виде значений `True` и `False`. Как мы узнали в главе 5, эти значения являются всего лишь разновидностями целых чисел 1 и 0. Этот тип реализован так, что он действительно является лишь незначительным расширением к понятиям «истина» 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 истинность они интерпретируются как «истина» и «ложь», которые в действительности 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 необязательно использовать только логические типы в логических инструкциях, таких как `if`, — любые объекты по своей природе могут быть истинными или ложными, и все логические концепции, упоминаемые в этой главе, работают, как описано, и с другими типами. 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 Python, и отображены взаимоотношения между ними. Наиболее значимые из них мы уже рассмотрели; большая часть других видов объектов на рис. 9.3 соответствует элементам программ (таким как функции и модули) или отражает внутреннее устройство интерпретатора (такие как кадры стека и скомпилированный программный код).

Главное, что следует учитывать, — в языке Python все элементы являются объектами и могут быть использованы в ваших программах. Например, можно передать класс в функцию, присвоить его переменной, заполнить им список или словарь и т. д.

Фактически даже сами типы в языке Python являются видом объектов: вызов встроенной функции `type(X)` возвращает тип объекта `X`. Объекты типов могут использоваться для сравнения типов вручную в инструкции `if`. Однако по причинам, изложенным в главе 4, ручная проверка типа в языке Python обычно рассматривается как нечто неправильное.

Одно замечание по поводу имен типов: в версии Python 2.2 у каждого базового типа появилось новое встроенное имя, добавленное для обеспечения поддержки настройки типов через объектно-ориентированный механизм наследования классов: `dict`, `list`, `str`, `tuple`, `int`, `long`, `float`, `complex`, `unicode`, `type` и `file` (`file` — это синоним 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.

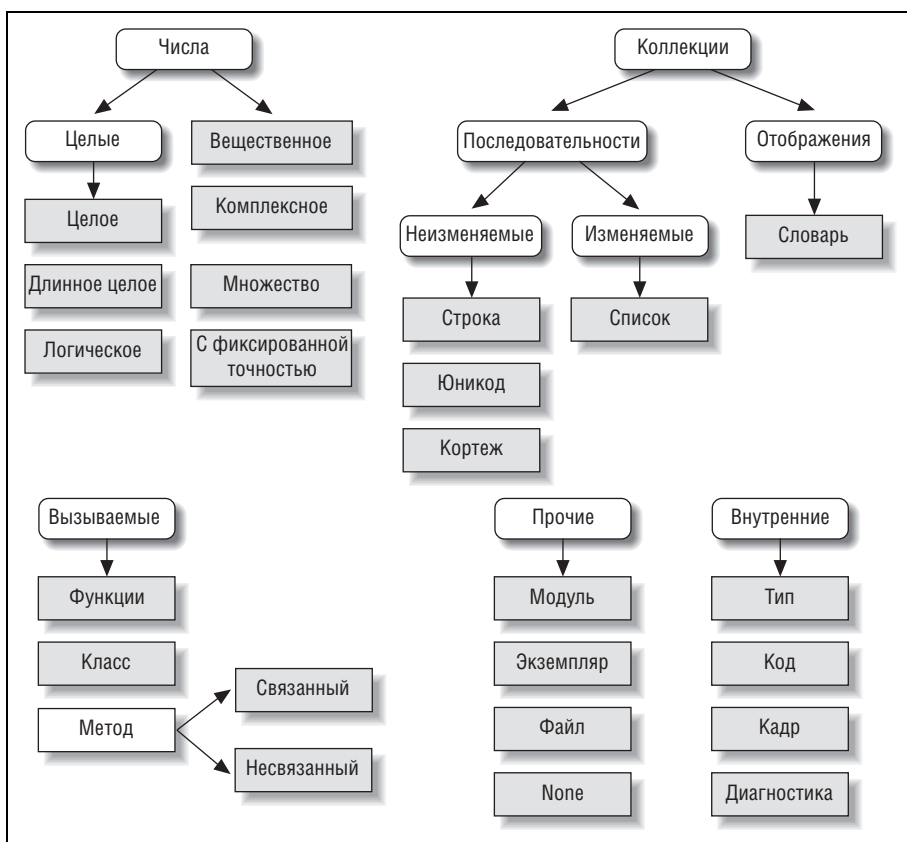


Рис. 9.3. Основные встроенные типы в языке Python, организованные по категориям. Все в языке Python является объектом, даже сами типы – это объекты! Тип любого объекта – это объект типа «type»

Модуль `types`, входящий в состав стандартной библиотеки, также предоставляет дополнительные имена типов (по большей части это синонимы встроенных типов) и предоставляет возможность проверять тип объекта с помощью функции `isinstance`. Например, в Python 2.2 и выше все следующие операции проверки возвращают истину:

```

isinstance([1],list)
type([1])==list
type([1])==type([])
type([1])==types.ListType

```

Поскольку в современных версиях Python от типов можно порождать дочерние классы, рекомендуется не пренебрегать функцией `isinstance`. Подробнее о создании дочерних классов от встроенных типов в версии Python 2.2 рассказывается в главе 26.

Другие типы в Python

Помимо базовых объектов, рассмотренных в этой части книги, в составе Python обычно устанавливаются десятки других типов объектов, доступных в виде связанных расширений на языке C или в виде классов языка Python – объекты регулярных выражений, файлы DBM, компоненты графического интерфейса, сетевые сокеты и т. д.

Главное отличие между этими и встроенными типами, которые мы до сих пор рассматривали, состоит в том, что для встроенных типов языком предоставляется специальный синтаксис создания объектов этих типов (например, 4 – для целого числа, [1, 2] – для списка и функция `open` – для файла). Другие типы обычно доступны в виде модулей стандартной библиотеки, которые необходимо импортировать перед использованием. Например, чтобы создать объект регулярного выражения, необходимо импортировать модуль `re` и вызвать `re.compile()`. Полное руководство по всем типам, доступным 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 пользователи (а иногда и эксперты), и их решений. Отчасти – это краткий обзор идей, которые мы уже рассматривали, но они достаточно важны, чтобы вновь вернуться к ним.

Операция присваивания создает ссылку, а не копию

Поскольку это центральное понятие, я напомним о нем еще раз: вы должны понимать, что может происходить с разделяемыми ссылками в вашей программе. Например, ниже создается объект списка, связанный с именем `L`, на это имя также ссылается элемент списка `M`. Таким образом, изменение списка 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 создать явно копию объекта. Для списков всегда можно создать поверх-

ностную копию с помощью операции извлечения среза с незадаанными 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 последовательности, откуда извлекается срез. Если оба значения опущены, в результате операции будут извлечены все элементы последовательности, что создаст поверхностную копию (новый, не разделяемый ни с кем объект).

Операция повторения добавляет один уровень вложенности

Операция повторения последовательности добавляет последовательность саму к себе заданное число раз. Это правда, но когда появляются вложенные изменяемые 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 повторенный 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 ссылки на изменяемый объект. Если вы помните, операции повторения, конкатенации 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 упражнении: если объект-коллекция содержит ссылку на себя, он называется *циклическим 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 циклическую ссылку в объекте, это случай заслуживает внимания, т. к. он может привести к сбоям — циклические структуры могут вызывать неожиданное заикливание программного кода, если вы не все предусмотрели. Например, 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 главе 3; кроме того, решение проблемы приводится также в конце главы 21, 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 т. д., 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 проблем, т. к. они не могут изменяться непосредственно и не подвержены, как списки, побочным эффектам такого рода.

В заключение

В этой главе были исследованы последние два базовых типа объектов – кортежи и файлы. Мы узнали, что кортежи поддерживают все операции, обычные для последовательностей, но не имеют методов и не позволяют выполнять изменения непосредственно в объекте, потому что они относятся к категории неизменяемых объектов. Мы также узнали, что объекты-файлы возвращаются функцией `open` и предоставляют методы чтения и записи данных. Мы исследовали 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 объектов и работа с двоичными данными). В заключение мы рассмотрели некоторые свойства, общие для всех типов объектов (например, разделяемые ссылки) и прошлись по списку часто встречающихся ошибок (ловушек), связанных с типами объектов.

В следующей части мы обратимся к теме синтаксиса операторов в языке Python – здесь мы исследуем все основные процедурные операторы. Следующая глава открывает эту часть книги с введения в общую синтаксическую модель языка Python, которая применима ко всем типам операторов. Однако, прежде чем двинуться дальше, ознакомьтесь с контрольными вопросами к главе, а затем проработайте упражнения к этой части, чтобы коротко повторить основные понятия. Операторы в основном всего лишь создают и обрабатывают объекты, поэтому, прежде чем продолжать чтение, вам необходимо проверить владение этой темой, выполнив упражнения.

Закрепление пройденного

Контрольные вопросы

1. Как определить размер кортежа?
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 элементов) любого контейнерного объекта, включая и кортежи. Это – встроенная функция, а не метод и потому может применяться к самым разным типам объектов.
2. Поскольку кортежи являются неизменяемыми, в действительности их нельзя изменить непосредственно, но можно создать новый кортеж с желаемым 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 элемента обязательно должна присутствовать завершающая запятая.) Также можно было бы преобразовать кортеж в список, выполнить необходимое изменение непосредственно в списке и произвести обратное преобразование в кортеж, но это более дорогостоящая последовательность действий, которая редко используется на практике; просто используйте списки, если заранее известно, что может потребоваться изменить объект непосредственно.
3. Аргумент режима открытия файла в функции `open` по умолчанию имеет значение `'r'`, т. е. файл открывается для чтения. Чтобы открыть текстовый файл для чтения, достаточно передать функции одно только имя файла.
4. Для сохранения объектов Python в файле можно воспользоваться модулем `pickle`, что устранил необходимость явного преобразования объектов в строки. Модуль `struct` позволяет выполнять похожие действия, но в предположении, что данные хранятся в файле в упакованном двоичном формате.
5. Чтобы скопировать все вложенные части структуры `X`, можно импортировать 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 поверхностную 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 соответственно.
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 пройти по основам встроенных объектов. Как и прежде, вам попутно могут встретиться новые понятия, поэтому обязательно сверьтесь с ответами в приложении В, ко-

гда закончите (и даже если еще не закончили). Если у вас не так много свободного времени, я рекомендую начать с упражнений 10 и 11 (так как они наиболее практичные), а затем, когда появится время, пройти все упражнения – от первого до последнего. Во всех упражнениях необходимо 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 части. Для начала запустите интерактивный сеанс работы с интерпретатором Python, введите все нижеследующие выражения и попробуйте объяснить происходящее:

```
2 ** 16
2 / 5, 2 / 5.0

"spam" + "eggs"
S = "ham"
"eggs " + S
S * 5
S[:0]
"green %s and %s" % ("eggs", S)

('x',)[0]
('x', 'y')[1]

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
D.keys(), D.values(), D.has_key((1, 2, 3))

[[]], [""], [], ( ), {}, 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:
 - а. Что произойдет, если попытаться получить доступ к элементу, индекс которого выходит за пределы списка (например, `L[4]`)?
 - б. Что произойдет, если попытаться извлечь срез, выходящий за пределы списка (например, `L[-1000:100]`)?
 - в. Наконец, как отреагирует интерпретатор на попытку извлечь последовательность в обратном порядке, когда нижняя граница больше верхней (например, `L[3:1]`)? Подсказка: попробуйте

выполнить операцию присваивания 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 список `L` с четырьмя элементами и присвойте одному из элементов пустой список (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, атрибутом или именем. Используйте ее для удаления элемента вашего списка (например, `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 после выполнения этой последовательности действий?

5. *Ключи словарей*. Рассмотрите следующий фрагмент:

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
```

Вы знаете, что словари не поддерживают доступ к элементам по смещениям; попробуйте объяснить происходящее здесь. Может быть, следующий пример прояснит ситуацию? (Подсказка: к какой категории типов относятся 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 элементу 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 списков при использовании индексов, выходящих за их пределы? Не напоминает ли вам такое поведение правила, применяемые к переменным?
7. *Общие операции*. Получите в интерактивной оболочке ответы на следующие вопросы:

- a. Что произойдет, если попытаться использовать оператор `+` с операндами различных типов (например, строка+список, список+кортеж)?
 - b. Будет ли работать оператор `+`, когда один из операндов является словарем?
 - c. Будет ли работать метод `append` со списками и со строками? Можно ли использовать метод `keys` со списками? (Подсказка: что предполагает метод `append` о заданном объекте?)
 - d. Наконец, какой тип объекта будет получен, когда операция конкатенации 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 односимвольными 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 строку 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, должность, адрес, электронный адрес и номер телефона. При построении структуры вы можете использовать любые комбинации встроенных объектов (списки, кортежи, словари, строки, числа). Затем попробуйте обратиться 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 новый файл с именем `myfile.txt`, и запишите 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 сценарии 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 записи в файлы не добавляют символ новой строки к записываемым строкам. Добавьте символ `\n` явно в конец вашей строки, если хотите получить в файле полностью завершенную строку.
12. **Функция `dir`.** Попробуйте ввести следующие выражения в интерактивной оболочке. Начиная с версии 1.5, функция `dir` была обобщена

так, что она выводит все атрибуты любого объекта, который вас может заинтересовать. Если вы пользуетесь более ранней версией интерпретатора, тот же эффект можно получить с использованием метода `__methods__`. Если вы пользуетесь интерпретатором Python 2.2, скорее всего функция `dir` окажется единственным работающим инструментом.

```
[].__methods__ _ # 1.4 или 1.5
dir([])          # 1.5 и более поздние версии

{}.__methods__ _ # Словарь
dir({})
```


III

Инструкции и синтаксис

10

Введение в инструкции языка Python

Теперь, когда вы познакомились с базовыми встроенными типами объектов языка Python, мы начинаем исследование фундаментальных форм инструкций. Как и в предыдущей части книги, мы начнем с общего представления синтаксиса инструкций и затем, в нескольких следующих главах, более подробно рассмотрим конкретные инструкции.

Выражаясь простым языком, *инструкции* – это то, что вы пишете, чтобы сообщить 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 инструкций; комбинируя инструкции, вы задаете процедуру, которую выполняет интерпретатор в соответствии с целями программы.

Структура программы на языке Python

Другой способ понять роль инструкций состоит в том, чтобы вновь вернуться к иерархии понятий, представленной в главе 4, в которой рассказывалось о встроенных объектах и выражениях, управляющих ими. Эта глава рассматривает 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. Выражения обрабатывают объекты и встраиваются в инструкции. Инструкции представляют собой более крупные *логические* блоки

программы – они напрямую используют выражения для обработки объектов, которые мы рассматривали в предыдущих главах. Кроме того, инструкции – это место, где создаются объекты (например, в инструкциях присваивания), а в некоторых инструкциях создаются совершенно новые виды объектов (функции, классы и т. д.). Инструкции всегда присутствуют в модулях, которые сами управляются инструкциями.

Инструкции в языке Python

В табл. 10.1 приводится набор инструкций языка Python.¹ 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 представлены 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 – функциям, классам, модулям и исключениям и заключают в себе крупные понятия программирования, поэтому каждой из них будет посвящен отдельный раздел. Более экзотические инструкции, такие как `exec` (которая компилирует и выполняет программный код в строке), раскрываются далее в книге или в стандартной документации по языку Python.

Таблица 10.1. Инструкции языка Python

Инструкция	Роль	Пример
Присваивание	Создание ссылок	<code>a, b, c = 'good', 'bad', 'ugly'</code>
Вызовы	Запуск функций	<code>log.write("spam, ham\n")</code>
<code>print</code>	Вывод объектов	<code>print 'The Killer', joke</code>
<code>if/elif/else</code>	Операция выбора	<code>if "python" in text: print text</code>

¹ Строго говоря, в Python 2.5 инструкция `yield` превратилась в выражение, а инструкции `try/except` и `try/finally` были объединены (ранее это были отдельные инструкции, теперь же допускается одновременно использовать и `except` и `finally` в одной инструкции `try`). Кроме того, в Python 2.6 появилась новая инструкция управления контекстом `with/as` – грубо говоря, она представляет собой альтернативу операциям `try/finally`, имеющим отношение к исключениям (в 2.5 инструкция `with/as` является необязательным расширением, которое недоступно, пока явно не будет подключено инструкцией `from __future__ import with_statement`). За дополнительной информацией обращайтесь к руководству по языку Python. В будущей версии 3.0 инструкции `print` и `exec` превратятся в функции и появится новая инструкция `nonlocal`, назначение которой будет напоминать назначение существующей ныне инструкции `global`.

Инструкция	Роль	Пример
for/else	Обход последовательности в цикле	for x in mylist: print x
while/else	Циклы общего назначения	while X > Y: print 'hello'
pass	Пустая инструкция-заполнитель	while True: pass
break, continue	Переходы в циклах	while True: if not line: break
try/except/finally	Обработка исключений	try: action() except: print 'action error'
raise	Возбуждение исключений	raise endSearch, location
import, from	Доступ к модулям	import sys from sys import stdin
def, return, yield	Создание функций	def f(a, b, c=1, *d): return a+b+c+d[0] def gen(n): for i in n, yield i*2
class	Создание объектов	class subclass(Superclass): staticData = []
global	Пространства имен	def function(): global x, y x = 'new'
del	Удаление ссылок	del data[k] del data[i:j] del obj.attr del variable
exec	Запуск фрагментов программного кода	exec "import " + modName exec code in gdict, ldict
assert	Отладочные проверки	assert X > Y
with/as	Менеджеры контекста (2.6)	with open('data') as myfile: process(myfile)

История о двух if

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 возможность сравнить его с другими синтаксическими моделями, которые, возможно, 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 теперь взгляните на эквивалентную инструкцию на языке Python:

```
if x > y:  
    x = 1  
    y = 2
```

Первое, что бросается в глаза, – инструкция на языке Python выглядит компактнее, точнее, в ней меньше синтаксических элементов. Это соответствует основным принципам языка; так как Python – это язык сценариев, его основная цель состоит в том, чтобы облегчить жизнь программистам за счет меньшего объема ввода с клавиатуры.

Если быть более точным, то, сравнив две синтаксических модели, можно заметить, 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 двоеточия (:). Все *составные инструкции* в языке Python (т. е. инструкции, которые 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 распространенной 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, обладающих 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 необязательны

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 изнашивать свою клавиатуру, а окружающим сразу будет видно, что вы – бывший программист, использовавший язык C, еще только изучающий язык Python (когда-то и я был таким же). Стиль языка Python состоит в том, чтобы вообще опускать скобки в подобных инструкциях.

Конец строки является концом инструкции

Второй, еще более важный синтаксический элемент, который вы не найдете в программном коде на языке Python, – это точка с запятой. В языке Python 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 считается 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, с которыми вы вскоре познакомитесь. 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 программировали на языке C (если такое состояние вообще возможно...), можете продолжать вставлять точки с запятой в конце каждой инструкции – синтаксис языка допускает 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

читься на использование языка Python. Стил ь 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 Python, и возможно, самый необычный для недавних ex-C-программеров (пока они не поработают с Python 10 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 вложенных 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 от левого края. По величине отступа интерпретатор определяет, где находится начало блока и где – конец:

```
if x > y:  
    x = 1  
    y = 2
```

Под *отступами* (*indentation*) 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 пробела или символы табуляции), и на величину отступов (допускается использовать любое число пробелов или символов табуляции). При этом отступ для одного вложенного блока может существенно отличаться от отступа для другого 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

¹ Кстати, в JavaScript также можно не ставить точку с запятой в конце строки, но широко распространена традиция ставить такой ограничитель, демонстрируя тем самым как раз хороший стиль оформления программного кода. – *Примеч. науч. ред.*

² Известна программистская шутка, уместная в этом контексте: это не bug, это feature. – *Примеч. науч. ред.*

программирования, но это было сделано преднамеренно и является одним из основных способов, которыми язык Python вынуждает программистов писать однородный и удобочитаемый программный код. По существу это означает, что вы должны выстраивать свой программный код вертикально, выравнивая его в соответствии с логической структурой. В результате получается менее противоречивый и более удобочитаемый программный код (в отличие от большей части кода, написанного на С-подобных языках).

Требования к выравниванию программного кода в соответствии с его логической структурой – это главная составляющая, способствующая созданию удобочитаемого кода, и, следовательно, кода, пригодного к многократному использованию и простого в сопровождении как вами, так и другими. Фактически, даже если ранее вы никогда не использовали Python, после прочтения этой книги у вас должна выработаться привычка оформлять отступы в программном 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 полный рабочий день, мне платили зарплату за работу над крупными старыми программами, написанными на языке C++, над которыми долгие годы трудились 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 примерно так:

```
while (x > 0) {
```

Даже еще не дойдя до отступов, мы можем столкнуться с тремя или четырьмя способами расстановки фигурных скобок в С-подобных языках. В организациях часто ведутся жаркие споры и пишутся стандарты по оформлению исходных текстов программ (обсуждение которых сильно выходит за рамки проблем, которые решаются в процессе программирования). Оставим этот вопрос в стороне, ниже приводится пример оформления, с которым мне часто приходилось сталкиваться в программном коде на языке C++. Первый программист, который работал 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 приспосабливать для многократного использования. Отступ – это важная составляющая поддержки удобочитаемости.

Ниже приводится другой пример, на котором, возможно, вам пришлось обжечься, если вы достаточно много программировали на С-подобном языке. Взгляните на следующие инструкции языка С:

```

if (x)
    if (y)
        statement1;
else
    statement2;

```

К какому оператору `if` относится инструкция `else`? Это удивительно, но инструкция `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 С и она может привести к неправильному пониманию программного кода тем, кто его изменяет, и к появлению ошибок при его изменении, которые будут обнаружены, только когда марсоход врежется в скалу!

Такого не может произойти в языке Python, потому что отступы для него имеют 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 вертикальной линии, связаны логически (внешний оператор `if x`). В некотором смысле Python – это язык типа WYSIWYG (What You See Is What You Get – что видишь, то и получаешь) – что вы видите, то и получаете, потому что порядок оформления программного кода определяет порядок его выполнения независимо 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 благополучной 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 репозиторий, он автоматически проверялся сценарием, анализировавшим оформление отступов. Если сценарий обнаруживал несоответствия, на следующий день утром мы получали электронные письма с замечаниями – и такой порядок завели наши начальники!

По моему мнению, даже если язык программирования и не требует этого, хороший программист должен понимать, какое важное значение имеет выравнивание 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 дружественным (для программистов) интерфейсом обладает встроенной поддержкой синтаксической модели языка Python. В Python-среде разработки IDLE, например, отступы оформляются автоматически¹, когда начинается ввод вложенного 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 отступов: 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 вложенный блок, и возвращайтесь на предыдущий уровень, чтобы закрыть его. 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

¹ Напомним, IDLE – среда разработки, специально созданная Guido van Rossum в проекте Python. Первый релиз вышел вместе с версией Python 1.5.2. Одно из объяснений использования такого слова (`idle` – бездействующий, ленивый), в отличие от принятого в других языковых средах термина IDE, объяснено 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. науч. ред.*

уберите фигурные скобки и ваш программный код будет удовлетворять 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 существуют 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 возможно 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 точки 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 строку, что, скорее всего, не нашло бы понимания у ваших коллег!).

Другое специальное правило, применяемое к инструкциям, по сути является обратным к предыдущему: допускается записывать одну инструкцию в нескольких 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 нескольких строках: инструкция не будет считаться законченной, пока интерпретатор Python не достигнет строки с закрывающей скобкой. Например, литерал 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. Словари также могут располагаться в нескольких строках, а с помощью круглых скобок можно оформить мно-

гострочные кортежи, вызовы функций и выражения. Отступы в строках, где продолжается инструкция, в учет не принимаются, хотя здравый смысл диктует, что строки все-таки должны иметь некоторые отступы для обеспечения удобочитаемости.

Круглые скобки являются самым универсальным средством, потому что в них можно заключить любое выражение. Добавьте левую скобку, и вы сможете перейти на следующую строку и продолжить свою инструкцию:

```
X = (A + B +  
     C + D)
```

Между прочим, такой прием допускается применять и к составным инструкциям. Если вам требуется записать длинное выражение, оберните его круглыми 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 следующую строку: если предыдущая строка заканчивается символом обратного слеша:

```
X = A + B + \  
     C + D
```

Но это устаревшее правило, которое не рекомендовано к использованию в новых программах, потому что символы обратного слеша мало заметны и ненадежны (не допускается наличие каких-либо других символов после символа обратного слеша). Кроме того, это рассматривается как пережиток языка C, где очень часто используются макроопределения `#define`; в Питонляндии нужно все делать так, как это делают питонисты, а не как программисты на языке C.

Специальный случай оформления блока

Как уже говорилось выше, инструкции во вложенном блоке обычно объединяются по величине отступа. Специальный случай: тело составной инструкции может располагаться в той же строке, что и основная инструкция, после символа двоеточия:

```
if x > y: print x
```

Это позволяет записывать в одной строке условные операторы, циклы и т. д. Однако такой прием будет работать, только если тело составной инструкции не содержит других составных инструкций. То есть после двоеточия могут следовать только простые инструкции – инструкции присваивания, инструкции `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 инструкции 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 отступы для вложенных блоков, ваш программный код будет проще читать и вносить в него изменения. Чтобы увидеть одно из этих правил в действии (когда однострочная инструкция `if` используется для прерывания выполнения цикла), давайте перейдем к следующему 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 следующих главах, но эти правила работают одинаково везде в языке Python. Для начала рассмотрим короткий практический пример, который продемонстрирует способ применения синтаксических правил к оформлению инструкций и вложенных блоков на практике и попутно познакомит вас с некоторыми инструкциями.

Простой интерактивный цикл

Предположим, что от нас требуется написать программу на языке Python, которая 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 необходимо написать цикл, который будет считывать одну или более строк, введенных пользователем с клавиатуры, и выводить их обратно на экран. Другими словами, вам нужно написать классический цикл, выполняющий операции чтения/вычисления/вывода.

В языке Python для реализации таких интерактивных циклов используется типичный шаблон, который выглядит, как показано ниже:

```
while True:
    reply = raw_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 инструкции `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 следует вложенный блок про-

граммного кода, который продолжает выполняться в цикле, пока условное выражение возвращает истину (слово `True` здесь всегда возвращает значение истины).

- Встроенная функция `raw_input`, с которой мы уже встречались ранее в этой книге, используется здесь как универсальное средство получения ввода с клавиатуры – она выводит подсказку, текст которой содержится в необязательном строковом аргументе, и возвращает введенный пользователем ответ в виде строки.
- Однострочная инструкция `if`, которая оформлена в соответствии со специальным правилом для вложенных блоков: тело инструкции `if` располагается в той же строке, что и основная инструкция, после символа двоеточия, а не на отдельной строке под ней с соответствующим отступом. Она одинаково 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 программа продолжает свою работу с первой инструкции, которая расположена вслед за циклом. Без этой инструкции цикл `while` работал бы вечно, поскольку его условное выражение всегда возвращает истину.

В результате такая комбинация инструкций означает следующее: «читать строки, введенные пользователем, и выводить их после преобразования всех символов в верхний регистр, пока пользователь не введет строку "stop"». Существуют и другие способы записи такого цикла, но данная форма очень часто встречается в программном коде на языке Python.

Обратите внимание, что все три строки, вложенные в инструкцию цикла `while`, имеют одинаковые отступы, благодаря этому они визуально образуют вертикальную линию блока программного кода, ассоциированного с инструкцией цикла `while`. Тело цикла завершается либо с концом файла, либо с первой инструкцией, имеющей меньший отступ.

Запустив этот фрагмент, мы получили следующий порядок взаимодействий с ним:

```
Enter text:spam
SPAM
Enter text:42
42
Enter text:stop
```

Математическая обработка данных пользователя

Итак, наш сценарий работает, а теперь предположим, что вместо преобразования символов текстовой строки в верхний регистр нам необходимо выполнить некоторые математические действия над введенными

пользователем числами, — например, вычислить квадраты чисел, что может получиться не совсем так, как ожидают наиболее молодые и нетерпеливые пользователи. Для достижения 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'
(Перевод: TypeError: неподдерживаемые типы операндов для **
или pow(): 'str' и 'int')
```

Это не работает, потому что (как обсуждалось в предыдущей части книги) интерпретатор выполняет преобразование типов объектов в выражениях, только если они являются числами, а ввод пользователя всегда передается сценарию в виде строки. Мы не можем возвести строку цифр в степень, не преобразовав ее в целое число вручную:

```
>>> int(reply) ** 2
400
```

Вооружившись этой информацией, мы можем переделать наш цикл для выполнения математических действий:

```
while True:
    reply = raw_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 выполнения 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 такого 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
Enter text:40
1600
Enter text:stop
Bye
```

Обработка ошибок проверкой ввода

Пока все идет хорошо, но посмотрите, что произойдет, если пользователь введет неверную строку:

```
Enter text:xxx
```



```
...текст сообщения об ошибке опущен...
ValueError: invalid literal for int() with base 10: 'xxx'
(Перевод: ValueError: неверный литерал для int() по основанию 10: 'xxx')
```

Встроенная функция `int` возбуждает исключение, когда сталкивается с ошибкой. 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 следующей версии нашего интерактивного сценария используется полная версия условного оператора `if`, с помощью которого предотвращается возможность появления исключений:

```
while True:
    reply = raw_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, одну или более необязательных проверок `elif` ("else if") и соответствующих им вложенных блоков кода и необязательная 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.

Части `if`, `elif` и `else` в предыдущем примере принадлежат одной и той же инструкции, так как вертикально они расположены на одной линии (т. е. имеют 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 обнаружены 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 языке Python существует более универсальный способ, который состоит в том, чтобы перехватывать и обрабатывать ошибок с помощью инструкции `try`. Эта инструкция подробно будет рассматриваться в последней части книги, но, используя инструкцию `try` в качестве предварительного знакомства, мы можем упростить программный код предыдущей версии сценария:

```
while True:
    reply = raw_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 заменили явную проверку наличия ошибки программным кодом, который предполагает, 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 пытаемся выполнить), с последующей частью `except`, где располагается программный код обработки исключения. Далее следует часть `else`, программный код которой выполняется, если в части `try` исключение не возникло. Интерпретатор сначала выполняет часть `try`, затем выполняет либо часть `except` (если возникло 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, ключевое 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.

Напомним, 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, проверяющего наличие ошибок, и представляет собой достаточно универсальный подход к обработке необычных ситуаций.

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 = raw_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 выражение `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 вправо. 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
```

В заключение

В этой главе мы коротко познакомились с синтаксисом инструкций языка Python. 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 синтаксиса языка Python). Кроме того, мы рассмотрели несколько исключений из этих правил, включая многострочные инструкции, однострочные условные инструкции и циклы, а также рассмотрели синтаксис инструкций в действии.

В следующей главе мы приступим к глубокому изучению основных процедурных инструкций языка Python. Как будет показано далее, все инструкции следуют 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 блоку 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 инструкций точкой с запятой?
7. Для чего предназначена инструкция `try`?
8. Какую наиболее распространенную ошибку допускают начинающие программисты на языке Python?

Ответы

1. Обязательными синтаксическими элементами в C-подобных языках программирования являются круглые скобки, окружающие выражения проверки 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 завершаться точками с запятой. Если инструкция располагается в нескольких строках, она должна завершаться закрывающей скобкой.
3. Все инструкции во вложенном блоке имеют одинаковые отступы, состоящие из одинакового числа символов пробела или табуляции.
4. Инструкция может располагаться в нескольких строках, если заключить ее в пару круглых, квадратных или фигурных скобок. Ин-

струкция заканчивается в строке, где интерпретатор Python встречает закрывающую парную 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 составной. Однако такой способ размещения инструкций приводит к снижению удобочитаемости программного кода.
7. Инструкция `try` используется для перехвата и обработки исключений (ошибок), возникающих в сценариях на языке Python. Как правило, она представляет альтернативу программному коду, который создается для выявления ошибочных ситуаций.
8. Наиболее распространенная ошибка среди начинающих программистов состоит в том, что они забывают добавлять двоеточие в конце основной части составных инструкций.

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 основы – инструкции присваивания, инструкции выражений и инструкцию вывода на экран. Мы уже видели все эти инструкции в действии, но здесь мы восполним подробности, которые опускались ранее. Несмотря на всю простоту этих инструкций, в чем вы могли убедиться ранее, каждая из них имеет дополнительные возможности, которые пригодятся, когда вы начнете писать настоящие программы на языке Python.

Инструкции присваивания

Ранее мы уже использовали инструкции присваивания для назначения имен объектам. В канонической форме *цель* присваивания записывается 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 большинстве случаев присваивание выполняется достаточно просто, однако оно обладает 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 создают ссылки на объекты и никогда не создают копии объектов. Вследствие этого переменные в языке Python больше напоминают указатели, чем области хранения данных.
- **Переменные создаются при первом присваивании.** Интерпретатор Python создает переменные, когда им впервые присваиваются зна-

чения (т. е. ссылка на объект), благодаря этому отсутствует необходимость предварительного 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 в выражении, оно замещается объектом, на который ссылается эта переменная.

- **Прежде чем переменную можно будет использовать, ей должно быть присвоено 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 опечаток в программном коде.
- **Инструкции `import`, `from`, `def`, `class`, `for`, а также передача аргументов функциям являются неявными инструкциями присваивания.** В этом разделе мы сосредоточим все свое внимание на инструкции `=`, однако в языке Python присваивание может выполняться в самых разных контекстах. Например, далее мы увидим, что импорт модуля, определение функции или класса, указание переменной в цикле `for` и передача аргументов функции неявно выполняют присваивание. Операция присваивания выполняется 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 инструкциях присваивания. В табл. 11.1 приводятся различные формы инструкции присваивания, которые встречаются в языке Python.

Таблица 11.1. Формы инструкции присваивания

Операция	Интерпретация
<code>spam = 'Spam'</code>	Каноническая форма
<code>spam, ham = 'yum', 'YUM'</code>	Присваивание кортежей (позиционное)
<code>[spam, ham] = ['yum', 'YUM']</code>	Присваивание списков (позиционное)
<code>a, b, c, d = 'spam'</code>	Присваивание последовательностей, обобщенное
<code>spam = ham = 'lunch'</code>	Групповое присваивание одного значения
<code>spams += 42</code>	Комбинированная инструкция присваивания (эквивалентно инструкции <code>spams = spams + 42</code>)

Первая форма из табл. 11.1 является наиболее распространенной: она связывает переменную (или элемент структуры данных) с единственным объектом. Другие формы имеют особое назначение и являются необязательными, хотя многие и находят их 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 присваивание слева направо. Например, во второй строке таблицы с именем `spam` ассоциируется строка `'yum'`, а с именем `ham` ассоциируется строка `'YUM'`. Внутри интерпретатор Python сначала создает элементы кортежа справа, поэтому часто эта операция называется присваиванием неупакованного кортежа.

Присваивание последовательностей

В недавних версиях Python операции присваивания кортежей и списков были обобщены в то, что теперь называется операцией присваивания последовательностей – любая последовательность имен может быть связана с любой последовательностью значений, и интерпретатор свяжет элементы согласно их позициям. Фактически в последовательностях мы можем смешивать разные типы. Инструкция присваивания в третьей строке табл. 11.1, например, связывает кортеж имен со строкой символов: имени `a` присваивается символ `'s'`, имени `b` присваивается символ `'p'` и т. д.

Групповое присваивание одного значения

Пятая строка в табл. 11.1 демонстрирует форму группового присваивания. В этой форме интерпретатор присваивает ссылку на один и тот же объект (самый правый объект) всем целям, расположенным левее. Инструкция в таблице присваивает обоим именам `spam` и `ham` ссылку на один и тот же объект, строку `'lunch'`. Результат будет тот же, как если бы были выполнены две инструкции: `ham = 'lunch'` и `spam = ham`, поскольку здесь `ham` интерпретируется как оригинальный объект-строка (т. е. не отдельная копия этого объекта).

Комбинированное (дополняющее) 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` дает тот же результат, что и инструкция `spam = spam + 42`, единственное отличие состоит в том, что комбинированная форма имеет более компактный вид и обычно выполняется быстрее. Для каждого двухместного оператора в языке Python существует своя комбинированная инструкция присваивания.

Присваивание последовательностей

В этой книге мы уже использовали инструкцию присваивания в канонической форме. Ниже приводятся несколько примеров инструкций присваивания последовательностей в действии:

```
% 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 выполняет присваивание значений в одной инструкции.

Операция присваивания кортежей дает возможность использовать прием, который представлен в упражнениях ко второй части книги. Так как в процессе выполнения инструкции интерпретатор создает временный кортеж, где сохраняются оригинальные значения переменных справа, данная форма присваивания 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 присваивать кортеж значений списку переменных, строки символов – кортежу переменных и т. д. В любом случае интерпретатор свяжет элементы последовательности 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 последовательностей допускается указывать не только последовательности, но и любые объекты, обеспечивающие возможность итераций по элементам. Эту, еще более общую концепцию, мы будем рассматривать в главах 13 и 17.

Дополнительные варианты инструкции присваивания последовательностей

Одно замечание – 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:

```
>>> 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
(Перевод: ValueError: слишком много распакованных значений)
```

В общем случае нам необходимо получить срез. Существует несколько вариантов 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 *вложенные последовательности*, и интерпретатор распаковывает их части в соответствии 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 выполняет присваивание второй строки целиком ('AM') переменной c. В этом случае вложенная последовательность слева, имеющая форму объекта, должна соответствовать 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, данный прием может также использоваться в списках аргументов функций, потому что передача аргументов выполняется присваиванием (как будет показано в следующей части книги).

Кроме того, операция присваивания последовательности с распаковкой 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 значениями 0, 1 и 2 соответственно (это эквивалент *перечислимых* типов данных в языке Python, которые, возможно, вам приходилось встречать в других языках программирования). Чтобы понять происходящее, вы должны знать, что встроенная функция `range` генерирует список последовательных целых чисел:

```
>>> range(3)      # Попробуйте 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:]
...     print front, L
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

Присваивание кортежа в цикле здесь можно было бы заменить двумя следующими 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`. В данном случае эффект, который дает операция присваивания кортежа, можно было бы получить инструкцией `front = L.pop(0)`,

но это будет операция непосредственного изменения объекта. О циклах `while` и о других (часто лучших) способах обхода последовательностей с помощью циклов `for` вы узнаете больше 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 памяти). 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 устанавливать начальные значения в счетчиках, прежде чем они смогут использоваться 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 присваиваемый объект является неизменяемым, совершенно неважно, как много ссылок на него будет создано.

Но, как обычно, следует проявлять осторожность, выполняя присваивание переменным изменяемых объектов, таких как списки или словари:

```
>>> a = b = []
>>> b.append(42)
>>> a, b
([42], [42])
```

На этот раз, поскольку `a` и `b` ссылаются на один и тот же объект, непосредственное добавление значения к объекту через переменную `b` будет воздействовать и на переменную `a`. В действительности это всего лишь другой пример взаимовлияния 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 создавался новый пустой объект с помощью отдельных литеральных выражений:

```
>>> a = []
>>> b = []
>>> b.append(42)
>>> a, b
([], [42])
```

Дополняющее присваивание

Начиная с версии Python 2.0, в языке появился набор дополнительных инструкций присваивания, перечисленных в табл. 11.2. Известные как *инструкции дополняющего 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 формы 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 поддерживаемого двухместного оператора. Например, ниже приводятся два способа прибавления 1 к переменной:

```
>>> x = 1
>>> x = x + 1      # Традиционная форма записи
>>> x
2
>>> x += 1         # Дополняющая
>>> x
3
```

Если дополняющую форму применить к строкам, будет выполнена операция конкатенации. Таким образом, вторая строка ниже эквивалентна более длинной инструкции `S = S + "SPAM"`:

```
>>> S = "spam"
>>> S += "SPAM"    # Выполняется операция конкатенации
>>> S
'spamSPAM'
```

Как показано в табл. 11.2, для каждого двухместного оператора (т. е. для оператора, слева и справа от которого располагаются значения, участвующие в операции) в языке Python существует своя инструкция

дополняющего присваивания. Например, инструкция $X *= Y$ выполняет умножение и присваивание, $X >>= Y$ — сдвиг вправо и присваивание, и т. д. Инструкция $X // = Y$ (деление с округлением вниз) была добавлена в версии Python 2.2.

Дополняющие присваивания обладают следующими преимуществами:¹

- Уменьшается объем ввода с клавиатуры. Нужно ли мне продолжать?
- Левая часть инструкции должна быть получена всего один раз. В инструкции « $X += Y$ » X может оказаться сложным выражением, которое в комбинированной форме должно быть вычислено всего один раз. В более длинной форме запись « $X = X + Y$ » X появляется дважды и поэтому данное выражение должно быть вычислено дважды. Вследствие этого дополняющие присваивания выполняются обычно быстрее.
- Автоматически выбирается оптимальный алгоритм выполнения. Для объектов, поддерживающих возможность непосредственного изменения, дополняющие присваивания автоматически выполняются непосредственно на самих объектах вместо выполнения более медленной операции копирования.

И последний момент, который требует дополнительных разъяснений. Дополняющие присваивания в случае операций, выполняемых непосредственно над объектами, могут служить для оптимизации изменяемых объектов. Вспомним, что списки могут расширяться 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 операцию конкатенации, либо вызвать метод `extend`:²

```
>>> 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 практически так же, как метод `extend`.

```
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

В обоих случаях операция конкатенации несет в себе меньше побочных эффектов при работе с разделяемыми ссылками на объекты, но вообще она выполняется медленнее, чем эквивалентные операции, воздействующие на объект непосредственно. Операция конкатенации должна создать новый объект, копию в списке слева, и затем копию в списке справа. В противовес ей метод, воздействующий на объект непосредственно, просто добавляет новый элемент в конец блока памяти.

При использовании дополняющего присваивания для расширения списка мы можем не думать об этих деталях – интерпретатор Python автоматически 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]
```

Дополняющее присваивание и разделяемые ссылки

Такой порядок вещей чаще всего именно то, что нам требуется, но необходимо учитывать – он предполагает, что применительно к спискам операция `+=` выполняет изменения непосредственно в объекте, а это далеко не то же самое, что операция конкатенации `+`, в результате которой всегда создается новый объект. Как всегда в случае использования разделяемых ссылок, различия в поведении 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 избежать 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 язы-

ке Python имена появляются в момент, когда им присваиваются некоторые значения, однако существует несколько правил, которым желательно следовать при выборе имен для всего сущего в ваших программах:

Синтаксис: (символ подчеркивания или алфавитный символ) + (любое число символов, цифр или символов подчеркивания)

Имена переменных должны начинаться с символа подчеркивания или с алфавитного символа, за которым может следовать произвольное число алфавитных символов, цифр или символов подчеркивания. Допустимыми именами являются: `_spam`, `spam` и `Spam_1`, а `1_spam`, `spam$` и `@#!` – недопустимыми.

Регистр символов в именах имеет значение: имена SPAM и spam считаются различными именами

В языке Python регистр символов всегда принимается во внимание, как в именах, которые вы создаете, так и в зарезервированных словах. Например, имена `X` и `x` – это две разные переменные. Для обеспечения переносимости регистр символов учитывается и в именах импортируемых модулей, даже на платформах, где файловые системы не учитывают регистр символов.

Запрещено использовать зарезервированные слова

Имена определяемых вами переменных не могут совпадать с зарезервированными словами, имеющими в языке Python специальное назначение. Например, если попытаться использовать переменную с именем `class`, интерпретатор выведет сообщение о синтаксической ошибке, однако имена `class` и `Class` являются вполне допустимыми. В табл. 11.3 перечислены слова, которые в настоящее время зарезервированы языком Python (и, следовательно, запрещены для использования в качестве имен переменных).

Таблица 11.3. Зарезервированные слова языка Python

<code>and</code>	<code>elif</code>	<code>if</code>	<code>pass</code>
<code>as</code> (в 2.6 и более поздних)	<code>else</code>	<code>import</code>	<code>print</code>
<code>assert</code>	<code>except</code>	<code>in</code>	<code>raise</code>
<code>break</code>	<code>exec</code>	<code>is</code>	<code>return</code>
<code>class</code>	<code>finally</code>	<code>lambda</code>	<code>try</code>
<code>continue</code>	<code>for</code>	<code>nonlocal</code> (в 3.0)	<code>while</code>
<code>def</code>	<code>from</code>	<code>not</code>	<code>with</code> (в 2.6 и более поздних)
<code>del</code>	<code>global</code>	<code>or</code>	<code>yield</code>



В версии Python 2.2 слово `yield` было необязательным расширением, но, начиная с версии 2.3, оно стало стандартным зарезервированным словом. Оно используется совместно с генераторами

функций – новой особенностью, которая будет обсуждаться в главе 17. Это один из небольшого числа элементов языка, в котором была нарушена обратная совместимость с прежними версиями. Однако такое преобразование слова `yield` было выполнено не вдруг – в версии 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 менеджере контекстов (новая форма обработки исключений). В версии 2.5 эти слова не считаются зарезервированными, пока менеджер контекстов не будет включен вручную (подробности приводятся в главе 27). При использовании в версии 2.5 слова `with` и `as` вызывают появление предупреждений о грядущих изменениях – исключение составляет версия IDLE в Python 2.5, где эта особенность включается автоматически (т. е. использование этих слов в качестве имен переменных в версии 2.5 будет вызывать ошибку, но только в интерфейсе IDLE).

Зарезервированные слова языка Python всегда записываются символами нижнего регистра и они действительно зарезервированы – в отличие от имен в области видимости по умолчанию, о которой будет рассказываться в следующей части книги, вы не можете переопределять зарезервированные слова посредством операции присваивания (например, `and = 1` приведет к появлению ошибки).¹

Кроме того, поскольку имена модулей в инструкции `import` становятся переменными в сценарии, это накладывает ограничения на имена файлов с модулями. Вы можете создать файлы с именами `and.py` и `mycode.py`, но вы не сможете импортировать их, потому что их имена без расширения «.py» в программном коде будут преобразованы в имена переменных и поэтому должны следовать только что обозначенным правилам (зарезервированные слова запрещены, а символы дефиса являются недопустимыми, хотя можно использовать символы подчеркивания).

Соглашения по именованию

Помимо указанных правил существует еще целый ряд *соглашений* – правил, которые не являются обязательными, но которым обычно следуют на практике. Например, имена с двумя символами подчеркивания в начале и в конце (например, `__name__`) обычно имеют особый смысл для интерпретатора, поэтому 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 импортируются инструкцией `from module import *` (см. главу 19).

¹ Тем не менее, в Jython, реализации Python на языке Java, имена пользовательских переменных иногда могут совпадать с зарезервированными словами языка Python.

- Имена, имеющие два символа подчеркивания в начале и в конце (`__X__`), являются системными именами, которые имеют особый смысл для интерпретатора.
- Имена, начинающиеся с двух символов подчеркивания и не оканчивающиеся двумя символами подчеркивания (`__X`), являются локальными (автоматически подменяются другими именами) для объемлющего класса (как описывается в главе 26).
- Имя, состоящее из единственного символа подчеркивания (`_`), хранит результат последнего выражения при работе в интерактивной оболочке.

В дополнение к этим соглашениям существует еще ряд других соглашений, которым обычно стремятся следовать программисты. Например, далее в книге мы увидим, что имена классов обычно начинаются с заглавного символа, а имена модулей – с прописного, и что имя `self`, хотя и не является зарезервированным, играет особую роль в классах. В четвертой части книги мы познакомимся с еще одной крупной категорией имен, которые называются *встроенными*, т. е. предопределенными, но не зарезервированными (вследствие чего они допускают переопределение: инструкция `open = 42` является вполне допустимой, но иногда 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 крайне важно сохранить четкое понимание различий между именами и объектами в языке Python. Как говорилось в главе 6 «Интерлюдия о динамической типизации», 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 связывается 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 программировании на языке Python.¹ В четвертой части книги вы

¹ Если вам приходилось пользоваться языком C++, возможно, вас заинтересует, что в отличие от C++ в языке Python отсутствует объявление `const`. Некоторые объекты могут быть неизменяемыми, но имена всегда допускают выполнение операции присваивания. Кроме того, в языке Python имеются средства сокрытия имен в классах и модулях, но они также не являются аналогами объявлений в языке C++.

узнаете, что имена находятся внутри *области видимости*, которая определяет, где эти имена могут использоваться – место, где выполняется присваивание, определяет, где это имя будет видимо.

Инструкции выражений

В языке Python выражения также могут использоваться в качестве инструкций (т. е. в отдельной строке). Однако, поскольку результат вычисления таких 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 возвращая 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 результаты 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
<code>spam(eggs, ham)</code>	Вызов функции
<code>spam.ham(eggs)</code>	Вызов метода
<code>spam</code>	Вывод значения переменной в интерактивной оболочке интерпретатора
<code>spam < ham and ham != eggs</code>	Составное выражение
<code>spam < ham < eggs</code>	Проверка на принадлежность диапазону

Последняя строка в таблице представляет специальную форму: язык Python позволяет объединять операции сравнения, чтобы выполнить серию сравнений в таких операциях, как проверка на принадлежность

диапазону значений. Например, выражение $(A < B < C)$ проверяет, принадлежит ли B диапазону значений от A до C — оно эквивалентно логическому выражению $(A < B \text{ and } B < C)$, но оно проще воспринимается визуально (и его проще вводить). Составные выражения обычно не записываются в виде самостоятельных инструкций, но с точки зрения синтаксиса являются вполне допустимыми и могут быть даже полезными при работе в интерактивной оболочке, когда вы не уверены в результате выражения.

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 выражения. Сделано это специально для того, чтобы помочь избежать ошибок — вы могли бы случайно изменить переменную, введя вместо оператора проверки равенства $==$ оператор присваивания $=$. В главе 13 будет показано, как отсутствие такой 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 используются 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 `append`, `sort` и `reverse`, всегда выполняют непосредственное изменение объекта, но они не возвращают список, который был изменен с их помощью.

В действительности 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 кон-

тексте выполнения некоторых операторов цикла, с которыми мы познакомимся в последующих главах.

Инструкция print

Инструкция `print` производит вывод на экран – это просто удобный для программистов интерфейс к стандартному потоку вывода. С технической точки зрения эта инструкция преобразует объекты в текстовое представление и посылает результат на устройство стандартного вывода.

Поток стандартного вывода в языке Python – это то же самое, что `stdout` в языке C; обычно он отображается на окно, в котором была запущена программа на языке Python (если стандартный вывод не был перенаправлен в файл или в канал средствами системной командной оболочки).

В главе 9 мы рассматривали некоторые методы для работы с файлами, которые выводят текст. Инструкция `print` чем-то похожа на них, но она имеет более специализированное назначение: инструкция `print` записывает объекты в поток `stdout` (с соблюдением некоторого форматирования по умолчанию), в то время как методы файлов записывают строки в произвольные файлы. Так как поток стандартного вывода в языке Python доступен в виде объекта `stdout` встроенного модуля `sys` (т. е., `sys.stdout`), вполне возможно имитировать поведение инструкции `print` с помощью методов записи в файл, хотя использование `print` выглядит проще.

Таблица 11.5. Перечень форм инструкции `print`

Операция	Интерпретация
<code>print spam, ham</code>	Вывод объектов в <code>sys.stdout</code> ; добавляет пробел между объектами и символ конца строки
<code>print spam, ham,</code>	То же самое, только на этот раз символ конца строки не добавляется
<code>print >> myfile, spam, ham</code>	Текст передается методу <code>myfile.write</code> , а не <code>sys.stdout.write</code>

Мы уже видели инструкцию `print` в действии. По умолчанию она добавляет пробел между элементами, отделенными запятыми, и символ конца строки в конце текущей строки вывода:

```
>>> 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 (чтобы позднее можно было продолжить вывод текста в текущую

строку), инструкцию `print` следует завершать символом запятой, как показано во второй строке табл. 11.5. Чтобы подавить вывод пробела между элементами, не следует производить вывод таким способом – вместо этого нужно самостоятельно собрать строку с помощью средств конкатенации и форматирования, о которых рассказывалось в главе 7, и вывести эту строку:

```
>>> print x + y
ab
>>> print 's...%s' % (x, y)
a...b
```

Программа «Hello World» на языке Python

Чтобы вывести сообщение «hello world», достаточно просто напечатать строку:

```
>>> print 'hello world'          # Вывести строковый объект
hello world
```

Поскольку результаты выражений в интерактивной оболочке выводятся автоматически, вы можете даже не использовать инструкцию `print` – просто введите выражение, которое требуется вывести, и результат его вычисления немедленно будет выведен:

```
>>> 'hello world'                # Интерактивная оболочка выводит автоматически
'hello world'
```

В действительности инструкция `print` – это всего лишь эргономичная особенность языка Python – она обеспечивает простой интерфейс к объекту `sys.stdout`, добавляя незначительный объем форматирования. Фактически, если вам нравится идти более трудным путем, вы можете запрограммировать вывод таким 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, связанного с потоком вывода. Инструкция `print` скрывает большую часть этих подробностей, предоставляя простой инструмент для решения простых задач вывода.

Перенаправление потока вывода

Итак, почему я показал пока единственный способ вывода? Как оказывается, объект `sys.stdout` обеспечивает возможность вывода, эквивалентную базовой методике, используемой в языке Python. Вообще говоря, инструкция `print` и объект `sys.stdout` связаны между собой следующим образом. Следующая инструкция:

```
print X
```

является эквивалентом более длинной:

```
import sys
sys.stdout.write(str(X) + '\n')
```

которая вручную выполняет преобразование объекта в строку с помощью функции `str`, добавляет символ новой строки с помощью оператора `+` и вызывает метод `write` потока вывода. Более длинная форма сама по себе не так удобна в использовании. Однако полезно знать, что она является точным эквивалентом инструкции `print`, потому что существует возможность перенаправить `sys.stdout` в место, отличное от стандартного потока вывода. Другими словами, эта эквивалентность 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
```

Здесь мы вручную перенаправили объект `sys.stdout` в файл, открытый вручную в режиме добавления. После этого все инструкции `print` в программе будут выводить текст в конец файла *log.txt*, а не в стандартный поток вывода. Инструкции `print` благополучно продолжают 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 инструкции `print` в программе.

Фактически, как будет говориться в ближайшей врезке, описывающей инструкцию `print` и объект `stdout`, существует возможность перенаправить `sys.stdout` в объект, который даже не является файлом, при условии, что он соответствует ожидаемому протоколу (метод `write`), — в этом случае объект может быть классом, который может обрабатывать и перенаправлять выводимый текст произвольным образом.

Этот прием с перенаправлением потока вывода в первую очередь может оказаться полезен в программах, изначально рассчитанных на использование инструкции `print`. Если известно, что весь вывод должен отправляться в файл, вы всегда сможете организовать вызов методов записи в файл. При перенаправлении потока вывода в программах, основанных на использовании инструкции `print`, настройка объекта `sys.stdout` обеспечивает удобную альтернативу изменению поведения всех инструкций `print` или применению перенаправления средствами командной оболочки системы.

Расширение `print >> file`

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 после вывода данных в файл потребуется вернуться обратно к выводу на экран. Но поскольку `sys.stdout` является обычным объектом, вы всегда можете в случае необходимости сохранить его и восстановить позднее:¹

```
>>> import sys
>>> temp = sys.stdout           # Сохранить для последующего восстановления
>>> sys.stdout = open('log.txt', 'a') # Перенаправить вывод в файл
>>> print 'spam'                # Вывести 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
```

Потребность в таком перенаправлении возникает на удивление часто, а ручное сохранение и восстановление оригинального потока вывода – процедура достаточно сложная, что привело к появлению расширения для инструкции `print`, которое делает такое перенаправление ненужным. Когда инструкция `print` начинается с символов `>>`, вслед за которыми указывается объект файла (или другой объект) для вывода, такая инструкция посылает текст методу `write` указанного объекта, но при этом не происходит перенаправления объекта `sys.stdout`. Поскольку в этом случае перенаправление носит временный характер, обычные инструкции `print` продолжают выводить текст в оригинальный поток вывода:

```
log = open('log.txt', 'a')
print >> log, x, y, z      # Вывести в объект, напоминающий файл
print a, b, c              # Вывести в оригинальный поток вывода
```

Расширение `>>` удобно использовать, когда в одной и той же программе необходимо организовать вывод и в файл, и в стандартный поток вывода. Однако, если вы собираетесь использовать эту форму инструкции, вам потребуется создать объект-файл (или объект, который имеет метод `write`, как и объект файла) и передавать инструкции этот объект, а не строку с именем файла:

¹ Можно также использовать относительно новый атрибут `__stdout__` объекта `sys.stdout`, 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 `__stdout__`, чтобы вернуться к оригинальному потоку вывода. За дополнительными подробностями обращайтесь к описанию модуля `sys` в руководстве по стандартной библиотеке.


```
>>> log = open('log.txt', 'w')
>>> print >> log, 1, 2, 3
>>> print >> log, 4, 5, 6
>>> log.close()
>>> print 7, 8, 9
7 8 9
>>> print open('log.txt').read()
1 2 3
4 5 6
```

Эта расширенная форма инструкции `print` нередко используется для вывода сообщений об ошибках в стандартный поток ошибок, `sys.stderr`. Вы можете либо использовать его метод `write` и форматировать выводимые строки вручную, либо использовать синтаксис перенаправления:

```
>>> import sys
>>> sys.stderr.write(('Bad!' * 8) + '\n')
Bad! Bad! Bad! Bad! Bad! Bad! Bad! Bad!

>>> print >> sys.stderr, 'Bad!' * 8
Bad! Bad! Bad! Bad! Bad! Bad! Bad! Bad!
```

Придется держать в уме: `print` и `stdout`

Эквивалентность инструкции `print` и метода `write` объекта `sys.stdout` имеет большое значение. Она позволяет перенаправить объект `sys.stdout` в определяемый пользователем объект, который поддерживает те же методы, что и файлы (например, `write`). Так как инструкция `print` всего лишь передает текст методу `sys.stdout.write`, вы можете перехватывать выводимый текст, перенаправив `sys.stdout` в объект, обладающий методом `write` для обработки текста.

Например, можно отправить текст в окно графического интерфейса или отправить его в несколько мест, определив объект с методом `write`, который выполнит все необходимые действия. Пример использования такой возможности вы увидите далее в книге, 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` является тем, что в следующей части книги мы назовем *полиморфной* операцией, — она не интересуется тем, что из себя представляет объект `sys.stdout`, ей нужен всего лишь метод (т. е. интерфейс) с именем `write`. В последних версиях Python перенаправление в объекты реализовано еще проще, с помощью расширенной формы `>>` инструкции `print`, благодаря чему нам не требуется явно перенаправлять объект `sys.stdout`:

```
myobj = FileFaker()           # Перенаправление вывода для одной инструкции
print >> myobj, someObjects    # Не влияет на объект sys.stdout
```

Встроенная функция `raw_input()` в языке Python читает данные из файла `sys.stdin`, поэтому существует возможность перенаправить ввод аналогичным образом, используя классы, реализующие метод `read`. Смотрите примеры использования функции `raw_input` и цикла `while`, которые приводятся в главе 10, чтобы получить дополнительные начальные сведения по этой теме.

Обратите внимание: вывод текста осуществляется в поток `stdout`, что 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
```



Предполагается, что в Python 3.0 инструкция `print` превратится во встроенную функцию с эквивалентными возможностями, но с несколько иным синтаксисом. Выходной файл и необходимость вывода символа конца строки будут определяться аргументами с зарезервированными именами. Например, инструкция `print x`, `y` превратится в вызов `print(x, y)`, а инструкция `print >> f, x` — в вызов `print(x, file=f, end='')`.

Но все это будет в будущем, поэтому за дополнительной информацией обращайтесь к примечаниям к выпуску 3.0. В настоящее время предложено включить в версию 3.0 сценарий (с рабочим названием «2to3»), который кроме всего прочего будет выполнять преобразование инструкций `print` в существующем программном коде в вызовы функции `print`. Он должен помочь вам в переходе с версии 2.x на версию 3.x (как и эта книга!).

В заключение

В этой главе мы начали детальный обзор инструкций языка Python, исследовав инструкции присваивания, выражения и `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 форма инструкции `print` с перенаправлением, например, позволяют уменьшить объем программного кода. Кроме того, мы изучили синтаксис имен переменных, приемы перенаправления потоков и различные часто встречающиеся ошибки, которых следует избегать, такие как обратное присваивание переменной значения, возвращаемого методом `append`.

В следующей главе мы продолжим наше знакомство с инструкциями, подробно 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 двинуться 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 последовательности (`A, B, C = 0, 0, 0`) или несколько инструкций присваивания в отдельных строках (`A = 0, B = 0, C = 0`). И последний способ, как было показано 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 непосредственное 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 проблема не проявляется.

3. Метод списков `sort`, как и метод `append`, выполняет изменения непосредственно в объекте и возвращает значение `None`, а не измененный

список. Обратное присваивание переменной `L` приведет к тому, что в нее запишется значение `None`, а не отсортированный список. Как будет показано далее в этой части книги, недавно в языке появилась новая функция `sorted`, которая выполняет сортировку любых последовательностей и возвращает новый список с результатами сортировки. Поскольку в этом случае изменения производятся не в самом объекте, есть смысл сохранить возвращаемое значение.

4. Можно вручную перенаправить объект `sys.stdout`, открыв файл перед вызовом инструкции `print`, или использовать расширенную форму инструкции `print >> file` для вывода в файл единственной инструкцией. Можно также перенаправить весь выводимый программой текст в файл с помощью синтаксических конструкций системной командной оболочки, но этот вариант к языку Python никакого отношения не имеет.

12

Условная инструкция if

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 более подробно рассмотрим общие концепции, на которых покоится синтаксическая модель инструкций в языке Python. А так как условная инструкция вводит понятие проверки, мы также будем иметь дело с логическими выражениями и остановимся на некоторых деталях относительно проверок истинности вообще.

Условные инструкции if

Если говорить простым языком, в Python инструкция `if` выбирает, какое действие 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 другие условные инструкции `if`. Фактически Python позволяет комбинировать инструкции в программные последовательности (чтобы они выполнялись одна за другой) и в произвольно вложенные конструкции (которые выполняются только при соблюдении определенных условий).

Общая форма

Условная инструкция `if` в языке Python – это типичная условная инструкция, которая присутствует в большинстве процедурных языков программирования. 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 инструкций, с отступом относительно основной инструкции. Во время выполнения условной инструкции `if` интерпретатор выполняет блок инструкций, ассоциированный 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

С целью демонстрации инструкции `if` в действии рассмотрим несколько простых примеров. Все части этой инструкции, за исключением основной части `if` с условным выражением и связанных с ней инструкций, являются необязательными. В простейшем случае остальные части инструкции опущены:

```
>>> if 1:
...     print 'true'
...
true
```

Обратите внимание, что приглашение к вводу изменяется на `...` для строк продолжения 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 строки 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!
```

Эта многострочная инструкция простирается от строки `if` до конца блока `else`. При выполнении этой инструкции интерпретатор выполнит вложенные инструкции 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 последовательности проверок `if/elif`, как в предыдущем примере, либо индексированием словарей, либо поиском в списках. Поскольку словари и списки могут создаваться 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 множественное ветвление через индексирование по ключу `choice` для выбора одного из нескольких 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 ситуации, когда не найдено ни одного совпадения. Как было показано в главе 8, значение по умолчанию при использовании словарей может быть получено с помощью методов `has_key` и `get` или при перехвате исключения. Те же самые приемы могут использоваться и здесь – для определения действия по умолчанию в случае реализации множественного ветвления на базе словаря. Ниже приводится пример с использованием метода `get` для получения значения по умолчанию:

```
>>> branch = {'spam': 1.25,  
...           'ham': 1.99,  
...           'eggs': 0.99}  
>>> print branch.get('spam', 'Bad choice')  
1.25  
>>> print branch.get('bacon', 'Bad choice')  
Bad choice
```

Словари хорошо подходят для выбора значений, ассоциированных с ключами, но как быть в случае более сложных действий, которые можно запрограммировать 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 играют роль значений в словаре, часто создаются как лямбда-функции и вызываются добавлением круглых скобок. Подробнее об этом вы прочитаете далее.

Множественное ветвление на базе словаря довольно удобно использовать в программах, которые имеют дело с динамическими данными, однако большинство программистов согласятся, что использование инструкции `if` является наиболее простым способом организации множественного ветвления. Обычно, при колебаниях при выборе того или иного подхода предпочтение следует отдавать более простому и более удобочитаемому.

Синтаксические правила языка Python

Первое знакомство с синтаксической моделью языка Python состоялось в главе 10. Теперь, когда мы подошли к таким крупным инструкциям, как `if`, настало время пересмотреть и дополнить сведения о синтаксисе, введенные ранее. Вообще язык программирования Python обладает простым синтаксисом, основанным на применении инструкций. Однако он обладает некоторыми особенностями, о которых вам необходимо знать:

- **Инструкции выполняются последовательно, одна за другой, пока не будет предусмотрено что-то другое.** Обычно интерпретатор выполняет инструкции в файле или в блоке от начала до конца, но та-

кие инструкции, как `if` (и, как будет показано далее, циклы), заставляют интерпретатор выполнять переходы внутри программного кода. Так как путь интерпретатора Python через текст программы называется *поток 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 принадлежность инструкций к вложенному блоку определяется по величине отступов. Так же инструкции в языке Python обычно не завершаются точкой с запятой; обычно признаком конца инструкции служит конец строки с этой инструкцией.
- **Составные инструкции = заголовок, «:», инструкции с отступами.** Все составные инструкции в языке Python оформляются одинаково: строка 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 инструкции `if` предложения `elif` и `else` являются не только частями инструкции `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). Пробелы внутри инструкций и выражений 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 отображаются специализированными инструментами.** В языке Python поддерживается дополнительная форма комментариев, называемая строками документирования, которые в отличие от комментариев, начинающихся с `#`, сохраняются для просмотра во время выполнения. Строки документирования – это обычные строки, которые располагаются в начале файлов с программами и в некоторых инструкциях. Интерпретатор игнорирует их содержимое, но они автоматически присоединяются к объектам во время выполнения и могут отображаться инструментами доступа к документации. Строки документирования являются частью стратегии документирования в языке Python и будут рассматриваться в последней главе этой части книги.

Как уже говорилось ранее, в языке Python отсутствует необходимость объявлять типы переменных – только один этот факт упрощает синтаксис языка больше, чем любые другие особенности. Но для большинства новых пользователей самой необычной особенностью синтаксиса языка Python кажется отсутствие фигурных скобок и точек с запятой,

используемых в качестве разделителей блоков и инструкций во многих других языках, поэтому давайте рассмотрим их поближе.

Разделители блоков

Интерпретатор автоматически определяет границы блоков по величине отступов – т. е. 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 выстроены 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 программного кода:

```
x = 1
if x:
    y = 2
    if y:
        print 'block2'
    print 'block1'
print 'block0'
```

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 вложенной инструкции if) имеет отступ из восьми пробелов.

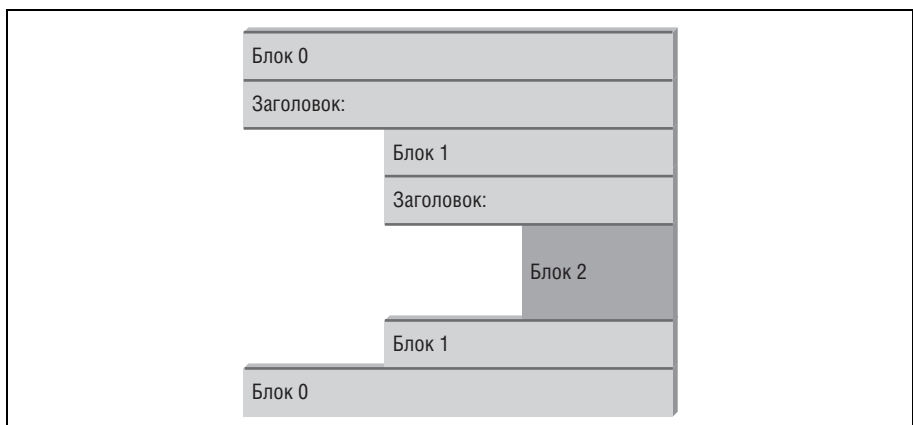


Рис. 12.1. Вложенные блоки программного кода: вложенный блок начинается с инструкции, имеющей больший отступ, и заканчивается либо когда будет встречена инструкция с меньшим отступом, либо в конце файла

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, главное, чтобы все инструкции в одном блоке имели одинаковые отступы. То есть для интерпретатора неважно, *как* будут оформляться отступы, главное, чтобы оформление было непротиворечивым. С технической точки зрения считается, что символ табуляции смещает текущую позицию в строке на восемь пробелов, но обычно не принято смешивать символы табуляции и пробелы внутри блока – используйте что-то одно – или пробелы, или символы табуляции.

Единственное, где в языке Python пробелы имеют большое значение, – это когда они находятся левее программного кода; в большей части других случаев неважно, есть пробелы или нет. При этом отступы являются частью синтаксиса языка, а не просто оформлением: все инструкции внутри любого заданного блока должны иметь одинаковые отступы, в противном случае интерпретатор сообщит о синтаксической ошибке. Так как вам не требуется явно отмечать начало и конец вложенного блока, различные синтаксические элементы, которые можно найти в других языках, в языке Python не нужны.

Отступы, как часть синтаксической модели, стали важным компонентом обеспечения удобочитаемости в языках структурного программирования, таких как Python. Иногда синтаксис языка Python описывают 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.

Программный код, в котором отступы оформлены непротиворечивым образом, всегда будет соответствовать правилам языка Python. Кроме того, многие текстовые редакторы (включая IDLE) упрощают следование модели отступов, автоматически выравнивая программный код по мере его ввода.

Разделители инструкций

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 нескольких строках:

- **Инструкции могут располагаться в нескольких строках, если они окружены синтаксической парой скобок.** Язык Python позволяет продолжить ввод инструкции в следующей строке, когда содержимое инструкции заключено в пару скобок `()`, `{}` или `[]`. Примерами инструкций, которые могут располагаться в нескольких строках, могут служить выражения в круглых скобках, литералы словарей и списков – инструкция не считается законченной, пока интерпре-

татор Python не встретит строку с закрывающей скобкой `()`, `}` или `]`). Промежуточные строки могут иметь любые отступы и не обязательно должны быть выровнены по вертикали.

- **Инструкции могут располагаться в нескольких строках, если они завершаются символом обратного слеша.** Это несколько устаревшая особенность, 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 круглые скобки для заключения длинных конструкций, символы обратного слеша практически никогда не используются.
- **Литералы строк в тройных кавычках могут располагаться в нескольких строках.** Очень длинные строковые литералы можно разместить в нескольких строках – блоки строк в тройных кавычках, с которыми мы встретились в главе 7, предназначены именно для этих целей.
- **Другие правила.** Существует еще несколько моментов, которые хотелось бы упомянуть. Хотя это и необычно, инструкции можно завершать точкой с запятой – иногда это соглашение используется, чтобы компактно разместить несколько инструкций в одной строке. Кроме того, в любом месте в файле могут присутствовать пустые строки и комментарии. Комментарии (которые начинаются с символа `#`) простираются до конца строки.

Несколько специальных случаев

Ниже показано, как выглядит инструкция при применении правила использования скобок. Конструкции, заключенные в скобки, могут занимать произвольное число строк:

```
L = ["Good",  
     "Bad",  
     "Ugly"]           # Открытая пара скобок может охватывать несколько строк
```

Этот прием также можно использовать с круглыми скобками: выражения, аргументы функций, заголовки функций (глава 15) и т. д. 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 при необходимости 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
```

Еще один специальный случай: в языке Python допускается записывать в одной строке несколько несоставных инструкций (т. е. инструкций, которые не имеют вложенных инструкций), разделяя их точками с запятой. Некоторые 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
```

И, наконец, Python позволяет располагать тело составной инструкции в одной строке с заголовком при условии, что тело образует простая (несоставная) инструкция. Вам часто придется видеть следующий вариант использования простых инструкций с единственным условием и действием:

```
if 1: print 'hello' # Простая инструкция в строке заголовка
```

Некоторые из этих специальных случаев можно комбинировать между собой, чтобы писать программный код, который будет сложно читать, но я не рекомендую поступать так – старайтесь записывать по одной инструкции в строке и выравнивайте все блоки, кроме простейших. 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 несколько отличаются от аналогичных операторов в таких языках как C. В языке Python:

- Любое число, не равное нулю, или непустой объект интерпретируется как истина.
- Числа, равные нулю, пустые объекты и специальный объект `None` интерпретируются как ложь.
- Операции сравнения и равенства применяются к структурам данных рекурсивно.
- Операции сравнения и равенства 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 результатов 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 равенства, сравнение с диапазоном значений и т. д.). 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 несколько примеров, чтобы понять, как они работают:

```
>>> 2 < 3, 3 < 2      # Менее чем: возвращает 1 или 0
(True, False)
```

Операция сравнения величин, как в данном случае, возвращает в качестве результата значение True или False, которые, как мы узнали в главах 5 и 9, в действительности 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 результат (не забывайте, что каждый объект может интерпретироваться как истина 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 будет найден первый объект, имеющий истинное значение. Это обычно называют *вычислением по короткой схеме*, так как конечный результат становится известен 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 { }
{ }
```

В первой строке предыдущего примера оба операнда (2 и 3) имеют истинные (т. е. ненулевые) значения, поэтому интерпретатор всегда бу-

дет останавливать вычисления и возвращать операнд слева. В других двух операциях левый операнд 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 станет 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 интерпретатор вычислит оба операнда и вернет объект справа. Во второй проверке левый операнд имеет ложное значение (`[ ]`), поэтому интерпретатор останавливает 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 простое 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 использоваться программистами на языке Python.

Трехместное выражение `if/else`

Одна из основных ролей логических операторов в языке Python заключается в образовании выражений, которые выполняются так же, как условная инструкция `if`. Рассмотрим следующую инструкцию, которая записывает в `A` значение `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 конст-

рукцию внутрь другой инструкции вместо того, чтобы выполнять присваивание переменной. По этим причинам (и, откровенно говоря, потому что в языке С имеется похожая возможность¹) в версии Python 2.5 появилась новая конструкция, позволяющая записать те же действия в виде единственного выражения:

```
A = Y if X else Z
```

Данное выражение дает тот же результат, что и предыдущая инструкция if, но выглядит она проще. Как и в предыдущей инструкции, интерпретатор выполняет выражение Y, только если проверка if X дает истинное значение, а выражение Z выполняется, только если проверка if X дает ложное значение. 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 объект 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 получаем: «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, необходимость 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 выражение дает похожий эффект – благодаря тому, что функция bool

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 С. Такой порядок был выбран в соответствии с анализом наиболее распространенных шаблонов программирования в языке Python, а также отчасти, чтобы помочь бывшим программистам на языке С избавиться от привычки злоупотреблять этим выражением. Не забывайте, что в языке Python исповедуется принцип – чем проще, тем лучше.

преобразует X в соответствующее целое число 1 или 0, которое затем используется для выбора требуемого значения из списка:

```
A = [Z, Y][bool(X)]
```

Придется держать в уме: логические значения

В языке Python часто используется прием выбора одного объекта из множества, основанный на необычном поведении логических операторов. 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 окажутся пустыми. Этот прием стал возможен благодаря тому, что оператор `or` возвращает один из двух объектов и, как оказывается, это весьма распространенная парадигма программирования на языке Python: чтобы выбрать непустой объект из фиксированного множества, достаточно просто объединить их в выражение с помощью оператора `or`.

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 вычисления по сокращенной схеме:

```
if f1() or f2(): ...
```

В данном случае, если функция $f1$ вернет истинное (или непустое) значение, интерпретатор никогда не вызовет функцию $f2$. Чтобы гарантировать вызов обеих функций, можно вызвать их до применения оператора `or`:

```
tmp1, tmp2 = f1(), f2()
if tmp1 or tmp2: ...
```

Вы уже видели другой вариант использования такого поведения: благодаря особенностям работы логических операторов выражение $((A \text{ and } B) \text{ or } C)$ может использоваться для имитации инструкции `if/else`.

Кроме того, обратите внимание: вследствие того, что все объекты могут расцениваться как истинные или ложные значения, в языке Python легко 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.

Например:

```
>>> ['f', 't'][bool('')]
'f'
>>> ['f', 't'][bool('spam')]
't'
```

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 комбинации `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).

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 операциями 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 рассмотрим 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 выражения?
3. Как можно разместить одну инструкцию в нескольких строках?

4. Что означают слова `True` и `False`?

Ответы

1. Самый простой, хотя и не самый краткий способ организации множественного ветвления заключается в использовании инструкции `if` с несколькими частями `elif`. Нередко того же результата можно добиться с помощью операции индексации в словаре, особенно если учесть, что словари могут содержать функции, созданные с помощью инструкций `def` или выражений `lambda`.
2. В версии Python 2.5 выражение `Y if X else Z` возвращает `Y`, если `X` имеет истинное 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 считаться законченной, когда интерпретатор обнаружит правую закрывающую скобку.
4. `True` и `False` – это всего лишь версии целых чисел `1` и `0`, соответственно. В языке Python они обозначают истинные и ложные значения.

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 последовательность 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.

В языке Python существуют и другие способы организации циклов, но две инструкции, которые описываются здесь, являются основными синтаксическими элементами, предоставляющими возможность программирования повторяющихся действий. Кроме того, мы попутно рассмотрим необычные инструкции (такие как `break` и `continue`), потому что они используются внутри циклов. Дополнительно в этой главе мы исследуем родственные концепции протокола итераций в языке Python и рассмотрим некоторые подробности о генераторах списков – близких родственниках цикла `for`.

Циклы `while`

Инструкция `while` является самой универсальной конструкцией организации итераций в языке Python. Проще говоря, она продолжает выполнять блок инструкций (обычно с отступами) до тех пор, пока условное выражение продолжает 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, расположенной сразу же за вложенным блоком тела цикла `while`. В результате тело цикла продолжает выполняться снова и снова, пока условное выражение возвраща-

ет истинное выражение, а если условное выражение сразу вернет ложное значение, тело цикла никогда не будет выполнено.

Как я только что заметил, инструкция `while` — это одна из двух инструкций циклов в языке Python, кроме нее существует еще инструкция `for`. Помимо этих инструкций в языке Python существует несколько способов неявной организации циклов (итераций): функции `map`, `reduce` и `filter`; оператор `in` проверки на вхождение; генераторы списков и многие другие. Некоторые из них мы рассмотрим в главе 17, потому что они имеют отношение к функциям.

Общий формат

В своей наиболее сложной форме инструкция `while` состоит из строки заголовка с условным выражением, тела цикла, содержащего одну или более инструкций 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 обычная практика — проверка истинности объектов осуществляется непосредственно вместо использования более растянутого эквивалента (`while x != ''`). Далее в этой главе мы рассмотрим другие способы обхода элементов строки с помощью цикла `for`. Обратите внимание на завершающий символ запятой в инструкции `print` — как рассказывалось в главе 11, такой прием обеспечивает вывод значений в одну строку:

```
>>> x = 'spam'
>>> while x:          # Пока x не пустая строка
...     print x,
...     x = x[1:]     # Вырезать первый символ из x
...
spam pam am m
```

Следующий фрагмент перебирает значения от *a* до *b*, не включая значение *b*. Ниже мы рассмотрим более простой способ выполнения 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,      # Or, a = a + 1
...     a += 1
...
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 имитировать, 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 перейти к следующему разделу и поближе познакомиться с инструкцией *break*.

break, continue, pass и else

Теперь, когда мы познакомились с несколькими циклами в действии, настало время обратить внимание на две простые инструкции, которые могут использоваться 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*.

break

Производит переход за пределы объемлющего цикла (всей инструкции цикла).

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 использования 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 цикла `while` (или `for`), но как правило, они используются в условных инструкциях `if`, чтобы выполнить необходимое действие в ответ на некоторое условие.

Примеры

Обратимся к нескольким простым примерам, чтобы увидеть, как эти инструкции 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, если потребуется создать бесконечный цикл, который ничего не делает, сделать это можно будет 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 программа (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 смыслом, например, 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                                # Настоящий программный код будет добавлен позже  
  
def func2():  
    pass
```

continue

Инструкция `continue` вызывает немедленный переход в начало цикла. Она иногда позволяет избежать использования вложенных инструкций. В следующем примере инструкция 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 кратные 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,
```

Так как инструкция `continue` выполняет переход в начало цикла, нам не потребовалось вкладывать инструкцию `print` в инструкцию `if` — она будет задействована, только если инструкция `continue` не будет выполнена. Если она напоминает вам инструкцию «`goto`», имеющуюся в других языках, то это справедливо. 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 `continue`. Не злоупотребляйте использованием этой инструкции, особенно когда вы только начинаете работать с языком Python. Например, последний пример выглядел 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,
```

break

Инструкция `break` вызывает немедленный выход из цикла. Так как программный код, следующий в цикле за этой инструкцией, не вы-

полняется, если эта инструкция запущена, то ее также можно использовать, чтобы избежать вложения. Например, ниже приводится простой интерактивный цикл (вариант более крупного примера, рассматривавшегося в главе 10), где производится ввод данных с помощью функции `raw_input` и выход из цикла, если в ответ на запрос имени будет введена строка «stop»:

```
>>> while 1:
...     name = raw_input('Enter name:')
...     if name == 'stop': break
...     age = raw_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 `raw_input` возвращает ввод пользователя в виде строки. В главе 29 вы увидите, что функция `raw_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 инструкцией `try`.

else

При объединении с частью `else` инструкция `break` часто позволяет избавиться от необходимости сохранять флаг состояния поиска, как это делается в других 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 = 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 делитель. При такой реализации в блоке `else` цикла, который

будет выполнен, только если инструкция `break` не была запущена, можно с уверенностью сказать, что число является простым.¹

Блок `else` цикла выполняется также в том случае, когда тело цикла ни разу не выполнялось, поскольку в этой ситуации инструкция `break` также не выполняется. В циклах `while` это происходит, когда первая же проверка условия в заголовке дает ложное значение. Вследствие этого в предыдущем примере будет получено сообщение «`is prime`» (простое число), если изначально `x` меньше или равно 1 (т. е., когда `y` равно 2).

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 начинает 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 необходимости 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 значение. 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 определить, увенчался поиск успехом или нет. Это вполне корректный программный код для языка Python, и он работает, однако это именно тот случай, когда можно использовать блок `else` в цикле. Ниже приводится эквивалентный фрагмент:

```
while x:                          # Выйти, когда x опустеет
    if match(x[0]):
```

¹ Более или менее. Числа, меньшие 2, не считаются простыми в соответствии со строгим математическим определением. Если быть более точным, этот программный код также будет терпеть неудачу при отрицательных значениях и при использовании чисел с плавающей точкой, и в будущем утратит работоспособность из-за грядущих изменений в поведении оператора `/`, который будет выполнять операцию «истинного деления», как описано в главе 5. Если вы захотите поэкспериментировать с этим фрагментом, загляните в упражнения к четвертой части книги, где этот пример обернут в функцию.

```
        print 'Ni'
        break          # Выход, в обход блока else
    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 инструкцию `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` – темой следующего раздела, потому что обход последовательностей выполняется неподконтрольно вам.

Циклы for

Цикл `for` является универсальным итератором последовательностей в языке Python: он может выполнять обход элементов в любых упорядоченных объектах последовательностей. Инструкция `for` способна работать со строками, списками, кортежами, с другими встроенными объектами, поддерживающими возможность выполнения итераций, и с новыми объектами, которые создаются с помощью классов, как будет показано позже.

Общий формат

Циклы `for` в языке Python начинаются со строки заголовка, где указывается переменная для присваивания (или – цель), а также объект, обход которого будет выполнен. Вслед за заголовком следует блок (обычно с отступами) инструкций, 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, присваивает элементы объекта последовательности переменной цикла и выполняет тело цикла для каждого из них. Для обращения к текущему элементу последовательности 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 присвоен следующий элемент последовательности, когда управление вернется в начало цикла. После выхода из цикла эта переменная обычно все еще ссылается на последний элемент последовательности, если цикл не был завершен инструкцией `break`.

Инструкция `for` также поддерживает необязательную часть `else`, которая работает точно так же, как и в циклах `while`, — она выполняется, если выход из цикла производится не инструкцией `break` (т. е., если в цикле был выполнен обход всех элементов последовательности). Инструкции `break` и `continue`, представленные выше, в циклах `for` работают точно так же, как и в циклах `while`. Полная форма цикла `for` имеет следующий вид:

```
for <target> in <object>: # Связывает 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'
```

Придется держать в уме: имитация циклов `while` языка C

В разделе главы 11, где рассматривались инструкции выражений, утверждалось, что язык Python не предусматривает возможность выполнять 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 выражение. 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 циклах `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 = 1
while x:
    x = next()
    if x:
        ...обработка x...
```

или вынести первое присваивание за пределы цикла:

```
x = next()
while x:
    ...обработка x...
    x = next()
```

Из этих трех вариантов первый, как могут полагать некоторые, — наименее структурированный, но он же представляется наиболее простым и наиболее часто используемым. (Простейший цикл for в языке Python также может заменить некоторые циклы языка C.)

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 поочередно, слева направо, присвоим переменной x каждый из трех элементов списка и выведем каждый из них с помощью инструкции print. Внутри инструкции print (в теле цикла), имя x ссылается на текущий элемент списка:

```
>>> for x in ["spam", "eggs", "ham"]:
...     print x,
...
spam eggs ham
```

Как отмечалось в главе 11, завершающая запятая в инструкции print подавляет вывод символа конца строки, в результате все элементы выводятся в одной строке.

В следующих двух примерах вычисляется сумма и произведение всех элементов в списке. В этой главе и далее в книге мы познакомимся с инструментами, которые применяют такие операции, как `+` и `*`, к элементам списка автоматически, 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 любым последовательностям. Например, цикл `for` может применяться к строкам и кортежам:

```
>>> S = "lumberjack"
>>> T = ("and", "I'm", "okay")
>>> for x in S: print x,          # Обход строки
...
l u m b e r j a c k

>>> for x in T: print x,         # 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!

Присваивание кортежа в цикле for

Если выполнить обход последовательности кортежей, переменная цикла сама фактически будет кортежем. Это лишь еще один случай операции присваивания кортежа. Не забывайте, что инструкция цикла `for` присваивает элементы объекта последовательности переменной цикла, а операция присваивания везде 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 сходно инструкции $(a, b) = (1, 2)$, второй проход – инструкции $(a, b) = (3, 4)$ и т. д. Это не какой-то особый случай – синтаксически после слова `for` выполняется присваивание переменной цикла любого вида.

Вложенные циклы for

Теперь рассмотрим нечто более сложное. Следующий пример иллюстрирует использование блока `else` в цикле `for` и вложенные инструкции. Имея список объектов (`items`) и список ключей (`tests`), этот фрагмент пытается отыскать каждый 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"
...             break
...     else:
...         print key, "not found!"
...
(4, 5) was found
3.14 not found!
```

Поскольку вложенная инструкция `if` вызывает инструкцию `break`, когда совпадение обнаружено, то 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 внутренний будет выполнять обход списка элементов в поисках каждого ключа. Уровень вложенности блока `else` имеет большое значение – он находится на уровне строки заголовка внутреннего цикла `for`, поэтому он соответствует внутреннему циклу (не инструкции `if` и не внешнему циклу `for`).

Примечательно, что этот пример можно упростить, если использовать оператор `in` для проверки вхождения ключа. Поскольку оператор `in` неявно выполняет 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 обработки данных – выборка одинаковых элементов из двух последовательностей (из строк). Это достаточно простая задача поиска пересечения двух множеств. После 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 переменными: `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.

Итераторы: первое знакомство

В предыдущем разделе упоминалось, что цикл `for` может работать с последовательностями любого типа в языке Python, включая списки, кортежи и строки, например:

```
>>> for x in [1, 2, 3, 4]: print x ** 2,
...
1 4 9 16

>>> for x in (1, 2, 3, 4): print x ** 3,
...
1 8 27 64

>>> for x in 'spam': print x * 2,
...
ss pp aa mm
```

Фактически цикл `for` имеет еще более универсальную природу, чем было показано, – он способен работать с любыми объектами, поддерживающими возможность выполнения итераций. На самом деле это верно для всех средств выполнения итераций, которые выполняют сканирование объектов слева направо, включая циклы `for`, генераторы списков, оператор `in` проверки на вхождение и встроенную функцию `map`.

Понятие «итерируемого объекта» является относительно новым в языке Python. По существу оно является обобщением понятия последовательности — объект считается итерируемым либо если он физически является последовательностью, либо если он является объектом, который воспроизводит по одному результату за раз в контексте инструментов выполнения итераций, таких как цикл `for`. В некотором смысле в категорию итерируемых объектов входят как физические после-

Придется держать в уме: сканирование файлов

Вообще циклы удобно использовать везде, где необходимо повторно выполнять некоторые действия или многократно обрабатывать данные. Файлы содержат множество символов и строк, поэтому они могут рассматриваться как один из типичных объектов применения циклов. Чтобы просто загрузить содержимое файла в строку одной инструкцией, достаточно вызвать метод `read`:

```
file = open('test.txt', 'r')
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
```

Здесь цикл `for` выполняет обработку каждого отдельного символа, но загрузка содержимого файла в память производится однократно. Чтение строками или блоками с помощью цикла `while` можно реализовать следующим образом:

```
file = open('test.txt')
while True:
    line = file.readline()  # Читать строку за строкой
    if not line: break
    print line,

file = open('test.txt', 'rb')
while True:
    chunk = file.read(10)   # Читать блоками по 10 байтов
    if not chunk: break
    print chunk,
```

Однако при использовании цикла `for` реализация построчного чтения выглядит проще и работает быстрее:

```
for line in open('test.txt').readlines():
    print line

for line in open('test.txt').xreadlines():
    print line

for line in open('test.txt'):
    print line
```

Функция `readlines` загружает файл целиком в список строк, тогда как функция `xreadlines` загружает очередную строку по требованию, благодаря чему исключается вероятность переполнения памяти при попытке загрузить большой файл. Последний пример основан на использовании файлового итератора, который является эквивалентом использования функции `xreadlines` (итераторы будут рассматриваться в следующем разделе). Имя `open` во всех примерах выше можно также заменить именем `file`, начиная с версии Python 2.2. Более подробную информацию о методах, использованных здесь, вы найдете в руководстве по библиотеке. Как правило, чем больше данных читается на каждом шаге, тем быстрее работает ваша программа.

довательности, так и последовательности виртуальные, которые вычисляются по требованию.

Итераторы файлов

Один из самых простых способов понять, что такое итераторы, – это посмотреть, как они работают со встроенными типами, такими как файлы. Напомню, 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 достижении конца файла возвращается пустая строка, что может служить сигналом для выхода из цикла:

```
>>> f = open('script1.py')
>>> f.readline()
'import sys\n'
>>> f.readline()
'print sys.path\n'
>>> f.readline()
'x = 2\n'
>>> f.readline()
'print 2 ** 33\n'
>>> f.readline()
''
```

Теперь файлы имеют также метод `next`, который производит практически тот же эффект – всякий раз, когда его вызывают, он возвращает следующую строку. Единственное значимое различие состоит в том, что по достижении конца файла метод `next` возбуждает встроенное исключение `StopIteration` вместо того, чтобы возвращать пустую строку:

```
>>> f = open('script1.py')
>>> f.next()
'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):
File "<pyshell#330>", line 1, in <module>
f.next()
StopIteration
```

Такое поведение в точности соответствует тому, что мы в языке Python называем *итерационным протоколом*, – объект реализует метод `next`, который возбуждает исключение `StopIteration` в конце серии результатов. Любой такой объект в языке Python считается итерируемым. 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 текстового файла, как уже упоминалось в главе 9, состоит не в том, чтобы прочитать его целиком, а в том, чтобы позволить циклу `for` автоматически вызывать метод `next` для перемещения к следующей строке в каждой итерации. Например, следующий фрагмент читает содержимое файла строку за строкой (попутно 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(),
...
IMPORT SYS
PRINT SYS.PATH
X = 2
PRINT 2 ** 33
```

Такой способ построчного чтения текстовых файлов 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, чтобы вызвать метод `readlines` для загрузки содержимого файла в память в виде списка строк:

```
>>> for line in open('script1.py').readlines():
...     print line.upper(),
...
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 подхода к использованию памяти. Из-за того что в этом случае файл загружается целиком, этот способ не позволит работать с файлами, слишком большими, чтобы поместиться в память компьютера. При этом версия, основанная на применении итераторов, не подвержена таким проблемам с памятью, так как содержимое файла считывается по одной строке за раз. Более того, итераторы были существенно оптимизированы, поэтому способ на базе итераторов должен иметь более высокую производительность.

Как упоминалось во врезке «Придется держать в уме: сканирование файлов», 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(),
...
...ВЫВОД ТОТ ЖЕ САМЫЙ...
```

Однако такой вариант наверняка будет работать медленнее версии, основанной на использовании итератора в цикле `for`, потому что итераторы внутри интерпретатора выполняются со скоростью, присущей программам, написанным на языке C, тогда как версия на базе цикла `while` работает со скоростью интерпретации байт-кода виртуальной машиной Python. Всякий раз, когда код на языке Python подменяется кодом на языке C, скорость его выполнения увеличивается.

Другие итераторы встроенных типов

С технической точки зрения итерационный протокол имеет еще одну сторону. В самом начале цикл `for` получает итератор из итерируемого объекта, передавая его встроенной функции `iter`, которая возвращает объект, имеющий требуемый 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):
  File "<pyshell#343>", line 1, in <module>
    I.next()
StopIteration
```

Кроме файлов и фактических последовательностей, таких как списки, удобные 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 вообще не обязательно использовать метод `keys` — словари имеют итератор, который автоматически возвращает по одному ключу за раз в контексте итераций, поэтому больше не требуется создавать в памяти сразу полный список ключей. В результате применения этого итератора скорость выполнения возрастает, память используется экономнее, а программный код выглядит проще:

```
>>> for key in D:
...     print key, D[key]
...
a 1
c 3
b 2
```

Другие контексты итераций

До настоящего момента я демонстрировал итераторы в контексте инструкции цикла `for`, которая является одной из основных обсуждаемых тем этой главы. Однако, имейте в виду, что каждый инструмент, который выполняет 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(),
...
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 `sum`, также основаны на применении итерационного протокола:

```
>>> 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'))
['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

>>> sorted(open('script1.py'))
['import sys\n', 'print 2 ** 33\n', 'print sys.path\n', 'x = 2\n']
```

Используемая здесь функция `map`, которую мы будем рассматривать в следующей части книги, представляет собой инструмент, вызывающий заданную функцию для каждого элемента итерируемого объекта, напоминая тем самым генераторы списков, хотя и с ограниченными возможностями, потому что ей можно передать только функцию и нельзя указать произвольное выражение. Так как генераторы списков связаны с циклами `for`, мы займемся их исследованием ниже в этой главе, а затем еще раз вернемся к ним в следующей части книги.

Функцию `sorted`, задействованную здесь, мы уже видели в работе в главе 4. Функция `sorted` — это относительно новая встроенная функция, которая использует итерационный протокол, — она напоминает метод списка `sort`, но в качестве результата возвращает новый отсортированный список и способна работать с любым итерируемым объектом. Существуют и другие, более новые встроенные функции, поддерживающие итерационный протокол. Например, функция `sum` вычисляет сумму всех чисел в любом итерируемом объекте, а встроенные функции `any` и `all` возвращают `True`, если любой (`any`) или все (`all`) элементы итерируемого объекта являются истинными значениями, соответственно:

```
>>> sorted([3, 2, 4, 1, 5, 0]) # Другие контексты итераций
[0, 1, 2, 3, 4, 5]
>>> sum([3, 2, 4, 1, 5, 0])
15
>>> any(['spam', '', 'ni'])
True
>>> all(['spam', '', 'ni'])
False
```

Интересно, что область влияния итерационного протокола в языке Python в настоящее время гораздо шире, чем было продемонстрировано

в примерах – любые встроенные инструменты в языке Python, которые выполняют обход объектов 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 итерируемых объектов), строковый метод `join` (который вставляет подстроку между строками, содержащимися в итерируемом объекте) и даже операция присваивания 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')
```

Итераторы, определяемые пользователем

Итераторы мы еще будем рассматривать в главе 17, вместе с функциями, и в главе 24, когда будем изучать классы. Как будет показано позднее, существует возможность превратить пользовательскую функцию в итерируемый объект с помощью инструкции `yield`. Генераторы списков также поддерживают протокол с помощью выражений-генераторов, а пользовательские классы можно сделать итерируемыми за счет методов перегрузки операторов `__iter__` и `__getitem__`. Итераторы, определяемые пользователем, позволяют использовать любые объекты и операции в любом из итерационных контекстов, с которыми мы встретились здесь.

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 последовательности. Однако существуют такие ситуации, когда необходимо выполнять обход каким-то особым способом. Например, как быть, когда необходимо выполнить обход каждого второго или каждого третьего элемента в списке, или попутно выполнить изменения в списке? Или когда необходимо реализовать параллельный обход более чем одной последовательности в одном и том же цикле `for`?

Такие уникальные ситуации всегда можно запрограммировать с помощью цикла `while` и извлечения элементов вручную, но Python предос-

твляет две встроенные возможности, позволяющие управлять обходом элементов в цикле `for`:

- Встроенная функция `range` возвращает список последовательно увеличивающихся целых чисел, которые можно использовать в качестве индексов внутри цикла `for`.¹
- Встроенная функция `zip` возвращает список кортежей, составленных из элементов входных списков с одинаковыми индексами, который может использоваться 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 пользоваться такими инструментами, которые позволят использовать цикл `for`. Рассмотрим каждый из этих встроенных инструментов по очереди.

Счетные циклы: `while` и `range`

Функция `range` является по-настоящему универсальным инструментом, который может использоваться в самых разных ситуациях. Чаще всего она используется для генерации индексов в цикле `for`, но вы можете использовать ее везде, где необходимы списки целых чисел:

```
>>> range(5), range(2, 5), 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 диапазоне 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 последующего значения (по умолчанию шаг равен 1). Существует возможность воспроизводить последовательности чисел в диапазоне отрицательных значений и в порядке убывания:

```
>>> range(-5, 5)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

¹ Современные версии Python предоставляют также встроенную функцию `xrange`, которая генерирует индексы по одному вместо того, чтобы создавать их сразу все и сохранять в списке, как это делает функция `range`. Функция `xrange` не дает никакого преимущества в скорости, но она удобна в смысле экономии памяти, если приходится генерировать огромное число значений. На момент написания этих строк ходят слухи, что в версии Python 3.0 функция `xrange` исчезнет, а функция `range` будет преобразована в объект-генератор, поддерживающий итерационный протокол для воспроизведения одного элемента за раз. Дополнительную информацию вы найдете в примечаниях к выпуску Python 3.0.


```
>>> range(5, -5, -1)
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```

Такое использование функции `range` само по себе может быть полезным, однако 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 соответствующего количества целых чисел:

```
>>> for i in range(3):
...     print i, 'Pythons'
...
0 Pythons
1 Pythons
2 Pythons
```

Функция `range` также часто используется для косвенного обхода последовательностей. Самый простой и самый быстрый способ выполнить обход последовательности заключается в использовании цикла `for`, когда основную работу выполняет интерпретатор :

```
>>> X = 'spam'
>>> for item in X: print item,      # Простейший цикл
...
s p a m
```

При таком использовании все задачи, касающиеся выполнения итераций, решаются внутренними механизмами цикла `for`. Если вам действительно необходимо явно управлять логикой доступа к элементам, вы можете использовать цикл `while`:

```
>>> i = 0
>>> while i < len(X):              # Обход с помощью цикла while
...     print X[i],; i += 1
...
s p a m
```

Однако управлять индексами вручную можно и в цикле `for`, если использовать функцию `range` для воспроизведения списка индексов:

```
>>> X
'spam'
>>> len(X)                          # Длина строки
4
>>> range(len(X))                   # Все допустимые смещения в X
[0, 1, 2, 3]
>>>
>>> for i in range(len(X)): print X[i], # Извлечение элементов вручную
...
s p a m
```

В этом примере выполняется обход списка *смещений* в строке `X`, а не фактических *элементов* строки – нам пришлось внутри цикла обращаться к строке `X`, чтобы извлечь каждый элемент.

Обход части последовательности: range

Последний пример в предыдущем разделе вполне работоспособен, но он выполняется гораздо медленнее, чем мог бы. Кроме того, нам пришлось выполнить больше работы, чем требуется для решения такой задачи. Если вы не предъявляете особых требований к индексам, всегда лучше использовать простейшую форму цикла `for` — используйте цикл `for` вместо `while` везде, где только возможно, 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 элементы:

```
>>> S = 'abcdefghijk'
>>> range(0, len(S), 2)
[0, 2, 4, 6, 8, 10]

>>> for i in range(0, len(S), 2): print S[i],
...
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 т. д. 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, можно использовать расширенную форму операции извлечения среза с тремя 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:

```
>>> for x in S[::2]: print x
... 
```

Изменение списков: range

Еще одно место, где можно использовать комбинацию функции `range` и цикла `for`, — это циклы, изменяющие список в процессе его обхода. Например, предположим, что по тем или иным причинам нам необходимо прибавить 1 к каждому элементу списка. Можно попытаться использовать для этой цели простейшую 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, например, переменная `x` является целым числом 1. На следующей итерации в переменную `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 использовать операцию присваивания по индексу и изменить значения во всех позициях, по которым осуществляется цикл. Необходимые нам индексы можно воспроизвести с помощью комбинации функций `range/len`:

```
>>> L = [1, 2, 3, 4, 5]

>>> for i in range(len(L)):    # Прибавить 1 к каждому элементу в списке L
...     L[i] += 1             # Or 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
...     i += 1
...
>>> L
[3, 4, 5, 6, 7]
```

В данном случае решение на базе функции `range` может быть неидеальным. Генератор списка в виде `[x+1 for x in L]` также даст желаемый результат, но первоначальный список при этом не изменится (мы могли бы присвоить получившийся новый список обратно переменной `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 обход отдельных частей последовательностей. В том же духе встроенная функция `zip` позволяет использовать цикл `for` для обхода нескольких последовательностей *параллельно*. Функция `zip` принимает одну или несколько последовательностей в качестве аргументов и возвращает список кортежей, составленных из соответствующих элементов этих последовательностей. Например, предположим, 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:

```
>>> zip(L1, L2)
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

Такой результат может пригодиться в самых разных ситуациях, но применительно 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`, т. е. пар, составленных из элементов двух списков. Обратите внимание, что в этом цикле используется операция присваивания кортежей для получения элементов 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 элементам производится вручную, но такой цикл будет сложнее в реализации и наверняка медленнее, чем прием, основанный на использовании `for/zip`.

Функция `zip` на самом деле более универсальна, чем можно было бы представить на основе этого фрагмента. Например, она принимает последовательности любого типа (в действительности – любые итерируемые объекты, включая и файлы) и позволяет указывать более двух аргументов:

```
>>> T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)
>>> T3
(7, 8, 9)
>>> zip(T1,T2,T3)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Длина списка, возвращаемого функцией `zip`, равна длине кратчайшей из последовательностей, если аргументы имеют разную длину:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
>>> zip(S1, S2)
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Родственная (и более старая) встроенная функция `map` объединяет элементы последовательностей похожим образом, но она дополняет недостающие элементы значениями `None`, если аргументы имеют разную длину:

```
>>> map(None, S1, S2)
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
```

В этом примере фактически используется простейшая форма обращения к встроенной функции `map`. Обычно она принимает функцию и одну или более последовательностей и собирает результаты вызова функции с соответствующими элементами, извлеченными из последовательностей.

Когда в первом аргументе передается значение `None` (как в данном случае), она просто объединяет элементы подобно функции `zip`. Функция `map` и прочие родственные ей инструменты будут рассматриваться в главе 17.

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 выполнения программы. Теперь, когда мы поближе познакомимся с этой функцией, 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? Например, 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`:

```
>>> zip(keys, vals)
[('spam', 1), ('eggs', 3), ('toast', 5)]

>>> 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 26). Этот вызов производит преобразование списка в словарь, но в действительности это вызов конструктора объекта. Далее в этой главе мы рассмотрим родственное, но более широкое понятие *генераторов списков*, которые позволяют создавать списки с помощью единственного выражения.

Генерирование индексов и элементов: `enumerate`

Ранее мы рассматривали использование функции `range` для генерации индексов (смещений) элементов в строке вместо получения самих элементов с этими индексами. Однако в некоторых программах необходимо получить и то, и другое: и элемент, и его индекс. При традиционном подходе можно было бы использовать 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
```

Функция `enumerate` возвращает *объект-генератор* – разновидность объекта, который поддерживает итерационный протокол, с которым мы встретились выше в этой главе и более подробно будем осуждать в следующей части книги. Он имеет метод `next`, возвращающий кортеж `(index, value)` для каждого элемента списка, который мы можем использовать для присваивания кортежей в цикле `for` (точно так же, как и в случае с функцией `zip`):

```
>>> E = enumerate(S)
>>> E.next()
(0, 's')
>>> E.next()
(1, 'p')
```

Обычно мы не видим всю эту механику, потому что во всех контекстах итераций (включая генераторы списков – тема следующего раздела) итерационный протокол выполняется автоматически:

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'p', 'aa', 'mmm']
```

Генераторы списков: первое знакомство

В предыдущем разделе 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 выражения генераторы списков переводят многое из того, что использовалось раньше,

в разряд устаревших приемов. Например, 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 меньше усилий и, скорее всего, этот вариант работает быстрее. Выражения генераторов 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 достаточно удобная, чтобы заслужить внимательного изучения здесь.

Основы генераторов списков

Впервые с генераторами списков мы встретились в главе 4. Синтаксис генераторов списков происходит от конструкций, используемых в теории множеств для описания операций над каждым элементом множества, но вам совсем необязательно знать теорию множеств, чтобы использовать их. Многие считают, что генераторы списков в языке Python напоминают цикл `for`, записанный задом наперед.

Давайте рассмотрим пример из предыдущего раздела более подробно. 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 итерируемый объект (`for x in L`).

Чтобы найти значение выражения, Python выполняет обход списка `L`, присваивая переменной `x` каждый очередной элемент, и собирает результаты пропускания всех элементов 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 точное представление внутреннего механизма генератора списков.

Но генераторы списков записываются компактнее, и данный способ сборки списков получил широкое распространение в языке Python, поэтому они оказываются очень удобными во многих ситуациях. Более того, генераторы списков могут выполняться значительно быстрее (зачастую почти в два раза), чем инструкции циклов `for`, потому что итерации выполняются со скоростью языка C, а не со скоростью программного кода на языке Python. Такое преимущество в скорости особенно важно для больших объемов данных.

Использование генераторов списков для работы с файлами

Рассмотрим еще один распространенный случай использования генераторов списков, исследуя в деталях их работу. Вспомним, что у объекта файла 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 новой строки (`\n`). Символ новой строки является препятствием для многих программ – приходится быть осторожным, чтобы избежать появления пустых строк при выводе и т. д. Было бы совсем неплохо, если бы мы могли одним махом избавиться от этих символов новой строки.

Всякий раз, когда мы заговариваем о выполнении операций над каждым элементом последовательности, мы попадаем в сферу действий генераторов списков. Например, предположим, что переменная `lines` находится в том же состоянии, 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 выражение извлечения среза `line[:-1]`, но только если бы мы были абсолютно уверены, что все строки завершаются символом новой строки):

```
>>> lines = [line.rstrip() for line in lines]
>>> lines
['import sys', 'print sys.path', 'x = 2', 'print 2 ** 33']
```

Этот метод работает, генераторы списков – это другой итерационный 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 которым 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 именно 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 – интерпретатор 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 генераторов списков может оказаться весьма существенным.

Синтаксис расширенного генератора списков

В действительности генераторы списков могут иметь еще более сложный вид. Например, в цикл `for`, вложенный в выражение, можно добавить оператор `if` для отбора результатов, для которых условное выражение дает истинное значение.

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 эквивалентный простой цикл `for` (вообще любой генератор списков можно перевести 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 ассоциированный с ним оператор `if` (подробнее о синтаксисе генераторов выражений рассказывается в главе 17).

Например, следующий фрагмент создает список результатов операции конкатенации `x+y` для всех `x` в одной строке и для всех `y` – в другой. В результате получаются 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. Следующий 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 простых итераций – для реализации сложных действий более простая инструкция `for` наверняка будет проще и для понимания, и для изменения в будущем. Обычно, если что-то в программировании для вас оказывается слишком сложным, возможно это не самое лучшее решение.

Мы еще вернемся к итераторам и генераторам списков в главе 17, где будем рассматривать их в контексте функций, где вы увидите, что они связаны с функциями не менее тесно, чем с инструкциями циклов.

В заключение

В этой главе мы исследовали инструкции циклов языка Python, а также некоторые концепции, имеющие отношение к циклам. Мы рассмотрели инструкции `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.

Дополнительно мы познакомились с итерационным протоколом в языке Python, обеспечивающим возможность выполнять итерации по объектам, не являющимся последовательностями, и с генераторами списков. Как было показано, генераторы списков, применяющие заданные

выражения ко всем элементам любого итерируемого объекта, напоминают циклы `for`.

Эта глава завершает наш обзор процедурных инструкций. Следующая глава, завершающая эту часть книги, обсуждает возможности документирования программного кода на языке Python. Документация также является частью синтаксической модели, и кроме того, она является важным компонентом хорошо написанных программ. В следующей главе мы также рассмотрим ряд упражнений к этой части книги, прежде чем перенесем свое внимание к более крупным структурам, таким как функции. Однако, как обычно, прежде чем двинуться дальше, попробуйте ответить на контрольные вопросы.

Закрепление пройденного

Контрольные вопросы

1. Когда выполняется блок `else` в циклах?
2. Как в языке Python можно запрограммировать счетный цикл?
3. Как взаимосвязаны циклы `for` и итераторы?
4. Как взаимосвязаны циклы `for` и генераторы списков?
5. Назовите четыре разных контекста итераций в языке Python.
6. Какой способ строчного чтения содержимого файлов считается наиболее оптимальным?
7. Какое оружие вы ожидали бы увидеть в руках испанской инквизиции?

Ответы

1. Блок `else` в циклах `while` или `for` выполняется один раз после выхода из цикла при условии, что цикл завершается обычным образом (без использования инструкции `break`). Инструкция `break` осуществляет немедленный выход из цикла и пропускает блок `else` (если таковой присутствует).
2. 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 просто обойти все элементы в последовательности, везде, где только возможно, используйте простой цикл `for`, без функции `range` или счетчиков. Такая реализация и выглядит проще, и обычно работает быстрее.
3. Цикл `for` использует итерационный протокол для обхода элементов объекта. На каждой итерации он вызывает метод `next` объекта и пе-

рехватывает исключение `StopIteration`, по которому определяет момент окончания итераций.

4. Оба они являются инструментами итераций. Генераторы списков представляют простой и эффективный способ выполнения задачи, типичной для циклов `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 напоминают заголовок инструкции `for`.
5. В число итерационных контекстов языка Python входят: цикл `for`, генераторы 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 протоколу (метод `next`) для обхода итерируемых объектов.
6. Рекомендуемый в настоящее время способ чтения строк из текстового файла – 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 автоматически извлекать по одной строке из файла с помощью метода `next`. Такой подход считается более оптимальным в смысле простоты программирования, скорости выполнения и использования памяти.
7. Любой из следующих вариантов я приму как правильный ответ: устрашение, шантаж, хорошие красные униформы, удобная кушетка и мягкие подушки.

14

Документация

Эта глава завершает третью часть книги изучением приемов и инструментов, используемых для документирования программного кода на языке Python. Несмотря на то, что программный код Python изначально обладает высокой удобочитаемостью, некоторый объем уместно расположенных, внятных комментариев может существенно облегчить другим людям понимание принципа действия ваших программ. Язык Python включает синтаксические конструкции и инструменты, 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 отношение к синтаксической модели языка Python, а отчасти как источник сведений для тех, кто из всех сил пытается понять возможности языка Python. Преследуя эту последнюю цель, я дополню указания о документировании, которые были даны в главе 4. Как обычно, эта глава завершается предупреждениями о наиболее часто встречающихся ловушках, контрольными вопросами к главе и упреждениями к этой части книги.

Источники документации в языке Python

К настоящему моменту вы уже наверняка начинаете понимать, что Python изначально включает в себя удивительно широкие функциональные возможности – встроенные функции и исключения, предопределенные атрибуты и методы 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 рассказывает о различных источниках документации, доступных в языке Python. Здесь также будут представлены строки документирования (docstrings) и система *PyDoc*, которая использует их. Эти темы мало связаны с самим языком программирования, но они будут иметь большое значение, как только вы подойдете к примерам и упражнениям в этой части книги.

Как показано в табл. 14.1, существует множество мест, где можно отыскать информацию о Python, и объем этой информации все увеличивается. Поскольку документация играет важную роль в практическом программировании, мы исследуем каждую из этих категорий в следующих разделах.

Таблица 14.1. Источники документации в языке Python

Форма	Назначение
Комментарии #	Документация внутри файла
Функция <code>dir</code>	Получение списка атрибутов объектов
Строки документирования: <code>__doc__</code>	Документация внутри файла, присоединяемая к объектам
PyDoc: функция <code>help</code>	Интерактивная справка по объектам
PyDoc: отчеты в формате HTML	Документация к модулям для просмотра в браузере
Стандартный набор руководств	Официальное описание языка и библиотеки
Веб-ресурсы	Интерактивные учебные руководства, примеры и т. д.
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 программистов. 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 описания 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 (т. е. методов и элементов данных). Она может быть вызвана для любого объекта, который имеет атрибуты. Например, чтобы узнать, что имеется в стандартном библиотечном модуле `sys`, просто импортируйте его и передайте имя модуля функции `dir`:

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__',
 '__stderr__', '__stdin__', '__stdout__', '_getframe', 'argv',
 'builtin_module_names', 'byteorder', 'copyright', 'displayhook',
 'dllhandle', 'exc_info', 'exc_type', 'excepthook',
 ...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 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__', ...остальные имена опущены...
 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith',
 'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit',
 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
 ...остальные имена опущены...]
```

Результаты работы функции `dir` для любого встроенного типа включают набор атрибутов, которые имеют отношение к реализации этого типа (методы перегруженных операторов); все они начинаются и заканчиваются двумя символами подчеркивания, чтобы сделать их отличными от обычных имен, и вы можете пока просто игнорировать их.

Того же эффекта можно добиться, передав функции `dir` имя типа вместо литерала:

```
>>> 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 преобразования, 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 говорить в шестой части книги, когда будем обсуждать классы.

Функция `dir` служит своего рода «кратким напоминанием» – она предоставляет список имен атрибутов, но ничего не сообщает о том, что эти имена означают. 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 к объектам и доступна во время выполнения. Синтаксически такие строки располагаются 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 файл *docstrings.py*. Строки документирования в нем располагаются в самом его начале, а также в начале функции и класса. Здесь для создания многострочных описаний файла и функции я использовал строки в тройных кавычках, но допускается использовать 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 обь-

ектами, достаточно просто импортировать файл и вывести значения их атрибутов `__doc__`, где интерпретатор 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
    can we have your liver then?

>>> print docstrings.employee.__doc__

class documentation
```

Обратите внимание, что для вывода строк документирования необходимо явно использовать инструкцию `print`, в противном случае будет выводиться единственная строка со встроенными символами новой строки.

Кроме того, существует возможность присоединять строки документирования к методам классов (эта возможность описывается ниже), но так как они представлены инструкциями `def`, вложенными в классы, это не является особым случаем. Чтобы извлечь строку с описанием метода класса, определяемого внутри модуля, необходимо указать имя модуля, класса и метода: `module.class.method.__doc__` (примеры строк документирования методов приводятся в главе 26).

Стандарты оформления строк документирования

Не существует какого-то общепринятого стандарта, который регламентировал бы, что должно входить в строки документирования (хотя в некоторых компаниях 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, положив 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 повезло. Однако я настоятельно рекомендую тщательно документировать свой программный 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 используется сходная методика присоединения документации — до и по

сле списка атрибутов, возвращаемых функцией `dir`. Например, чтобы увидеть удобочитаемое 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
...остальной текст опущен...

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 присоединено к их атрибутам `__doc__`:

```
>>> print sys.getrefcount.__doc__
getrefcount(object) -> integer

Return the current reference count for the object.
...остальной текст опущен...
```

Кроме того, можно прочесть 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.
...остальной текст опущен...

>>> print open.__doc__
file(name[, mode[, buffering]]) -> file object

Open a file. The mode can be 'r', 'w' or 'a' for reading
...остальной текст опущен...
```

Просматривая таким способом строки документирования встроенных инструментов, вы можете получить богатый объем информации, однако вам не требуется этого делать — эту информацию функция `help`, тема следующего раздела, предоставляет вам автоматически.

PyDoc: функция help

Методика использования строк документирования оказалась настолько удобной, что теперь в состав Python входит инструмент, который упрощает их отображение. Стандартный инструмент PyDoc написан на языке Python, он умеет извлекать строки документирования вместе с информацией о структуре программных компонентов и формировать из них удобно отформатированные отчеты различных типов.

Существуют различные способы запуска PyDoc, включая сценарий командной строки (за дополнительной информацией обращайтесь к ру-

ководству по библиотеке Python). Два, пожалуй, самых заметных интерфейса к PyDoc – это встроенная функция `help` и графический интерфейс к PyDoc для воспроизводства отчетов в формате HTML. Функция `help` вызывает PyDoc для создания простых 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 current reference count for the object.
    ...остальной текст опущен...
```

Обратите внимание: чтобы вызывать функцию `help`, не обязательно импортировать 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 информацию 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)

DESCRIPTION
    This module provides access to some objects used
    or maintained by the interpreter and to functions
    ...остальной текст опущен...

FUNCTIONS
    __displayhook__ = displayhook(...)
        displayhook(object) -> None

        Print an object to sys.stdout and also save it
        ...остальной текст опущен...

DATA
    __name__ = 'sys'
    __stderr__ = <open file '<stderr>', mode 'w' at 0x0082BEC0>
    ...остальной текст опущен...
```

Часть информации в этом отчете извлечена из строк документирования, а часть этих сведений (например, сигнатуры функций) – это информация о структуре программных компонентов, которую PyDoc извлекает автоматически, в результате анализа внутреннего устройства объектов. Кроме того, функция `help` может использоваться для полу-

чения 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 описаниями всех методов, доступных для этого типа:

```
>>> help(dict)
Help on class dict in module __builtin__:

class dict(object)
| dict( ) -> new empty dictionary.
| ...остальной текст опущен...

>>> help(str.replace)
Help on method_descriptor:

replace(...)
    S.replace (old, new[, maxsplit]) -> string

    Return a copy of string S with all occurrences
    ...остальной текст опущен...

>>> help(ord)
Help on built-in function ord:

ord(...)
    ord(c) -> integer

    Return the integer ordinal of a one-character string.
```

Наконец, функция `help` может извлекать информацию не только из встроенных, 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 автоматически, в результате исследования структуры объектов:

```
>>> help(docstrings.square)
Help on function square in module docstrings:

square(x)
    function documentation
    can we have your liver then?

>>> help(docstrings.employee)
...остальной текст опущен...

>>> help(docstrings)
Help on module docstrings:

NAME
    docstrings

FILE
    c:\python22\docstrings.py

DESCRIPTION
```

```
Module documentation
Words Go Here

CLASSES
    employee
    ...остальной текст опущен...

FUNCTIONS
    square(x)
        function documentation
    can we have your liver then?

DATA
    __file__ = 'C:\\PYTHON22\\docstrings.pyc'
    __name__ = 'docstrings'
    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 интерфейс (простой и переносимый сценарий Python/Tkinter), с помощью которого можно создавать отчеты в формате HTML, доступные для просмотра в любом веб-браузере. В этом случае PyDoc может выполняться как локально, так и удаленно, в режиме клиент/сервер. Внутри отчетов автоматически создаются гиперссылки, которые позволяют щелчком мыши перемещаться к описаниям взаимосвязанных компонентов в вашем приложении.

Чтобы запустить PyDoc в этом режиме, сначала необходимо запустить поисковый механизм, графический интерфейс которого представлен на рис. 14.1. Сделать это можно, выбрав пункт меню `Module Docs` в меню

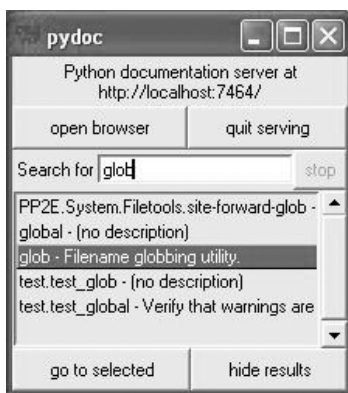


Рис. 14.1. Главное окно графического интерфейса PyDoc: введите имя требуемого модуля, нажмите клавишу `Enter`, выберите модуль и затем щелкните на кнопке «go to selected» (или, не вводя имя модуля, щелкните на кнопке «open browser», чтобы увидеть список всех доступных модулей)

Python кнопки Пуск в Windows или запустив сценарий *pydocgui.pyw* в каталоге *Tools*, где был установлен Python (также можно запустить сценарий *pydoc.py* с ключом *-g*). Введите имя интересующего вас модуля и нажмите клавишу Enter – PyDoc обойдет каталоги в пути поиска модулей (*sys.path*) и отыщет ссылки на указанный модуль.

Отыскав нужную запись, выберите ее и щелкните на кнопке «go to selected» (перейти к выбранному элементу). PyDoc откроет веб-браузер и отобразит отчет в формате HTML. На рис. 14.2 показано, как выглядит информация, представленная PyDoc, для встроенного модуля *glob*.

Обратите внимание на гиперссылки в разделе «Modules» (модули) на этой странице – вы можете щелкать на них мышью и перемещаться на страницы с описаниями этих (импортированных) модулей. Для больших страниц PyDoc также генерирует гиперссылки на различные разделы на этой странице.

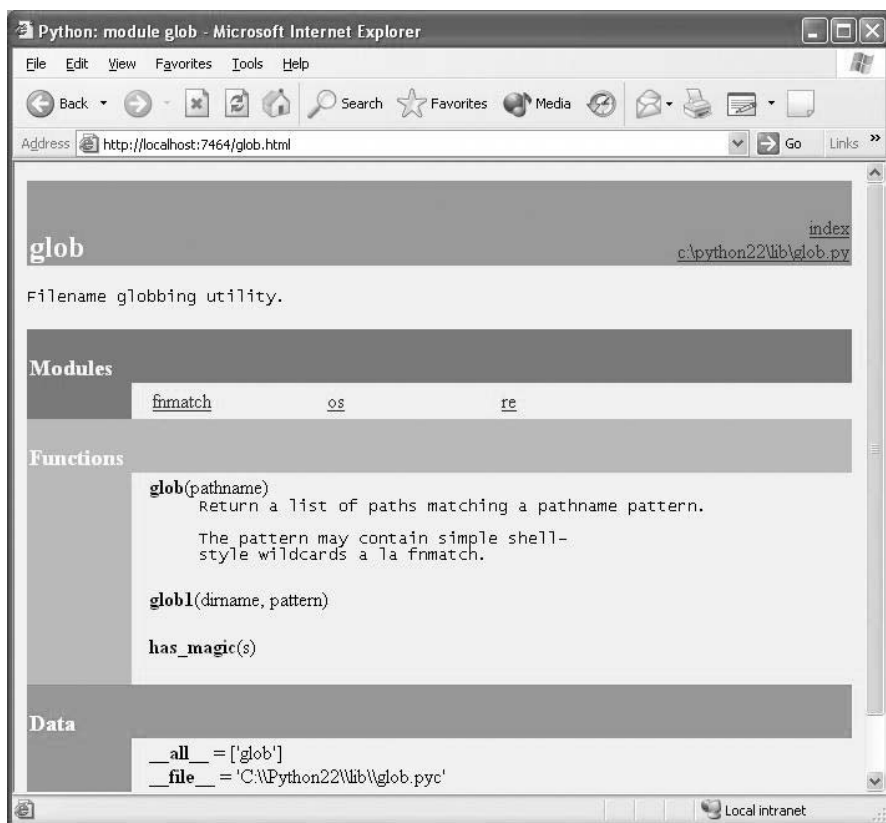


Рис. 14.2. Отыскав требуемый модуль в окне, представленном на рис. 14.1, щелкните на кнопке «go to selected», и описание модуля в формате HTML будет отображено в веб-браузере, как в данном случае, где отображено описание модуля из стандартной библиотеки

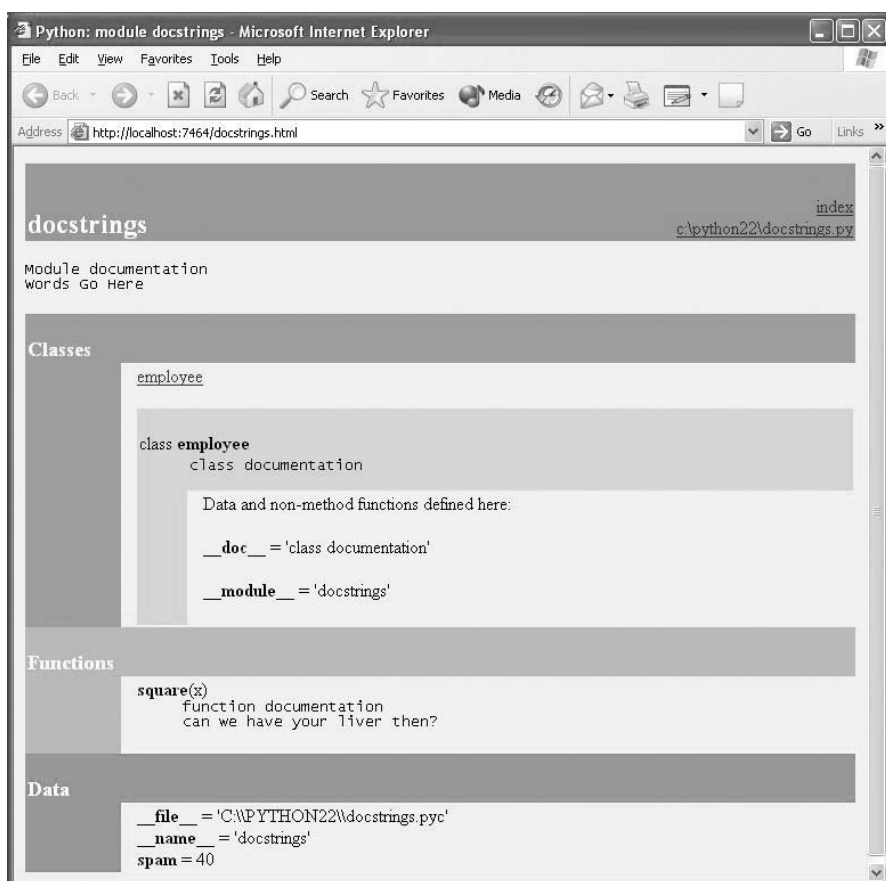


Рис. 14.3. PyDoc может служить источником документации как для встроенных, так и для пользовательских модулей. Здесь приводится страница с описанием пользовательского модуля, где можно видеть все строки документирования (docstrings), извлеченные из файла с исходными текстами

Подобно функции `help` графический интерфейс может извлекать информацию и из пользовательских модулей. На рис. 14.3 показана страница с информацией, извлеченной из нашего файла `docstrings.py`.

PyDoc можно настраивать и запускать разными способами, но мы не будем рассматривать эти возможности здесь – за дополнительной информацией обращайтесь 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 документирования в своих файлах, PyDoc сделает все необходимое, чтобы собрать и отформатировать их соответствующим образом. PyDoc – это всего лишь средство получения справки об объектах, таких как функции и модули, но он обеспечивает

простой доступ к документации с описанием этих компонентов. Его отчеты более полезны, чем просто списки атрибутов, хотя и менее исчерпывающи, чем стандартные руководства.



Совет дня: если поле ввода имени модуля в окне на рис. 14.1 оставить пустым и щелкнуть на кнопке «open browser» (открыть браузер), PyDoc воспроизведет веб-страницу с гиперссылками на все модули, доступные для импорта на данном компьютере. Сюда входят модули стандартной библиотеки, расширения сторонних производителей, пользовательские модули, расположенные в пути поиска импортируемых модулей, и даже модули, написанные на языке C, скомпонованные статически или динамически. Такую информацию сложно получить иными путями, если не писать свой программный код, который будет заниматься исследованием исходных текстов набора модулей.

Кроме того, PyDoc может сохранять документацию в формате HTML для последующего просмотра или вывода на печать; указания о том, как это сделать, вы найдете в документации. Следует отметить, что PyDoc может не совсем корректно работать со сценариями, которые читают данные из потока стандартного ввода, — PyDoc импортирует целевой модуль для последующего исследования, но при работе в режиме с графическим интерфейсом 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 описания языка Python и набора инструментальных средств. Руководства распространяются в формате HTML и в других форматах и в Windows устанавливаются 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 (следуйте по ссылке Documentation (Документация)). Руководства в системе Windows оформлены в виде файлов справки, поддерживающих возможность поиска; электронная версия на сайте *Python.org* также имеет страницу поиска.

После открытия руководства в операционной системе Windows оно отображает начальную страницу, как показано на рис. 14.4. Двумя самыми важными, пожалуй, здесь являются ссылки «Library Reference» (справочное руководство по библиотеке, где описываются встроенные типы, функции, исключения и модули 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 краткому введению для начинающих изучение языка.

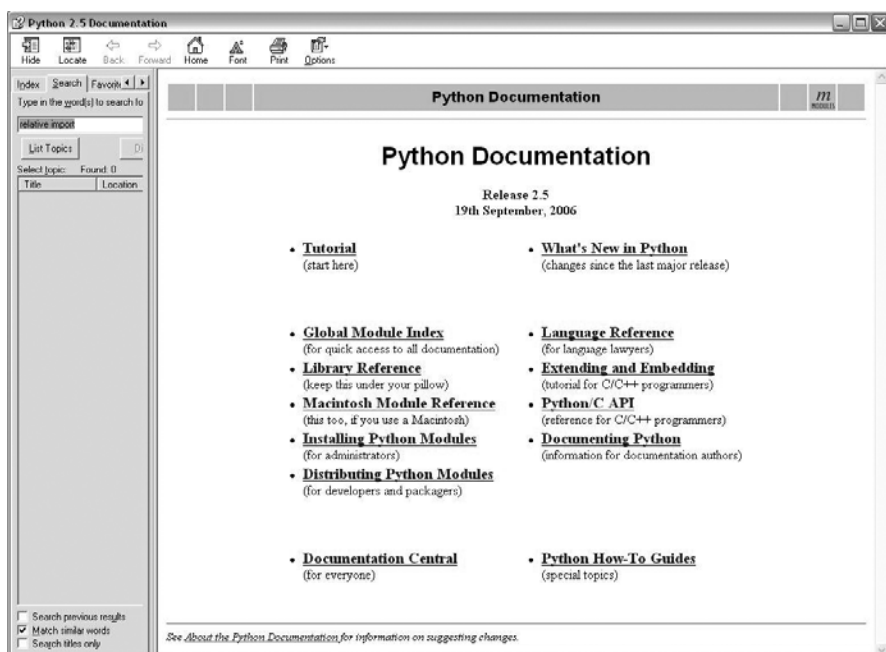


Рис. 14.4. Стандартный набор руководств по языку Python, доступный на сайте www.python.org в меню Help (Справка) среды разработки IDLE и в меню кнопки Пуск (Start) в операционной системе Windows. В Windows набор руководств оформлен в виде файла справки, поддерживающего возможность поиска; электронная версия на веб-сайте также имеет страницу поиска. Из всех предлагаемых руководств самым востребованным является Library reference (справочное руководство по библиотеке)

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 по ссылке «Documentation», можно получить доступ к электронному учебнику и к «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 ссылок на такие ресурсы, попробуйте поискать по строке «Python programming» в поисковой системе Google.

Печатные издания

Последний источник информации – это огромная коллекция печатных справочных 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, свойственных самому процессу издания. Обычно книга выходит в свет с отставанием на 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 программисты 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:

- **Не забывайте про двоеточия.** Никогда не забывайте вводить символ двоеточия в конце заголовков составных инструкций (первая строка таких инструкций, как `if`, `while`, `for` и других). Сначала вы наверняка будете забывать 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 относится 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 игнорируются, но когда программный код вводится в интерактивной оболочке, они завершают составные инструкции. Другими словами, ввод пустой строки сообщает интерактивной командной оболочке, что вы закончили ввод составной инструкции – если вам необходимо продолжить ввод такой инструкции, не нажимайте клавишу `Enter`, когда отображается строка приглашения к вводу `...`, пока вы действительно не закончите ее ввод.
- **Используйте отступы непротиворечивым способом.** Старайтесь не смешивать символы табуляции и пробелы при оформлении отступов в блоке, если вы не знаете точно, как текстовый редактор интерпретирует символы табуляции. В противном случае интерпретатор

Python будет видеть совсем не то, что вы видите на экране, когда он будет выполнять преобразование символов 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 текстовом 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 круглые скобки в инструкциях `if` и `while` (например, `if (X == 1):`). Это допустимо (любое выражение можно заключить в круглые скобки), но в данном контексте они совершенно излишни. Кроме того, не заканчивайте все инструкции 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 помните – не встраивайте инструкции присваивания в условные выражения циклов `while` и не заключайте блоки в фигурные скобки `{ }` (вложенные блоки 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 циклы `while` или основанные на использовании функции `range`. Так как в простых циклах `for` извлечение элементов последовательностей производится внутренними механизмами интерпретатора, они выполняются, порой, в два раза быстрее, чем эквивалентные циклы `while`. Избегайте искушения считать 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 использовании изменяемых объектов в инструкциях множественного присваивания (`a = b = []`), а также в расширенных инструкциях присваивания (`a += [1, 2]`). В обоих случаях непосредственные изменения могут затронуть другие переменные. Более подробно об этом рассказывается в главе 11.
- **Не ожидайте получения результатов от функций, выполняющих непосредственные 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 (отличных от `None`), поэтому их следует вызывать без присваивания возвращаемого значения. Начинающие программисты часто допускают ошибку, 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 – когда выполняется попытка обойти элементы словаря в порядке сортировки. Очень часто можно увидеть, например, такой программный код: `for k in D.keys().sort():`. Он почти работает – метод `keys` создает список ключей, а метод `sort` упорядочивает его, но, так как метод `sort` возвращает объект `None`, цикл `for` терпит неудачу, потому что, в конечном счете, выполняется попытка обойти элементы объекта `None` (который не является последовательностью). Этот алгоритм можно реализовать либо с помощью новой встроенной функции `sorted`, которая возвращает отсортированный список, либо необходимо разделить вызовы методов на инструкции: `Ks = 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 итераторов 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 независимо от наличия входных аргументов (например, вызов функции должен выглядеть как `function()`, а не `function`). В четвертой части книги вы узнаете, что функции – это простые объекты, которые могут выполнять специальную операцию – вызов – при обращении к имени с круглыми скобками.

Похоже, что эта проблема наиболее часто возникает в классах при работе с файлами – нередко можно увидеть, как начинающие программисты пытаются оформить вызов метода как `file.close`, а не как `file.close()`. Поскольку обращение к имени метода без круглых скобок в языке Python считается 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 сведения 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 другие расширения (например, `.рус`), указание расширения не только является нарушением синтаксиса, но и вообще не имеет смысла. Синтаксис определения пути зависит от типа платформы и определяется настройками параметра 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 познакомились 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 двинуться 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 использовать 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 функционального описания, где поясняются принципы использования модулей, функций, классов и методов. Комментарии, начинающиеся с символа решетки, лучше подходят для пояснений к выражениям и инструкциям. Такой порядок принят не только потому, что строки документирования проще отыскать в файле с исходными текстами, но и потому, что они могут извлекаться и просматриваться с помощью системы PyDoc.
2. Получить содержимое строк документирования можно с помощью атрибута `__doc__` объекта, передав его функции `help`, или выбирая модули в поисковой системе PyDoc с графическим интерфейсом, в режиме клиент/сервер. Дополнительно PyDoc обладает возможностью сохранять описание модулей 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 имени модуля и щелкните на кнопке «open browser» (открыть

броузер). В результате будет открыта веб-страница, содержащая ссылки на описания всех модулей, доступных вашей программе.

5. Мои, конечно. (А если серьезно, в предисловии имеется перечень некоторых книг, как справочных пособий, так и учебников, рекомендуемых для дальнейшего прочтения.)

Упражнения к третьей части

Теперь, когда вы узнали, как описывается логика работы программы, в следующих упражнениях вам будет предложено реализовать решение некоторых простых задач с использованием инструкций. Самым большим является упражнение 4, где вам будет предложено рассмотреть альтернативные варианты реализации. Одна и та же задача всегда может быть решена разными способами и отчасти изучение языка Python заключается в том, чтобы находить более оптимальные решения.

Решения приводятся в приложении В, в разделе «Часть III, Инструкции и синтаксис».

1. Основы циклов.

- a. Напишите цикл `for`, который выводит ASCII-коды всех символов в строке с именем `S`. Для преобразования символов в целочисленные ASCII-коды используйте встроенную функцию `ord(character)`. (Поэкспериментируйте с ней в интерактивной оболочке, чтобы понять, как она работает.)
- 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 новый список, содержащий ASCII-коды всех символов в строке. Дает ли выражение `map(ord, S)` похожий результат? (Подсказка: прочитайте четвертую часть.)

2. *Символы обратного следа.* 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 мало-понятные символы (символы, экранированные обратным слешем, приводятся в табл. 7.2).

3. *Сортировка словарей.* В главе 8 мы видели, что словари представляют собой 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 фрагмент, где для поиска числа 2 в пятой степени (32) в списке степеней числа 2, используются цикл `while` и флаг `found`. Этот фрагмент хранится в файле `power.py`.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

found = 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, принятые 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 – использование файла существенно упростит это упражнение):

- Сначала добавьте в цикл `while` блок `else`, чтобы избавиться от флага `found` и последней инструкции `if`.
- 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.
- Затем вообще избавьтесь от цикла, реализовав решение на основе оператора `in` проверки вхождения. (Подробности вы найдете в главе 8 или попробуйте ввести такое выражение: `2 in [1,2,3]`.)
- Наконец, вместо литерала списка `L` используйте цикл `for` и метод `append` для заполнения списка степеней двойки.

Более глубокие улучшения:

- Как вы думаете, повысится ли производительность, если выражение `2 ** X` вынести за пределы циклов? Как это можно сделать?
- 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 17.

IV

Функции

15

Основы функций

В третьей части книги мы рассмотрели основные процедурные инструкции языка Python. В этой части мы переходим к исследованию набора дополнительных инструкций, которые используются при создании функций.

Если говорить просто, то *функция* – это устройство, которое группирует набор инструкций так, что в программе они могут запускаться неоднократно. Функции могут вычислять некоторый результат и позволять указывать входные параметры, отличающиеся по своим значениям от вызова к вызову. Возможность оформления операций в виде функций – это очень удобный инструмент, который мы можем использовать в самых разных ситуациях.

С принципиальной точки зрения функции устраняют необходимость вставлять 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 программирование: 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, обеспечивающие *многократное использование* программного кода и уменьшающие его *избыточность*. Как будет показано далее, функции – это еще и средство проектирования, которое позволяет разбить сложную систему на достаточно простые и легко управляемые части. В табл. 15.1 приводятся основные инструменты, имеющие отношение к функциям, которые мы будем изучать в этой части книги.

Таблица 15.1. Инструкции и выражения, имеющие отношение к функциям

Инструкция	Примеры
Вызов	<code>myfunc("spam", "eggs", meat=ham)</code>
<code>def</code> , <code>return</code> , <code>yield</code>	<pre>def adder(a, b=1, *c): return a+b+c[0]</pre>
<code>global</code>	<pre>def changer(): global x; x = 'new'</pre>
<code>lambda</code>	<code>Funcs = [lambda x: x**2, lambda x: x*3]</code>

Зачем нужны функции?

Прежде чем перейти к обсуждению деталей, мы нарисуем себе четкую картину, что из себя представляют функции. Функции – это практически универсальное средство структурирования программы. Возможно, раньше вам уже приходилось сталкиваться с ними в других языках программирования, где они могли называться *подпрограммами* или *процедурами*. В процессе разработки функции играют две основные роли:

Максимизировать многократное использование программного кода и минимизировать его избыточность

Как и в большинстве других языков программирования, функции в языке Python представляют собой простейший способ упаковки логики выполнения, которая может использоваться в разных местах программы и более чем один раз. До сих пор весь программный код, который нам приходилось писать, выполнялся немедленно. Функции позволяют группировать и обобщать программный код, который может позднее использоваться произвольное 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 трудозатраты 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 т. д. 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 делать». 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 полиморфизма, введенному ранее в этой книге, поскольку на данном уровне его важность становится еще более очевидной. Как вы увидите, функции приносят 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 менее, мы уже использовали некоторые из них в предыдущих главах. Например, для создания объекта файла мы вызывали функцию `open`; точно так же мы использовали 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. Функции, 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 чтобы 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, каждую из которых мы будем изучать в этой части книги:

- **def – это исполняемый программный код.** Функции в языке Python создаются с помощью новой инструкции `def`. В отличие от функций в компилирующих 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 допустимо (а иногда даже полезно) вкладывать инструкции `def` внутрь инструкций `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 импортирования.
- **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 присваивания, имя становится ссылкой на объект-функцию. В имени функции нет ничего необычного – как будет показано далее, объект-функция может быть связан с несколькими именами, сохраняться в списке и т. д. Кроме того, функции могут создаваться с помощью выражения `lambda` (более сложная концепция, которая будет рассматриваться в одной из последующих глав).

- **return передает объект результата вызывающей программе.** Когда функция вызывается, вызывающая программа приостанавливает свою работу, пока функция не завершит работу и не вернет управление. Функции, вычисляющие какое-либо значение, возвращают его с помощью инструкции `return` – возвращаемое значение становится результатом обращения к функции. 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 рассматриваться позже.
- **Аргументы передаются посредством присваивания (в виде ссылок на объекты).** В языке Python аргументы передаются функциям посредством выполнения 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 аргументов 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 производится внутри функций, являются локальными для этих функций и существуют 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 производится в *области видимости* – там, где хранятся переменные, – а операция присваивания связывает имена с областями видимости.
- **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 объект любого типа и т. д. Как следствие этого одна и та же функция может применяться к объектам различных типов – допустимыми считаются любые объекты, поддерживающие 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 программного кода. А теперь начнем изучение некоторых из этих идей и рассмотрим несколько 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 отступами (или простая инструкция вслед за двоеточием). Блок инструкций образует *тело* функции, т. е. программный код, который выполняется интерпретатором всякий раз, когда производится вызов функции.

В строке заголовка инструкции `def` определяются *имя* функции, с которым будет связан объект функции, и список из нуля или более *аргументов* (иногда их называют параметрами) в круглых скобках. Имена аргументов в строке заголовка 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. Инструкция `return` содержит объектное выражение, которое дает результат функции. Инструкция `return` является необязательной – если она отсутствует, работа функции завершается, когда поток управления достигает конца тела функции. С технической точки зрения, функция без инструкции `return` автоматически 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 их рассмотрение мы отложим до главы 17, 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 инструкцией, `def` может появляться везде, где могут появляться инструкции, – даже внутри других инструкций. Например, даже при том, что инструкции `def` обычно исполняются, когда выполняется импорт вмещающего их модуля, допускается вкладывать

определения функций внутрь инструкций `if`, что позволяет 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 инструкция `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 интерпретируются, пока они не будут достигнуты и выполнены потоком выполнения, а программный код внутри инструкции `def` не выполняется, пока функция не будет вызвана позднее.

Так как определение функции происходит во время выполнения, имя функции не является однозначно определенным. 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 использованием нового имени. Как и все остальное в языке Python, функции — это обычные объекты; они явно записываются в память во время выполнения программы.

Первый пример: определения и вызовы

Кроме таких концепций времени выполнения (которые кажутся наиболее уникальными для программистов, имеющих опыт работы с традиционными компилирующими языками программирования) в использовании функций нет ничего сложного. Давайте напишем первый пример, в котором продемонстрируем основы. Как видите, функции имеют две стороны: *определение* (инструкция `def`, которая создает функцию) и *вызов* (выражение, которое предписывает интерпретатору выполнить тело функции).

Определение

Ниже приводится определение в интерактивной оболочке, которое определяет функцию с именем `times`. Эта функция возвращает результат обработки двух аргументов:


```
>>> def times(x, y):      # Создать функцию и связать ее с именем
...     return x * y      # Тело, выполняемое при вызове функции
... 
```

Когда интерпретатор достигает эту инструкцию `def` и выполняет ее, он создает новый объект функции, в который упаковывает программный код функции и связывает объект с именем `times`. Как правило, такие инструкции размещаются в файлах модулей и выполняются во время импортирования модулей, однако такую небольшую функцию можно определить и в интерактивной оболочке.

Вызов

После выполнения инструкции `def` вы сможете вызывать функцию в своей программе, добавляя 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 упоминалось 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 данном случае тело функции составляет единственная инструкция `return`, которая отправляет обратно результат выражения. Возвращаемый объект был выведен интерактивной оболочкой автоматически (как и в большинстве языков, $2 * 4$ в языке Python равно 8), однако, если бы результат потребовался бы нам 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 совершенно разных типов:

```
>>> times('Ni', 4)        # Функции не имеют типа
'NiNiNiNi'
```

На этот раз функция выполнила нечто совершенно иное. Вместо двух чисел в аргументах `x` и `y` функции были переданы строка и целое число. Вспомните, что оператор `*` может работать как с числами, так и с последовательностями; 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 определяются аргументами, которые ей передаются. Это основная

идея языка Python (и, возможно, ключ к использованию языка), которую мы рассмотрим в следующем разделе.

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 действительности оператор `*` — это всего лишь указание для обрабатываемых объектов.

Такого рода зависимость от типов известна как *полиморфизм* — термин, впервые встретившийся нам в главе 4, который означает, что смысл операции зависит от типов обрабатываемых объектов. Поскольку Python — это язык с динамической 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 автоматически применяться к целой категории типов объектов. Пока объекты поддерживают 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 это означает, что *любые* два объекта, поддерживающие оператор `*`, смогут обрабатываться функцией, и неважно, что они из себя представляют и когда были созданы. Эта функция будет работать с числами (выполняя операцию умножения), с двумя строками или со строкой и числом (выполняя операцию повторения) и с любыми другими комбинациями объектов, поддерживающими ожидаемый интерфейс — даже с объектами, порожденными на базе классов, которые мы еще пока не создали.

Кроме того, если функции будут переданы объекты, которые *не* поддерживают ожидаемый интерфейс, интерпретатор обнаружит ошибку при выполнении выражения `*` и автоматически возбудит исключение. Поэтому для нас совершенно бессмысленно предусматривать проверку на наличие ошибок в программном коде. Фактически, добавив такую проверку, мы ограничим область применения нашей функции, так как она сможет работать только с теми типами объектов, которые мы предусмотрели.

Это важнейшее отличие философии языка Python от языков программирования со статической типизацией, таких как C++ и Java: программный код на языке Python не делает предположений о конкретных типах данных. В противном случае он сможет работать только с теми типами данных, которые ожидалось на момент его написания, и он не будет поддерживать объекты других совместимых типов, кото-

рые 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 говоря, при программировании на языке Python во внимание принимают *интерфейсы* объектов, а не типы данных.

Конечно, такая модель полиморфизма предполагает необходимость тестирования программного кода на наличие ошибок, так как из-за отсутствия объявлений 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 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 простое 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 велика, как могла бы быть, потому что он может работать только с определенными переменными и не может быть использован повторно. Безусловно, можно было бы просто скопировать этот блок кода и вставлять его везде, где потребуется, но такое решение нельзя признать ни удачным, ни универсальным – нам по-прежнему придется редактировать каждую копию, изменяя имена последовательностей; изменение алгоритма также влечет за собой необходимость вносить 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 подход несет нам следующие преимущества:

- Оформив программный код в виде функции, мы получаем возможность использовать его столько раз, сколько потребуется.
- Так как вызывающая программа может передавать функции произвольные аргументы, функция сможет использоваться с любыми двумя последовательностями (или итерируемыми объектами) для получения их пересечения.
- Когда логика работы оформлена в виде функции, достаточно изменить программный код всего в одном месте, чтобы изменить способ получения пересечения.

- Поместив функцию в файл модуля, ее можно будет импортировать и использовать в любой программе на вашем компьютере.

В результате программный код, обернутый в функцию, превращается в универсальную 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 функция возвращает результат, мы также добавили инструкцию `return`, которая возвращает полученный объект результата вызывающей программе.

Вызов

Прежде чем функцию можно будет вызвать, ее необходимо создать. Для этого нужно выполнить инструкцию `def`, либо введя ее в интерактивной оболочке, либо поместив 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.

Еще о полиморфизме

Как и любая другая функция в языке Python, функция `intersect` также является полиморфной. То есть она может обрабатывать объекты произвольных типов при условии, что они поддерживают ожидаемый интерфейс:

```
>>> x = intersect([1, 2, 3], (1, 4))    # Смешивание типов
>>> x                                    # Сохраненный объект с результатом
[1]
```

На этот раз функции были переданы объекты разных типов – список и кортеж – и это не помешало ей отыскивать общие элементы. Так как отсутствует необходимость предварительного объявления типов аргументов, функция `intersect` благополучно будет выполнять итерации по объектам последовательностей любых типов при условии, что они будут поддерживать ожидаемые интерфейсы.

Для функции `intersect` это означает, что первый объект должен обладать поддержкой циклов `for`, а второй – поддержкой оператора `in`, выполняющего проверку 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 13, 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 интерфейсы (например, числа), интерпретатор автоматически обнаружит несоответствие и возбудит исключение, т. е. 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 повышаем его гибкость.

Локальные переменные

Переменная `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.

¹ Два важных замечания. Первое: с технической точки зрения файл может использоваться только в качестве первого аргумента функции `intersect`, потому что во время первого же применения оператора `in` файл будет просканирован до конца. Например, такой вызов, как `intersect(open('data1.txt'), ['line1\n', 'line2\n', 'line3\n'])`, будет работать, но вызов `intersect(open('data1.txt'), open('data2.txt'))` – нет, за исключением случая, когда первый файл содержит единственную строку. Для настоящих последовательностей итерационный контекст вызывает метод `iter` для получения итератора, который всегда переустанавливается в начало последовательности, но операции открытия и чтения файла приводят к исчерпанию итератора.

Второе: в случае с классами нам, возможно, пришлось бы использовать метод перегрузки операторов `__iter__` или более старый `__getitem__`, которые будут описаны в главе 24, чтобы обеспечить поддержку ожидаемого итерационного протокола. В этом случае мы можем определить, что означает понятие итерации для наших данных.

Почти все имена в функции `intersect` являются локальными переменными:

- Переменная `res` явно участвует в операции присваивания, поэтому она – локальная 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`, стоящая в конце функции `intersect`, возвращает *объект* результата, а имя `res` исчезает. Однако, чтобы полностью исследовать понятие локальности, нам необходимо перейти к главе 16.

В заключение

В этой главе были представлены основные идеи, на которых основано определение функции, – синтаксис и принцип действия инструкций `def` и `return`, выражений вызова функций, а также суть и преимущества полиморфизма в функциях языка Python. Как было показано, инструкция `def` – это исполняемый программный код, который создает объект функции. Когда позднее производится 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 реализованы 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 подробно эти темы будут рассматриваться в главе 16. 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 определения 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 означает, что в будущем нам потребуется изменить единственную копию действующего 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 разбить сложную систему на небольшие и легко управляемые части, каждая из которых может разрабатываться отдельно.
2. Функция создается, когда интерпретатор достигает инструкции `def` и выполняет ее; эта инструкция создает объект функции и связывает его с именем функции. Обычно это происходит, когда файл модуля, включающего функцию, импортируется другим модулем (вспомните, что во время импорта программный код файла выполняется от начала до конца, включая и инструкции `def`), но это может происходить, когда инструкция `def` вводится 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 инструкции `return`, по умолчанию функция возвращает объект `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 работать 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, которые используются функцией.)

16

Области видимости и аргументы

В главе 15 были описаны основы определения и вызова функций. Как мы видели, базовая модель функций в языке Python достаточно проста в использовании. В этой главе будут представлены подробности, лежащие в основе *областей видимости* – мест, где определяются переменные и где выполняется их поиск, – и механизма *передачи аргументов* – способа передачи объектов в функции.

Правила видимости

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, интерпретатор 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* подразумевается пространство имен: т. е. место в программном коде, где имени было присвоено значение, определяет область видимости этого имени для программного кода.

Практически все, что имеет отношение к именам, включая классификацию областей видимости, в языке Python связано с операциями присваивания. Как мы уже видели, имена появляются в тот момент, когда им впервые присваиваются 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 выполняется при-

сваивание, определяет пространство имен, в котором будет находиться имя, а, следовательно, и область его видимости.

Помимо упаковки программного кода функции приносят 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 пределами 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 значение за пределами данной инструкции `def` (например, в другой инструкции `def` или на верхнем уровне модуля) полностью отлочно от имени `X`, которому присвоено значение внутри инструкции `def`.

В любом случае область видимости переменной (где она может использоваться) всегда определяется местом, где ей было присвоено значение, и никакого отношения не имеет к месту, откуда была вызвана функция. Если присваивание переменной выполняется внутри инструкции `def`, переменная является *локальной* для этой функции; если присваивание производится за пределами инструкции `def`, она является *глобальной* для всего файла. Мы называем это *лексической областью видимости*, потому что области видимости 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` создает глобальную переменную с именем `X` (она видима из любого места в файле), а инструкция `X = 88` создает локальную переменную `X` (она видима только внутри 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 избежать конфликтов имен в программах и превращают функции в самостоятельные элементы программ.

Основы видимости имен в языке Python

До того как мы начали писать функции, весь программный код размещался на верхнем уровне модуля (например, он не был вложен в инструкции `def`), поэтому все имена, которые мы использовали, либо нахо-

дились на верхнем уровне в модуле, либо относились к предопределенным именам языка Python (например, `open`).¹ Функции образуют вложенные пространства имен (области видимости), которые ограничивают доступ к используемым в них именам, благодаря чему имена внутри функций не вступают в конфликт с именами за их пределами (внутри модуля или внутри других функций). Повторю еще раз, функции образуют *локальную область видимости*, а модули – глобальную. Эти две области взаимосвязаны между собой следующим образом:

- **Объемлющий модуль – это глобальная область видимости.** Каждый модуль – это глобальная область видимости, т. е. пространство имен, в котором создаются переменные на верхнем уровне в файле модуля. Глобальные переменные для внешнего мира становятся атрибутами объекта модуля, но внутри модуля могут использоваться как простые переменные.
- **Глобальная область видимости охватывает единственный файл.** Не надо заблуждаться насчет слова «глобальный» – имена на верхнем уровне файла являются глобальными только для программного кода в этом файле. На самом деле в языке Python не существует такого понятия, как всеобъемлющая глобальная для всех файлов область видимости. Имена всегда относятся к какому-нибудь модулю и всегда необходимо явно импортировать модуль, 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 – т. е. пространство имен, в котором находятся имена, определяемые внутри функции. Каждую инструкцию `def` (и выражение `lambda`) можно представить себе как определение новой локальной области видимости. 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 словами, каждый вызов создает новое локальное пространство имен. Рекурсию удобно использовать, когда выполняется обработка данных, структура которых заранее неизвестна.
- **Операция присваивания создает локальные имена, если они не были объявлены глобальными.** По умолчанию все имена, которым присваиваются значения внутри функции, помещаются в локальную область видимости (пространство имен, ассоциированное с вызовом функции). Если необходимо присвоить значение имени верх-

¹ Программный код, который вводится в интерактивной оболочке, в действительности находится на уровне модуля `__main__`, поэтому имена, создаваемые в интерактивной оболочке, также находятся внутри модуля и следуют обычным правилам видимости. Подробнее о модулях будет рассказываться в пятой части книги.

него уровня в модуле, который вмещает функцию, это имя необходимо объявить внутри функции глобальным с помощью инструкции `global`.

- **Все остальные имена являются локальными в объемлющей области видимости, глобальными или встроенными.** Предполагается, что имена, которым не присваивались значения внутри определения функции, находятся в объемлющей локальной области видимости (внутри объемлющей инструкции `def`), глобальной (в пространстве имен модуля) или встроенной (предопределенные имена в модуле `__builtin__`).

Обратите внимание, что любые операции присваивания, выполняемые внутри функции, классифицируют имена как локальные: инструкция `=`, инструкция `import`, инструкция `def`, передача аргументов и т. д. Следует также заметить, что операции непосредственного изменения объектов не рассматривают имена как локальные – это свойственно только операциям присваивания. Например, если имени `L` присвоен список, определенный на верхнем уровне в модуле, то такая инструкция, как `L.append(X)`, внутри функции не будет классифицировать имя `L` как локальное, тогда как инструкция `L = X` будет. В первом случае список `L` будет найден в глобальной области видимости и инструкция изменит глобальный список.

Разрешение имен: правило LEGB

Если предыдущий раздел показался вам запутанным, спешу успокоить – в действительности все сводится к трем простым правилам. Для инструкции `def`:

- Поиск имен ведется самое большее в четырех областях видимости: локальной, затем в объемлющей функции (если таковая имеется), затем в глобальной и, наконец, во встроенной.
- По умолчанию операция присваивания создает локальные имена.
- Глобальные объявления отображают имена на область видимости вмещающего модуля.

Другими словами, все имена, которым присваиваются значения внутри инструкции `def` (или внутри выражения `lambda`, с которым мы познакомимся позже), по умолчанию являются локальными; функции могут использовать имена в лексически (т. е., физически) объемлющих функциях и в глобальной области видимости, но чтобы иметь возможность изменять их, они должны быть объявлены 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, интерпретатор пытается отыскать его в четырех областях видимости – в локальной (*local*, *L*), затем в локальной области любой объемлющей инструкции `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.

- Когда внутри функции выполняется операция присваивания (а не обращение к имени внутри выражения), интерпретатор всегда создает или изменяет имя в локальной области видимости, если в этой функции оно не было объявлено глобальным.
- Когда выполняется присваивание имени за пределами функции (т. е. на уровне модуля или в интерактивной оболочке), локальная область видимости совпадает с глобальной – с пространством имен модуля.

На рис. 16.1 показаны четыре области видимости в языке Python. Примечательно, 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.¹

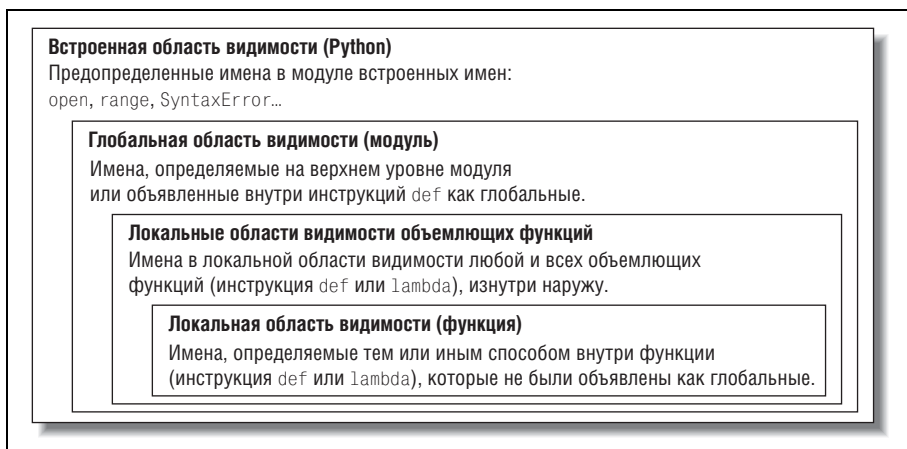


Рис. 16.1. Правило LEGB поиска имен в областях видимости.

Когда производится 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 видимости 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 область видимости

¹ В первом издании этой книги правило поиска в областях видимости было названо «правилом LGB». Уровень 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, поэтому мы рассмотрим ее позднее в этой главе.

Кроме того, имейте в виду, что эти правила применяются только к простым именам *переменных* (таким как `spam`). В пятой и шестой частях книги мы увидим, что полные имена *атрибутов* (такие как `object.spam`) принадлежат определенным 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 механизмом, который называется «наследованием» (рассматривается в шестой части книги).

Пример области видимости

Рассмотрим более крупный пример, демонстрирующий суть областей видимости. 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: result=100
```

В этом примере функция и модуль используют в своей работе несколько имен. Применяя правила области видимости языка Python, можно классифицировать эти имена следующим образом:

Глобальные имена: X и func

X — это глобальное имя, так как она объявлена на верхнем уровне модуля. К ней можно обращаться внутри функции, не объявляя ее глобальной. `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 выполнения функции), потому что присваивание значений обоим именам осуществляется внутри определения функции: присваивание переменной Z производится с помощью инструкции `=`, а Y — потому что аргументы всегда передаются через операцию присваивания.

Суть такого разделения имен заключается в том, что локальные переменные играют роль временных имен, которые необходимы только на время исполнения 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.

Встроенная область видимости

Встроенная область видимости, уже упоминавшаяся выше, немного проще, чем можно было бы подумать. В действительности, встроенная область видимости – это всего лишь встроенный модуль с именем `__builtin__`, но для того, чтобы использовать имя `builtin`, необходимо импортировать модуль `__builtin__`, потому что это имя само по себе не является встроенным.

Я вполне серьезен! Встроенная область видимости реализована как модуль стандартной библиотеки с именем `__builtin__`, но само имя не находится во встроенной области видимости, поэтому, чтобы исследовать его, необходимо импортировать модуль. После этого можно будет воспользоваться функцией `dir`, чтобы получить список предопределенных имен:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'DeprecationWarning', 'EOFError', 'Ellipsis',
 ...множество других имен опущено...
 'str', 'super', 'tuple', 'type', 'unichr', 'unicode',
 'vars', 'xrange', '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 поиск имен в этом модуле в последнюю очередь. Все имена из этого списка вы получаете в свое распоряжение по умолчанию, т. е., чтобы их использовать, не требуется импортировать какие-либо модули. Благодаря этому существует два способа вызвать встроенную функцию – используя правило LEGB или импортируя модуль `__builtin__` вручную:

```
>>> zip                                # Обычный способ
<встроенная функция zip>
>>> import __builtin__                # Более сложный способ
>>> __builtin__.zip
<встроенная функция zip>
```

Второй способ иногда удобно использовать при выполнении сложных действий. Внимательный читатель может заметить, что согласно правилу LEGB поиск имени прекращается, когда будет найдено первое

подходящее имя, откуда 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 встроенной области видимости. Например, внутри функции можно создать переменную 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 располагается во встроенной области видимости, окажется скрытой. Обычно это считается ошибкой, и самое неприятное, что интерпретатор Python не выведет сообщения с предупреждением (иногда в программировании возникают ситуации, когда действительно бывает необходимо подменить встроенные имена, переопределив их в своем коде).¹

Таким же способом функции могут переопределять имена глобальных переменных, 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 модуле, за пределами функции. Вследствие этого внутри функции нет никакой возможности изменить переменную, расположенную за пределами функции, если не добавить объявление `global` в инструкцию `def` (как описано в следующем разделе).

¹ Вот вам еще один пример того, что допустимо в языке Python, но чего не следует делать: имена `True` и `False` — это всего лишь переменные во встроенной области видимости и вполне возможно их переопределить с помощью такой инструкции: `True = False`. При этом вы не нарушите общую логическую целостность! Эта инструкция всего лишь переопределяет значение слова `True` в единственной области видимости. Что еще интереснее, можно выполнить даже такую инструкцию: `__builtin__.True = False`, и тогда истина станет ложью для всей программы! Такая возможность в будущем, скорее всего, будет ликвидирована (кстати, она отправляет среду IDLE в странное состояние, когда пользовательский процесс сбрасывается). 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, выводят предупреждения о типичных ошибках программирования, включая случайное переопределение встроенных имен (эта возможность, встроенная в PyChecker, известна как «shadowing» (сокрытие)).

Инструкция `global`

Инструкция `global` – это единственное, что в языке Python отдаленно напоминает инструкцию объявления. Однако это не объявление типа или размера – это объявление пространства имен. Она сообщает интерпретатору, что функция будет изменять одно или более глобальных имен, т. е. имен, которые находятся в области видимости (в пространстве имен) вмещающего модуля. Инструкция `global` уже упоминалась выше, а ниже приводится общая информация о ней:

- Глобальные имена – это имена, которые определены на верхнем уровне вмещающего модуля.
- Глобальные имена должны объявляться, только если им будут присваиваться значения внутри функций.
- Обращаться к глобальным именам внутри функций можно и без объявления их глобальными.

Инструкция `global` состоит из 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 выполнении 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 инструкции `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`:

```
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 глобальной потому, что она перечислена в инструкции `global`, которая явно отображает ее в 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, однако, следуя правилу LEGB, интерпретатор автоматически отыщет их в области 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, являются локальными, поэтому, если необходимо изменять имена за пределами функций, необходимо использовать инструкцию `global`. Это сделано в соответствии с общей идеологией языка Python – чтобы сделать что-то «неправильное», необходимо писать дополнительный программный код. Иногда бывает удобно использовать глобальные переменные, однако по умолчанию, если переменной присваивается значение внутри инструкции `def`, она становится локальной, потому что это, как правило, наилучшее решение. Изменение глобальных переменных может привести к появлению проблем, хорошо известных в разработке программного обеспечения: когда значения переменных зависят от порядка, 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 времени – значение переменной `x` зависит от выбранного момента времени, так как оно зависит от того, какая функция вызывалась последней (этого нельзя сказать только по одному файлу модуля).

В результате, чтобы понять этот программный код, необходимо знать путь потока выполнения всей программы. И если возникнет необходимость изменить этот модуль или использовать его в другой программе, необходимо будет удерживать в своей памяти всю программу. В этой ситуации невозможно использовать одну функцию, не принимая во внимание другую. От них зависит значение глобальной переменной. Это типичная проблема глобальных переменных – они вообще делают программный код более сложным для понимания и использования, в отличие от кода, состоящего только из независимых функций, логика выполнения которых построена на использовании локальных имен.

С другой стороны, за исключением случаев использования классов и принципов объектно-ориентированного программирования, глобальные переменные являются едва ли не самым удобным способом хранения информации о состоянии (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 видимости объемлющих функций, но они слишком сложны по сравнению с глобальными переменными.

Некоторые программы определяют отдельный глобальный модуль для хранения всех глобальных имен – если это предусмотрено заранее, это не так вредно. Кроме того, программы на языке Python, использующие многопоточную модель выполнения для параллельной обработки данных, тесно связаны с глобальными переменными – они играют роль памяти, совместно используемой функциями, исполняющимися в параллельных потоках, и выступают в качестве средств связи (описание многопоточной модели выполнения выходит далеко за рамки данной книги, поэтому за дополнительной информацией обращайтесь к книгам, упомянутым в предисловии).

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 поступали таким образом.

Минимизируйте количество изменений в соседних файлах

В этом разделе описывается еще одна проблема, связанная с областями видимости: несмотря на то, что *существует возможность* непосредственно изменять переменные в другом файле, этого следует избегать. Рассмотрим следующие два модуля:

```
# first.py
X = 99

# second.py
import first
first.X = 88
```

Первый модуль определяет переменную `X`, а второй – изменяет ее значение в инструкции присваивания. Обратите внимание, что для этого 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. В терминах этой главы глобальная область видимости модуля после импор-

тирования *превращается* в пространство имен атрибутов объекта модуля – импортирующий модуль автоматически получает доступ ко всем глобальным переменным импортируемого модуля, поэтому при импортировании глобальная область видимости модуля, по сути, трансформируется в пространство имен атрибутов.

После импортирования первого модуля второй модуль присваивает его переменной новое значение. Проблема состоит в том, что эта операция выполняется слишком неявно: для любого, кто занимается сопровождением или использует первый модуль, будет сложно догадаться, что какой-то другой модуль, далеко 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. Здесь также устанавливается слишком тесная зависимость между этими двумя модулями, потому что оба они зависят от значения переменной `X`, будет трудно понять или повторно использовать один модуль без другого.

И этом случае лучшая рекомендация – не использовать такую возможность; лучше организовать взаимодействие между модулями через вызовы функций, передавая им аргументы и получая возвращаемые значения. В данном конкретном случае было бы лучше добавить функцию доступа, которая будет выполнять изменения:

```
# first.py
X = 99
def setX(new):
    global X
    X = new

# second.py
import first
first.setX(88)
```

Для этого потребуется добавить дополнительный программный код, но он имеет огромное значение в смысле обеспечения удобочитаемости и удобства в сопровождении – 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. Мы не можем полностью избавиться от изменений в соседних файлах, однако здравый смысл диктует необходимость минимизировать их число, если это не является широко распространенным явлением в программе.

Другие способы доступа к глобальным переменным

Интересно, что благодаря трансформации глобальных переменных в атрибуты объекта загруженного модуля существует возможность имитировать инструкцию `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 загруз-

женных модулей `sys.modules` (подробнее об этой таблице рассказывает 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
    thismod.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 функция ничего не добавляет):

```
>>> import thismod
>>> thismod.test()
99
102
>>> thismod.var
102
```

Этот пример иллюстрирует эквивалентность глобальных имен и атрибутов модуля, однако, чтобы явно выразить свои намерения, нам потребовалось написать немного больше, чем при использовании инструкции `global`.

Области видимости и вложенные функции

До сих пор мы не еще рассматривали 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. Уровень *E* появился относительно недавно (он был добавлен в Python 2.2) – это локальные области видимости объемлющих инструкций `def`. Иногда объемлющие области видимости называ-

ют *статически вложенными областями видимости*. В действительности вложение является лексическим – вложенные области видимости соответствуют физически вложенным блокам программного кода в исходных текстах программы.



В Python 3.0 предполагается появление инструкции `nonlocal`, которая позволит получать доступ на запись к переменным в областях видимости объемлющих функций, так же, как нынешняя инструкция `global` позволяет получить доступ на запись к областям видимости объемлющего модуля. Синтаксически эта инструкция будет выглядеть так же, как инструкция `global`, только в ней будет использоваться слово `nonlocal`. Однако это в будущем, поэтому дополнительную информацию ищите в примечаниях к выпуску 3.0.

Вложенные области видимости

С появлением областей видимости вложенных функций правила поиска переменных стали немного более сложными. Внутри функции:

- Операция присваивания (`X = value`) создает или изменяет имя `X` в текущей локальной области видимости по умолчанию. Если имя `X` объявлено глобальным внутри функции, операция присваивания создает или изменяет имя `X` в области видимости объемлющего модуля.
- При обращении к переменной (`X`) поиск имени `X` сначала производится в локальной области видимости (функции); затем в локальных областях видимости всех лексически объемлющих функций, изнутри наружу; затем в текущей глобальной области видимости (в модуле); и, наконец, во встроенной области видимости (модуль `__builtin__`). Поиск имен, объявленных в инструкции `global`, начинается сразу с глобальной области видимости.

Обратите внимание, что инструкция `global` отображает имена в область видимости объемлющего модуля. Когда имеются вложенные функции, можно получить значения переменных в объемлющих функциях, но их нельзя изменить. Чтобы пояснить эти положения, рассмотрим их на примере программного кода.

Примеры вложенных областей видимости

Ниже приводится пример вложенной области видимости:

```
def f1():
    x = 88
    def f2():
        print x
        f2()

f1()                                # Выведет 88
```

Прежде всего – это вполне допустимый программный код на языке Python: инструкция `def` – это обычная исполняемая инструкция, которая

может появляться в любом месте программы, где могут появляться любые другие инструкции, включая вложение в другую инструкцию `def`. В этом примере вложенная инструкция `def` выполняется в момент вызова функции `f1` — она создает функцию и связывает ее с именем `f2`, которое является локальным и размещается в локальной области видимости функции `f1`. В некотором смысле `f2` — это временная 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 производится вывод переменной `x`, она ссылается на переменную `x` в локальной области видимости объемлющей функции `f1`. Функции имеют возможность обращаться 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 другую функцию:

```
def f1():
    x = 88
    def f2():
        print x
    return f2

action = f1() # Создает и возвращает функцию
action()      # Вызов этой функции: выведет 88
```

В этом фрагменте при вызове `action` фактически запускается функция, созданная во время исполнения функции `f1`. Функция `f2` помнит переменную `x` в области видимости объемлющей функции `f1`, которая уже неактивна.

Фабричные функции

В зависимости от того, кому задается вопрос о том, как называется такое поведение, можно услышать такие термины, как *замыкание* или *фабричная* функция, — объект функции, который сохраняет значения в объемлющих областях видимости, даже тогда, когда эти области могут прекратить свое существование. Классы (описываются в шестой части книги) обычно лучше подходят для сохранения состояния, потому что они позволяют делать это явно, посредством присваивания значений атрибутам, тем не менее, подобные функции обеспечивают другую альтернативу.

Например, фабричные функции иногда используются в программах, когда необходимо создавать обработчики событий прямо в процессе исполнения, в соответствии со сложившимися условиями (например,

когда желательно запретить пользователю вводить данные). Рассмотрим в качестве примера следующую функцию:

```
>>> def maker(N):
...     def action(X):
...         return X ** N
...     return action
... 
```

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 внутреннюю функцию, созданную при выполнении вложенной инструкции `def`. Если теперь вызвать то, что было получено 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`. Самое 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 управление. 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 сохраненная функция по-прежнему будет возвращать квадрат аргумента:

```
>>> g = maker(3)
>>> g(3)                        # 3 ** 3
27
>>> f(3)                        # 3 ** 2
9
```

Это довольно сложный прием, который вам вряд ли часто придется часто встречать на практике, впрочем, он распространен среди программистов, обладающих опытом работы с функциональными языками программирования (и иногда его можно встретить в выражениях `lambda`, как будет описано далее). Вообще классы, которые будут обсуждаться позднее, лучше подходят на роль «памяти», как в данном случае, пото-

му что они обеспечивают явное сохранение информации. Помимо классов основными средствами хранения информации о состоянии функций в языке Python являются глобальные переменные, объемлющие области видимости, как в данном случае, и аргументы по умолчанию.

Сохранение состояния объемлющей области видимости с помощью аргументов по умолчанию

В предыдущих версиях Python такой программный код, как в предыдущем разделе, терпел неудачу из-за отсутствия вложенных областей видимости в инструкциях `def` – при обращении к переменной внутри функции `f2` поиск производился сначала в локальной области видимости (`f2`), затем в глобальной (программный 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 значения аргументов по умолчанию для передачи (сохранения) объектов, расположенных в объемлющей области видимости:

```
def f1():
    x = 88
    def f2(x=x):
        print x
    f2()

f1()                                # Выведет 88
```

Этот фрагмент будет работать во всех версиях Python, и такой подход по-прежнему можно встретить в существующих программах на языке Python. Аргументы со значениями по умолчанию мы рассмотрим ниже, в этой же главе. А пока в двух словах замечу, что конструкция `arg = val` в заголовке инструкции `def` означает, что аргумент `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 будет 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` (т. е. объект 88).

Все это довольно сложно и полностью зависит от того, когда вычисляется значение по умолчанию. Фактически поиск во вложенных областях видимости был добавлен в Python, чтобы избавиться от такого способа использования значений по умолчанию – сейчас Python автоматически сохраняет любые значения в объемлющей области видимости для последующего использования во вложенных инструкциях `def`.

Безусловно, наилучшей рекомендацией будет просто избегать вложения инструкций `def` в другие инструкции `def`, так как это существенно

упростит программы. Ниже приводится фрагмент, который является эквивалентом предшествующего 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 исполнена до того, как первая функция попытается вызвать ее, – программный код внутри инструкции `def` не выполняется, пока не будет произведен фактический вызов функции:

```
>>> def f1():
...     x = 88
...     f2(x)
...
>>> def f2(x):
...     print x
...
>>> f1()
88
```

При использовании такого способа можно забыть о концепции вложенных областей видимости в языке Python, если вам не потребуется создавать фабричные функции, обсуждавшиеся выше, – по крайней мере при использовании инструкций `def`. Выражения `lambda`, которые практически всегда вкладываются в инструкции `def`, часто используют вложенные области видимости, как описывается в следующем разделе.

Вложенные области видимости и `lambda`-выражения

Несмотря на то, что вложенные области видимости на практике редко используются непосредственно для инструкций `def`, вам наверняка придется столкнуться с областями видимости вложенных функций, когда вы начнете использовать выражения `lambda`. Мы не будем подробно рассматривать эти выражения до главы 17, 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 использование 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 выражения. Таким образом, в настоящее время следующий программный код будет работать исключительно 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

    return action

x = func()
print x(2)                     # Выведет 16, 4 ** 2

```

До того как появилось понятие областей видимости вложенных функций, программисты использовали значения по умолчанию для передачи значений из объемлющей области видимости в выражения `lambda` 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

```

Поскольку `lambda` — это выражения, они естественно (и даже обычно) вкладываются внутрь инструкций `def`. Следовательно, именно они извлекли наибольшую выгоду от добавления областей видимости объемлющих функций в правила 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 области видимости, которая изменяется в цикле, все функции, созданные в этом цикле, будут иметь одно и то же значение — значение, которое имела переменная 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):
...         acts.append(lambda x: i ** x) # Сохранить каждое значение i
...     return acts
...
>>> acts = makeActions()
>>> acts[0]
<function <lambda> at 0x012B1680>

```

Такой подход не дает желаемого результата, потому что поиск переменной в объемлющей области видимости производится позднее, при вызове вложенных функций, в результате все они получают 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
>>> acts[4](2)      # Здесь должно быть 4 ** 2
16
```

Это один из случаев, когда необходимо явно сохранять значение из объемлющей 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 значение переменной из объемлющей области видимости в виде значения по умолчанию. 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 практике, особенно в программном коде, который генерирует функции-обработчики событий для элементов управления в графическом интерфейсе (например, обработчики нажатия кнопок). Подробнее о значениях по умолчанию и `lambda`-выражениях мы поговорим в следующей главе, поэтому позднее вам может потребоваться вернуться к этому разделу.¹

¹ В разделе «Ошибки при работе с функциями» к этой части книги, в конце следующей главы, мы также увидим, что существуют определенные проблемы с использованием изменяемых 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 единственных объектов, изменяемые объекты по умолчанию сохраняют свое состояние от вызова к вызову, а не инициализируются заново при каждом вызове. В зависимости от того, кому задается вопрос, эта особенность может рассматриваться или как особенность, поддерживающая возможность сохранения состояния, или как недостаток языка. Подробнее об этом будет говориться в следующей главе.

Произвольное вложение областей видимости

Прежде чем закончить это исследование, я должен заметить, что области видимости 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!
...             f3()
...         f2()
...     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 языке Python считается, что плоское лучше вложенного – ваша жизнь и жизнь ваших коллег будет проще, если вы сведете к минимуму количество вложенных определений функций.

Передача аргументов

Раньше уже говорилось, что передача аргументов производится посредством операции присваивания. Здесь имеется несколько моментов, не всегда очевидных 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 локальным именам.** Аргументы функции – ссылки на объекты, которые (возможно) используются совместно с вызывающей программой – это всего лишь результат еще одной из разновидностей операции присваивания. Ссылки в языке Python реализованы в виде указателей, поэтому все аргументы фактически передаются по указателям. Объекты, которые передаются в виде аргументов, никогда не копируются.
- **Операция присваивания именам аргументов внутри функции не оказывает влияния на вызывающую программу.** При вызове функции имена аргументов, указанные в ее заголовке, становятся новыми локальными именами в области видимости функции. Это не является совмещением имен между именами аргументов и именами в вызывающей программе.

- **Изменение внутри функции аргумента, который является изменяемым объектом, может оказывать влияние на вызывающую программу.** С другой стороны, так как аргументы – это всего лишь результат операции присваивания полученных объектов, функции могут воздействовать на полученные изменяемые объекты и тем самым оказывать влияние на вызывающую 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 – изменяемые 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 аргументов в языке Python буде казаться вам еще проще – согласно этой модели выполняется присваивание объектов именам аргументов, и она одинаково работает с любыми объектами, как с изменяемыми, так и с неизменяемыми.

Аргументы и разделяемые ссылки

Ниже приводится пример, который иллюстрирует работу этих положений:

```
>>> def changer(a, b):      # Функция
...     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                    # X - не изменилась, L - изменилась
(1, ['spam', 2])
```

В этом фрагменте функция `changer` присваивает значения аргументу `a` и компоненту объекта, на который ссылается аргумент `b`. Две операции присваивания в функции имеют незначительные синтаксические различия, но дают совершенно разные результаты:

- Так как `a` – это локальное имя в области видимости функции, первая операция присваивания не имеет эффекта для вызывающей программы – она всего лишь изменяет локальную переменную `a` и не изменяет связанное с ней имя `X` в вызывающей программе.
- `b` – также локальное имя, но в нем передается изменяемый объект (список, с именем `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 эта операция не изменяет объект `b`, она изменяет часть объекта, на который ссылается аргумент `b`, и это изменение оказывает влияние на вызывающую программу.

Рис. 16.2 иллюстрирует связи имя/объект, которые имеют место непосредственно 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,

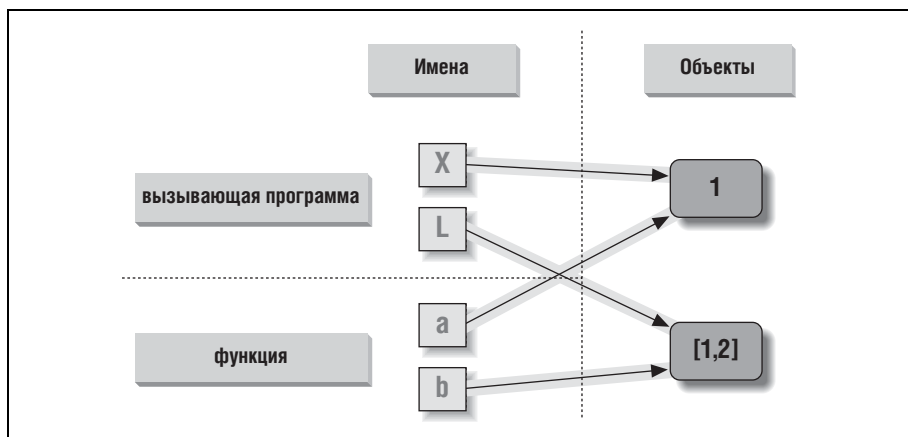


Рис. 16.2. Ссылки: аргументы. Так как аргументы передаются посредством операции присваивания, имена аргументов могут ссылаться на объекты, на которые ссылаются переменные, участвующие в вызове функции. Следовательно, непосредственные воздействия на изменяемые аргументы внутри функции могут оказывать влияние на вызывающую программу. В данном случае `a` и `b` внутри функции изначально ссылаются на те же объекты, на которые ссылаются переменные `X` и `L` при первом вызове функции. Изменение списка, выполненное через переменную `b`, можно наблюдать в переменной `L` после того, как функция вернет управление

как последовательность простых инструкций присваивания. Для первого аргумента операции 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
>>> 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'  # Непосредственное изменение: 'L' также изменяется
>>> print L
['spam', 2]
```

Вспомните обсуждение вопросов совместного использования изменяемых объектов 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 роль средства вывода информации.

Как избежать воздействий на изменяемые аргументы

В языке Python по умолчанию аргументы передаются в функции по ссылкам (т.е. 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 множества копий и легко изменять эти объекты в процессе работы. Однако, если нам требуется избежать влияния непосредственных изменений объектов, производимых в функциях, на вызывающую программу, мы можем 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 нам не требуется изменять полученный объект независимо от того, как была вызвана функция:

```
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. Чтобы действительно предотвратить изменения, мы всегда можем преобразовать изменяемые объекты в неизменяемые, что позволит выявить источники проблем. Например, при попытке изменения кортежа будет возбуждено исключение:

```
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 итерируемого объекта). Это особенно важно, потому что вынуждает писать функцию так, чтобы она никогда не пыталась изменять передаваемые ей аргументы. Однако такое решение накладывает на функцию больше ограничений, чем следует, 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 воздействия на передаваемые им изменяемые объекты (например, на списки и словари). Это не всегда является проблемой и часто может приносить пользу. Но вам действительно необходимо знать об этой особенности – если изменения в объектах происходят неожиданно для вас, проверьте вызываемые функции и в случае необходимости передавайте им копии объектов.

Имитация выходных параметров

Мы уже обсудили инструкцию `return` и использовали ее в нескольких примерах. Эта инструкция может возвращать объект любого типа, поэтому с ее помощью можно возвращать сразу несколько значений, упаковав их в кортеж или в коллекцию любого другого типа. В языке Python фактически отсутствует такое понятие, которое в некоторых других языках называется «передача аргументов по ссылке», однако мы можем имитировать такое поведение, возвращая 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 необязательные окружающие скобки просто опущены. После возврата из функции можно использовать операцию присваивания кортежа, чтобы извлечь отдельные элементы. (Если вы забыли, как это работает, вернитесь к разделу «Кортежи» в главе 4 и «Инструкции присваивания» в главе 11.) Такой прием позволяет имитировать выходные параметры, имеющиеся в других языках программирования, за счет использования явной операции присваивания. Переменные *X* и *L* изменятся после вызова функции, но только потому, что мы явно это запрограммировали.

Специальные режимы сопоставления аргументов

Как только что было показано, в языке Python аргументы всегда передаются через операцию *присваивания* – передаваемые объекты присваиваются именам, 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 заголовке функции. Эти возможности можно и не использовать, но они позволяют писать функции, поддерживающие более гибкие схемы вызова.

По умолчанию сопоставление аргументов производится в соответствии с их позициями, слева направо, и функции должно передаваться столько аргументов, 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.

Это достаточно сложный раздел, и прежде чем окунуться в обсуждение синтаксиса, 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 первую очередь для разработчиков библиотек, а не для разработчиков приложений. 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.

Сопоставление по ключам: соответствие определяется по указанным именам аргументов

Вызывающая программа имеет возможность указать соответствие между аргументами функции и их значениями в момент вызова, используя синтаксис `name=value`.

Значения по умолчанию: указываются значения аргументов, которые могут не передаваться

Функции могут определять значения аргументов по умолчанию на тот случай, если вызывающая программа передаст недостаточное количество значений. Здесь также используется синтаксис `name=value`.

Переменное число аргументов: прием произвольного числа аргументов, позиционных или в виде ключей

Функции могут использовать специальный аргумент, имени которого предшествует символ `*`, для объединения произвольного числа дополнительных аргументов в коллекцию (эта особенность часто называется *varargs*, как в языке C, где также поддерживаются списки аргументов переменной длины).

Переменное число аргументов: передача произвольного числа аргументов, позиционных или в виде ключей

Вызывающая программа также может использовать символ `*` для передачи коллекции аргументов в виде отдельных аргументов. 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 коллекции в виде произвольного числа отдельных аргументов.

В табл. 16.1 приводится синтаксис использования специальных режимов сопоставления.

Таблица 16.1. Виды сопоставления аргументов функций

Синтаксис	Местоположение	Интерпретация
<code>func(value)</code>	Вызывающая программа	Обычный аргумент: сопоставление производится по позиции
<code>func(name=value)</code>	Вызывающая программа	По ключу: сопоставление производится по указанному имени
<code>func(*name)</code>	Вызывающая программа	Все объекты в указанном имени передаются как отдельные позиционные аргументы
<code>func(**name)</code>	Вызывающая программа	Все пары ключ/значение в указанном имени передаются как отдельные аргументы по ключевым словам
<code>def func(name)</code>	Функция	Обычный аргумент: сопоставление производится по позиции или по имени

Синтаксис	Местоположение	Интерпретация
<code>def func(name=value)</code>	Функция	Значение аргумента по умолчанию, на случай, если аргумент не передается функции
<code>def func(*name)</code>	Функция	Определяет и объединяет все дополнительные аргументы в коллекцию (в кортеж)
<code>def func(**name)</code>	Функция	Определяет и объединяет все дополнительные аргументы по ключам (в словарь)

В вызывающей программе (первые четыре строки таблицы) при использовании простых значений соответствие именам аргументов определяется по позиции, но при использовании формы `name=value` соответствие определяется по именам аргументов – это называется *передачей аргументов по ключам*. Использование символов `*` и `**` в вызывающей программе позволяет передавать произвольное число объектов по позиции или по ключам в виде последовательностей и словарей соответственно.

В заголовке функции при использовании простых значений соответствие определяется по позиции или по имени (в зависимости от того, как вызывающая программа передает значения), но при использовании формы `name=value` определяются значения по умолчанию. При использовании формы `*name` все дополнительные аргументы объединяются в кортеж, а при использовании формы `**name` – в словарь.

Наиболее часто в программном коде на языке Python используются форма передачи аргументов по ключам и по умолчанию. Возможность передачи аргументов по ключам позволяет указывать значения аргументов вместе с их именами, чтобы придать вызову функции больше смысла. Со значениями по умолчанию мы уже встречались, когда рассматривали 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.

Специальные режимы сопоставления позволяют обеспечить свободу выбора числа аргументов, которые должны передаваться функции в обязательном порядке. Если функция определяет значения по умолчанию, они будут использоваться, когда функции передается недостаточное число аргументов. Если функция использует форму `*` определения списка аргументов переменной длины, она сможет принимать большее число аргументов – дополнительные аргументы будут собраны в структуру данных под именем с символом `*`.

Примеры использования ключей и значений по умолчанию

Предыдущие пояснения в программном коде выглядят гораздо проще. В языке Python по умолчанию сопоставление значений и имен аргументов производится по позиции, 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 т. д.:

```
>>> f(1, 2, 3)
1 2 3

```

Ключи

В языке Python существует возможность явно определить соответствия между значениями и именами аргументов при вызове функции. Ключи 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 сопоставляет имя *c* в вызове функции с именем аргумента *c* в заголовке определения функции и затем передает значение 3 в этот аргумент. Результат этого вызова будет тем же, что и предыдущего. Обратите внимание, что при использовании ключей порядок следования аргументов не имеет никакого значения, потому что сопоставление производится по именам, а не по позициям. Существует даже возможность объединять передачу аргументов по позициям и по ключам в одном вызове. В этом случае сначала будут сопоставлены все позиционные аргументы, слева направо, а потом будет выполнено сопоставление ключей с именами:

```
>>> f(1, c=3, b=2)
1 2 3

```

Большинство из тех, кто впервые сталкивается с такой возможностью, задаются вопросом: где такая возможность может пригодиться? В языке Python ключи обычно играют две роли. Первая: они делают вызовы 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, – ключи играют роль меток для данных, участвующих в вызове. Вторая: ключи используются вместе со значениями по умолчанию, о которых мы поговорим далее.

Значения по умолчанию

О значениях по умолчанию мы немного говорили ранее, когда обсуждали области 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 значения 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 ключу, а значения для аргументов `b` и `c` можно опустить. Если значения аргументов `b` и `c` будут опущены, они примут значения по умолчанию 2 и 3, соответственно:

```
>>> f(1)
1 2 3
>>> f(a=1)
1 2 3

```

Если функции передать только два значения, аргумент `c` примет значение по умолчанию, а если три – ни одно из значений по умолчанию не будет использовано:

```
>>> f(1, 4)
1 4 3
>>> f(1, 4, 5)
1 4 5

```

Наконец, ниже приводится пример взаимодействия режимов передачи значений по ключу и по умолчанию. Поскольку ключи могут нарушать обычный порядок отображения 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` получит 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 именем аргумента по ключу, а в заголовке

определяет значение по умолчанию для необязательного аргумента. В обоих случаях это не инструкция присваивания – это особая синтаксическая конструкция для этих двух случаев, которая модифицирует механику сопоставления значений с именами аргументов, используемую по умолчанию.

Примеры произвольного числа аргументов

Последние два расширения * и ** механизма сопоставления аргументов и их значений предназначены для поддержки возможности передачи произвольного числа аргументов функциям. Оба варианта могут появляться как в определениях 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 позиционных аргументов в кортеж:

```
>>> def f(*args): print args
... 
```

При вызове этой функции интерпретатор Python соберет все позиционные аргументы в новый кортеж и присвоит этот кортеж переменной args. Это будет обычный объект кортежа, поэтому из него можно извлекать элементы по индексам, выполнять обход в цикле for и т. д.:

```
>>> f()
()
>>> f(1)
(1,)
>>> f(1,2,3,4)
(1, 2, 3, 4)
```

Комбинация ** дает похожий результат, но применяется при передаче аргументов по ключам – в этом случае аргументы будут собраны в новый словарь, который можно обрабатывать обычными инструментами, предназначенными для работы со словарями. В определенном смысле форма ** позволяет преобразовать аргументы, передаваемые по ключам, в словари, которые можно будет обойти с помощью метода keys, итераторов словарей и т. д.:

```
>>> 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:

```
>>> 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 более сложные комбинации, которые на первый взгляд могут показаться неоднозначными – к этой идее мы еще вернемся ниже, в этой главе. А сейчас посмотрим, как используются формы * и ** в вызовах функций.

Извлечение аргументов из коллекции

В последних версиях Python форму * можно также использовать в вызовах функций. В этом случае данная форма передачи аргументов имеет противоположный 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 интерпретатору 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 также 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 позиционные аргументы и аргументы по ключу:

```
>>> 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
```

Такого рода программный код удобно использовать, когда заранее невозможно 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 заголовках функций и в их вызовах – в заголовке они означают сбор всех лишних аргументов в коллекцию, а в вызове выполняют распаковку коллекций в отдельные аргументы.

Мы еще вернемся к этим формам в следующей главе, когда будем рассматривать встроенную функцию `apply` (для замены которой, в значительной степени, предназначены эти конструкции).

Комбинирование ключей и значений по умолчанию

Ниже приводится немного больший по объему пример, демонстрирующий передачу аргументов по ключам и по умолчанию в действии. В этом примере вызывающая программа всегда должна передавать функции 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)
```

Обращаю снова ваше внимание: когда в вызовах используются ключи аргументов, порядок следования аргументов 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 выполняться как по позиции, так и по ключам. Обратите также внимание на то, что форма `name=value` имеет разный смысл в вызове функции и в инструкции `def` (ключ – в вызове и значение по умолчанию – в заголовке).

Функция поиска минимума

Чтобы придать обсуждению больше конкретики, выполним упражнение, которое демонстрирует практическое применение механизмов сопоставления аргументов. Предположим, что вам необходимо написать функцию, которая способна находить минимальное значение из произвольного множества аргументов с произвольными 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: числами, строками, списками, списками словарей, файлами и даже `None`.

Первое требование представляет собой обычный пример того, как можно найти применение форме `*` – мы можем собирать аргументы в кортеж и выполнять их обход с помощью простого цикла `for`. Второе требование тоже не представляет 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 интерпретатору 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 обходит остальную часть коллекции, отсекая первый элемент (нет никакого смысла сравнивать объект сам с собой, особенно, если это довольная крупная структура данных).
- Вторая версия позволяет интерпретатору самому выбрать первый аргумент и остаток, благодаря чему отсутствует необходимость извлекать первый аргумент и получать срез.
- Третья версия преобразует кортеж в список с помощью встроенной функции *list* и использует метод списка *sort*.

Метод *sort* написан на языке С, поэтому иногда он может обеспечивать более высокую производительность по сравнению с другими версиями функции, но линейный характер сканирования в первых двух версиях в большинстве случаев обеспечивает им более высокую скорость.¹ Файл *mins.py* содержит реализацию всех трех решений:

```
def min1(*args):
    res = args[0]
    for arg in args[1:]:
        if arg < res:
            res = arg
    return res

def min2(first, *rest):
    for arg in rest:
        if arg < first:
            first = arg
    return first
```

¹ В действительности все очень не просто. Функция *sort* в языке Python написана на языке С и реализует высокоэффективный алгоритм, который пытается использовать существующий порядок следования сортируемых элементов. Этот алгоритм называется «timsort» в честь его создателя Тима Петерса (Tim Peters), а в его документации говорится, что время от времени он показывает «сверхъестественную производительность» (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 результате сортировка выполняется быстрее, когда элементы последовательности частично упорядочены, и медленнее в других случаях. Даже при всем при этом производительность самого интерпретатора может изменяться по времени и тот факт, что сортировка реализована на языке С, может существенно помочь. Для более точного анализа производительности вы можете использовать модули *time* и *timeit*, с которыми мы познакомимся в следующей главе.

```
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 возбudit исключение, если та или иная функция не получит ни одного аргумента. В первом случае исключение 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, которое можно было бы вернуть в качестве признака ошибки. Из этого правила есть свои исключения (например, когда приходится выполнить дорогостоящие действия, прежде чем появится ошибка), но вообще лучше исходить из предположения, 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 заменить `<` 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 отыскивать 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 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 отыскивать максимальное (или любое другое) значение, мы могли бы просто передать ей нужную функцию *test*. На первый взгляд может показаться, что мы делаем лишнюю работу, однако главное преимущество такого обобщения функций (вместо того, чтобы содержать две версии, отличающиеся единственным символом) 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! Встроенные версии функций работают практически так же, как и наши, но они написаны на языке C для получения более высокой скорости работы.

Более полезный пример: универсальные функции

Теперь рассмотрим более полезный пример использования специальных режимов сопоставления аргументов. В конце предыдущей главы мы написали функцию, которая возвращала пересечение двух последовательностей (она отбирала элементы, 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 коллекции. Все аргументы передаются в тело функции в составе кортежа, поэтому для их обработки можно использовать простой цикл *for*. Ради интереса мы напишем функцию, возвращающую объединение,

которая также принимает произвольное число аргументов и собирает вместе все элементы, имеющиеся в любом из операндов:

```
def intersect(*args):
    res = []
    for x in args[0]:
        for other in args[1:]:
            if x not in other: break
        else:
            res.append(x)
    return res

def union(*args):
    res = []
    for seq in args:
        for x in seq:
            if not x in res:
                res.append(x)
    return res
```

Поскольку эти функции могут использоваться многократно (и они слишком большие, чтобы вводить их в интерактивной оболочке), мы сохраним их в модуле с именем *inter2.py* (подробнее о модулях рассказывается в пятой части книги). В обе функции аргументы передаются в виде кортежа *args*. Как и оригинальная версия *intersect*, обе они работают с любыми типами последовательностей. Ниже приводится пример обработки более чем двух последовательностей:

```
% python
>>> from inter2 import intersect, union
>>> s1, s2, s3 = "SPAM", "SCAM", "SLAM"

>>> intersect(s1, s2), union(s1, s2)
(['S', 'A', 'M'], ['S', 'P', 'A', 'M', 'C'])

>>> intersect([1,2,3], (1,4))
[1]

>>> intersect(s1, s2, s3)
['S', 'A', 'M']

>>> union(s1, s2, s3)
['S', 'P', 'A', 'M', 'C', 'L']
```



Следует заметить, что в языке Python появился новый тип данных — множества (описывается в главе 5), поэтому, строго говоря, ни одна из этих функций больше не требуется, — они включены в книгу только для демонстрации подходов к программированию функций. (Так как Python постоянно улучшается, наблюдается странная тенденция — мои книжные примеры делаются устаревшими с течением времени!)

Сопоставление аргументов: практические детали

Ниже приводятся несколько правил в языке Python, которым вам необходимо следовать, если у вас появится потребность использовать специальные режимы сопоставления аргументов:

- В вызове функции все аргументы, которые передаются не по ключу (`name`), должны находиться в начале списка, далее должны следовать аргументы, которые передаются по ключу (`name=value`), вслед за ними должна следовать форма `*name` и, наконец, `**name`.
- В заголовке функции аргументы должны стоять в том же самом порядке: обычные аргументы (`name`), вслед за ними аргументы со значениями по умолчанию (`name=value`), далее форма `*name`, если необходимо, и, наконец, форма `**name`.

Если поставить аргументы в любом другом порядке, вы получите сообщение о синтаксической ошибке, потому что в этих случаях комбинация получается неоднозначной. Сам интерпретатор использует следующий порядок сопоставления аргументов перед выполнением операций присваивания:

1. Определяются позиционные аргументы, которые передаются не по ключу.
2. Определяются аргументы, которые передаются по ключу, и производится сопоставление имен.
3. Определяются и помещаются в кортеж `*name` дополнительные аргументы, которые передаются не по ключу.
4. Определяются и помещаются в словарь `**name` дополнительные аргументы, которые передаются по ключу.
5. Определяются аргументы, имеющие значения по умолчанию, которые отсутствуют в вызове.

После этого интерпретатор убеждается, что в каждом аргументе передается единственное значение, в противном возбуждает исключение. Это действительно так же сложно, как выглядит, но изучение алгоритма сопоставления аргументов в языке Python поможет вам разобраться с некоторыми сложными случаями, особенно когда одновременно используются несколько режимов. Дополнительные примеры со специальными режимами сопоставления приводятся в упражнениях в конце четвертой части.

Как видите, режимы сопоставления аргументов могут быть весьма сложными. Но использовать их совершенно необязательно – вы можете ограничиться исключительно позиционными аргументами и это, пожалуй, будет наилучший выбор. Однако некоторые инструменты языка Python используют эти режимы, поэтому их понимание играет важную роль.

Придется держать в уме: аргументы, передаваемые по ключу

Аргументы, которые передаются по ключу, играют важную роль в библиотеке Tkinter, которая фактически стала стандартным средством для разработки графического интерфейса в языке Python. Мы познакомимся с Tkinter далее в этой книге, но в качестве предварительного знакомства замечу, что при использовании этой библиотеки для установки значений параметров компонентов графического интерфейса используются аргументы, передаваемые по ключу. Например, следующий вызов:

```
from Tkinter import *  
widget = Button(text="Press me", command=someFunction)
```

создает новую кнопку и определяет текст на кнопке и функцию обратного вызова, с помощью ключевых аргументов `text` и `command`. Так как графические компоненты могут иметь большое число параметров, аргументы, передаваемые по ключу, позволяют указывать только необходимые вам параметры. В противном случае пришлось бы перечислять все возможные параметры в соответствии с их позициями или надеяться, что аргументы со значениями по умолчанию будут правильно интерпретироваться во всех возможных ситуациях.

В заключение

В этой главе мы изучили две ключевые концепции, имеющие отношение к функциям: области видимости (как выполняется поиск переменных при обращениях к ним) и аргументы (как объекты передаются в функции). Здесь мы узнали, что переменные считаются локальными для определений функций, где выполняется присваивание значений этим переменным при условии, что они не объявлены как глобальные. Кроме того, мы узнали, что аргументы передаются функции через операцию присваивания, т. е. в виде ссылок на объекты, которые в действительности являются указателями.

Мы также познакомились с дополнительными особенностями областей видимости и аргументов, например с областями видимости вложенных функций и аргументами, которые передаются по ключу. Наконец мы познакомились с некоторыми рекомендациями по проектированию приложений (стремиться минимизировать количество глобальных переменных и избегать изменений переменных в соседних файлах) и 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 непо-

средственно на изменяемый объект может отразиться на вызывающей программе.

Следующая глава завершает тему функций исследованием более сложных концепций, связанных с функциями: `lambda`-выражений, генераторов, итераторов, функциональных инструментов, таких как `map`, и т. д. Многие из этих концепций исходят из того, что функции в языке Python являются обычными объектами и потому поддерживают дополнительные, очень гибкие режимы работы. Однако, прежде чем углубиться в эти темы, изучите контрольные вопросы к этой главе, чтобы закрепить знания, полученные здесь.

Закрепление пройденного

Контрольные вопросы

1. Что выведет следующий фрагмент и почему?

```
>>> X = 'Spam'
>>> def func():
...     print X
...
>>> func()
```

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. И в последний раз: что выведет следующий фрагмент и почему?

```
>>> def func(a, b, c=3, d=4): print a, b, c, d
...
>>> func(1, *(5,6))
```

7. Назовите три-четыре способа в языке Python сохранять информацию о состоянии в функциях.
8. Назовите три способа, которые могут использоваться для передачи результатов из функции в вызывающую программу.

Ответы

1. В данном случае будет выведена строка 'Spam', потому что функция обращается к глобальной переменной в объемлющем модуле (если внутри функции переменной не присваивается значение, она интерпретируется как глобальная).
2. В данном случае снова будет выведена строка 'Spam', потому что операция присваивания внутри функции создает локальную переменную и тем самым 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` предписывает выполнять присваивание внутри функции переменной, находящейся 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 вложенной функции отыщет имя в локальной области видимости объемлющей функции, а инструкция `print` в конце фрагмента отыщет имя в глобальной области видимости.
6. Здесь будет выведено "1 5 6 4": 1 соответствует аргументу в первой позиции, 5 и 6 соответствуют аргументам `b` и `c` в соответствии с фор-

мой `*name` (значение `b` переопределяет значение по умолчанию аргумента `c`) и `d` получит значение по умолчанию `4`, потому что четвертый аргумент в вызове функции отсутствует.

7. Так как значения локальных переменных исчезают, когда функция возвращает 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 значениями по умолчанию. Альтернативный способ заключается в использовании классов и приемов ООП, который обеспечивает лучшую поддержку возможности сохранения информации о состоянии, чем любой из трех предыдущих приемов, потому что этот способ делает сохранение явным, позволяя выполнять присваивание значений атрибутам.
8. Функции могут возвращать результаты с помощью инструкции `return`, воздействуя на изменяемые объекты, передаваемые в аргументах, а также изменением глобальных переменных. Вообще глобальные переменные использовать для этих целей не рекомендуется (за исключением редких случаев, таких как многопоточные программы), потому что это усложняет понимание и использование программного кода. Наилучшим способом является 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 устройства, как файлы и сокеты, но это уже выходит за рамки данной главы.

17

Расширенные возможности функций

В этой главе будут представлены дополнительные расширенные возможности, имеющие отношение к функциям: `lambda`-выражения, средства функционального программирования, такие как функция `map` и генераторы списков, функции-генераторы и выражения-генераторы и многие другие. Отчасти искусство использования функций лежит в области интерфейсов между ними, поэтому здесь мы также исследуем некоторые общие принципы проектирования функций. Так как это последняя глава в четвертой части, ее завершает раздел с описанием типичных ошибок и упражнения, которые помогут вам приступить к использованию отраженных здесь идей.

Анонимные функции: `lambda`

Вы уже знаете, как писать свои собственные обычные функции на языке Python. В следующих разделах будет описано несколько более сложных концепций, имеющих отношение к функциям. Большинство из них не являются обязательными, но при грамотном использовании они могут упростить задачи программирования.

Помимо инструкции `def` в языке Python имеется возможность создавать объекты функций в форме выражений. Из-за сходства с аналогичной возможностью в языке LISP она получила название `lambda`.¹ Подобно инструкции `def` это выражение создает функцию, которая будет вызываться позднее, но в отличие от инструкции `def`, выражение возвращает

¹ Название «`lambda`» отпугивает многих программистов, хотя в нем нет ничего страшного. Это название происходит из языка программирования LISP, в котором это название было заимствовано из лямбда-исчисления — разновидности символической логики. Однако в языке Python это просто ключевое слово, которое вводит выражение синтаксически.

функцию, а не связывает ее с именем. Именно поэтому `lambda`-выражения иногда называют *анонимными* (т. е. безымянными) функциями. На практике они часто используются как способ получить встроенную функцию или отложить выполнение фрагмента программного кода.

lambda-выражения

В общем виде `lambda` – это ключевое слово, за которым 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 инструкцией `def`, но здесь есть несколько различий, которые делают `lambda`-выражения удобными в некоторых специализированных случаях:

- **`lambda` – это выражение, а не инструкция.** По этой причине ключевое слово `lambda` может появляться там, где синтаксис языка Python не позволяет использовать инструкцию `def`, например внутри литералов или в вызовах функций. Кроме того, `lambda`-выражение возвращает значение (новую функцию), которое при желании можно присвоить переменной, в противовес инструкции `def`, которая всегда связывает функцию с именем в заголовке, а не возвращает ее в виде результата.
- **Тело `lambda` – это не блок инструкций, а выражение.** Тело `lambda`-выражения сродни тому, что вы помещаете в инструкцию `return` внутри определения `def`, – вы просто вводите результат в виде выражения вместо 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 возможность создания большого числа уровней вложенности программ; `lambda`-выражения предназначены для создания простых функций, а инструкции `def` – для решения более сложных задач.

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 автоматически.

В `lambda`-выражениях точно так же можно использовать аргументы со значениями по умолчанию:

```
>>> x = (lambda a="fee", b="fie", c="foe": a + b + c)
>>> x("wee")
'weefief'oe'
```

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 автоматически 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 имени `title` передавалось бы в виде значения по умолчанию — если вы забыли, почему, вернитесь к главе 16, где рассматривались области видимости.

Зачем использовать `lambda`-выражения?

Вообще говоря, `lambda`-выражения очень удобны для создания очень маленьких функций, к тому же они позволяют встраивать определения функций в программный код, который их использует. Они не являются предметом первой необходимости (вы всегда сможете вместо них использовать инструкции `def`), но они позволяют упростить сценарии, где требуется внедрять небольшие фрагменты программного кода.

Например, позднее мы увидим, что функции обратного вызова часто реализуются в виде `lambda`-выражений, встроенных непосредственно в список аргументов, вместо инструкций `def` где-то в другом месте в модуле и передаваемых по имени (примеры вы найдете во врезке «Придется держать в уме: функции обратного 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), (lambda x: x**3), (lambda x: x**4)]  
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 инструкции `def`, когда необходимо вставить маленькие фрагменты исполняемого программного кода туда, где использование инструкций недопустимо. Например, 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.

Подобные таблицы действий в языке Python можно создавать с помощью словаря и других структур данных:

```
>>> key = 'got'  
>>> {'already': (lambda: 2 + 2),  
...  'got':      (lambda: 2 * 4),  
...  'one':      (lambda: 2 ** 6)  
...  }[key]()  
8
```

В данном случае, когда интерпретатор Python создает словарь, каждое из вложенных lambda-выражений генерирует и оставляет после себя функцию для последующего использования – обращение по ключу извлекает одну из этих функций, а круглые скобки обеспечивают вызов извлеченной функции. При таком подходе словарь превращается в более универсальное средство множественного выбора, чем то, что я смог реализовать на основе инструкции `if` в главе 12.

Чтобы реализовать то же самое без использования lambda-выражений, пришлось бы написать три отдельные инструкции `def` за пределами словаря, в котором эти функции используются:

```
def f1(): return 2 + 2  
def f2(): return 2 * 4  
def f3(): return 2 ** 6  
...  
key = 'one'  
{ 'already': f1, 'got': f2, 'one': f3}[key]()
```

Этот прием тоже будет работать, но ведь инструкции `def` могут располагаться в файле модуля достаточно далеко, несмотря на то, что они очень короткие. *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 имеет 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.

`lambda`-выражения также очень удобно использовать в списках аргументов функций – для определения временных функций, которые больше нигде в программе 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 инструкций), казалось бы, устанавливает серьезное ограничение на объем логики, которую можно упаковать в `lambda`-выражение. Однако если вы понимаете, 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 действие, которое выполняет инструкция `print`). Точно так же в `lambda`-выражение можно заложить логику в виде трехместного выражения `if/else`, представленного в главе 13, или использовать эквивалентную, хотя и более сложную комбинацию операторов `and/or`, также описанную в главе 13. Как говорилось ранее, следующую инструкцию:

```
if a:
    b
else:
    c
```

можно имитировать одним из следующих примерно эквивалентных выражений:

```
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 главе 13 и вернемся к ним еще раз ниже, в этой главе):

```
>>> import sys
>>> showall = (lambda x: map(sys.stdout.write, x))
```

```
>>> 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 (запутанным). Вообще, простое лучше сложного, явное лучше неявного, а понятные инструкции лучше таинственных выражений. 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 видимости вложенной функции (символ `E` в названии правила **LEGB**, с которым мы познакомились в главе 16). Например, ниже lambda-выражение находится внутри инструкции `def` — типичный случай — и потому получает значение имени `x` из области видимости окружающей функции, имевшееся на момент ее вызова:

```
>>> def action(x):
...     return (lambda y: x + y)    # Создать и вернуть ф-цию, запомнить x
...
>>> act = action(99)
>>> act
<function <lambda> at 0x00A16A88>
>>> act(2)
101
```

В предыдущей главе, где обсуждались области видимости вложенных функций, не говорилось о том, что lambda-выражения обладают доступом к именам во всех объемлющих 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
```

Придется держать в уме: функции обратного вызова

Другое распространенное применение `lambda`-выражений состоит в определении функций обратного вызова для Tkinter GUI API. Например, следующий фрагмент создает кнопку, по нажатию которой на консоль выводится сообщение:

```
import sys
x = Button(
    text='Press me',
    command=(lambda:sys.stdout.write('Spam\n')))
```

Здесь в качестве обработчика события регистрируется функция, сгенерированная `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 обработчика до того момента, пока не произойдет событие: вызов метода `write` произойдет, когда кнопка будет нажата, а не когда она будет создана.

Поскольку правила областей видимости вложенных функций применяются и к `lambda`-выражениям, их проще использовать в качестве функций 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 экземпляра `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.display("spam")))
    def display(self, message):
        ...использовать текст сообщения...
```

В предыдущих версиях даже `self` приходилось передавать в виде аргумента со значением по умолчанию.

Эта структура `lambda`-выражений создает функцию, которая при вызове создает другую функцию. В обоих случаях вложенное `lambda`-выражение имеет доступ к переменной `x` в объемлющем `lambda`-выражении. Это фрагмент будет работать, но программный код выглядит весьма замысловато, поэтому в интересах соблюдения удобочитаемости лучше избегать использования вложенных друг в друга `lambda`-выражений.

Применение функций к аргументам

В некоторых программах бывает необходимо вызывать самые разные функции одинаковым образом, заранее не зная ни имен функций, ни их аргументов (примеры, где этот прием может быть полезен, будут показаны позднее). Сделать это можно с помощью встроенной функции `apply` и специального синтаксиса вызова в языке Python.



К моменту написания этих строк в версии Python 2.5 можно использовать и функцию `apply`, и специальный синтаксис вызова, описываемые в этом разделе, но вполне вероятно, что в Python 3.0 функция `apply` исчезнет. Если для вас желательно, чтобы ваш программный код сохранил свою работоспособность в будущем, используйте специальный синтаксис вызова, а не функцию `apply`.

Встроенная функция `apply`

Для обеспечения динамичности сгенерированные функции можно вызывать, передавая их в качестве аргументов функции `apply` вместе с кортежем аргументов, которые должны быть переданы этой функции:

```
>>> def func(x, y, z): return x + y + z
...
>>> apply(func, (2, 3, 4))
9
>>> f = lambda x, y, z: x + y + z
>>> apply(f, (2, 3, 4))
9
```

Функция `apply` просто вызывает функцию, переданную ей в первом аргументе, и сопоставляет полученный кортеж аргументов с аргументами, которые ожидает получить функция. Так как список аргументов передается в виде кортежа (т. е. в виде структуры данных), программа может создавать его во время выполнения.¹

Истинная мощь функции `apply` заключается в том, что ей не требуется знать, сколько аргументов принимает вызываемая ею функция. Например, можно с помощью условной инструкции `if` организовать выбор из множества функций и списков аргументов и затем передавать их функции `apply` для вызова:

```
if <test>:
    action, args = func1, (1,)
else:
    action, args = func2, (1, 2, 3)
```

¹ Будьте внимательны, не путайте функцию `apply` с функцией `map`, которая описывается в следующем разделе. Функция `apply` вызывает функцию, полученную в аргументе, всего один раз, в то время как `map` вызывает функцию несколько раз, по одному разу для каждого элемента последовательности.

```
...
apply(action, args)
```

В общем случае функцию `apply` удобно использовать, когда заранее нельзя определить список аргументов. Например, когда пользователь выбирает произвольную функцию пользовательского интерфейса, может оказаться невозможным жестко определить вызов функции в сценарии. Чтобы решить эту проблему, можно просто создать список аргументов в виде кортежа и вызвать функцию косвенно, с помощью `apply`:

```
>>> args = (2,3) + (4,)
>>> args
(2, 3, 4)
>>> apply(func, args)
9
```

Передача аргументов по ключу

Функция `apply` поддерживает третий необязательный аргумент, в котором можно указать словарь, содержащий аргументы, передаваемые функции по ключу:

```
>>> def echo(*args, **kwargs): print args, kwargs
...
>>> echo(1, 2, a=3, b=4)
(1, 2) {'a': 3, 'b': 4}
```

Эта возможность позволяет конструировать во время выполнения списки позиционных аргументов и аргументов передаваемых по ключу:

```
>>> pargs = (1, 2)
>>> kargs = {'a':3, 'b':4}
>>> apply(echo, pargs, kargs)
(1, 2) {'a': 3, 'b': 4}
```

apply-подобный синтаксис вызова

Тот же самый эффект, что дает функция `apply`, в языке Python можно получить с использованием специального синтаксиса вызова. Этот синтаксис отражает синтаксис передачи произвольного числа аргументов в заголовке инструкции `def`, с которым мы познакомились в главе 16. Например, предположим, что в этом примере используются имена с присвоенными им значениями из примера выше:

```
>>> apply(func, args)           # Традиционный способ: кортеж
9
>>> func(*args)                 # Новый, apply-подобный синтаксис
9
>>> echo(*pargs, **kargs)       # Словари с ключами также допустимы
(1, 2) {'a': 3, 'b': 4}
```

Этот специальный синтаксис вызова является более новым, чем функция `apply`, и, в общем, более предпочтительным. Он не дает каких-ли-

бо очевидных преимуществ перед явным вызовом `apply`, кроме соответствия заголовку инструкции `def` и необходимости ввода меньшего числа символов. Однако новый альтернативный синтаксис вызова позволяет передавать дополнительные аргументы и является более универсальным:

```
>>> echo(0, *pargs, **kargs)    # Обычный, *tuple, **dictionary
(0, 1, 2) {'a': 3, 'b': 4}
```

Отображение функций на последовательности: map

Одна из наиболее часто встречающихся задач, которые решаются в программах, 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 простого 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 предоставляет встроенную функцию, которая выполняет большую часть этой работы. Функция `map` применяет указанную функцию к каждому элементу последовательности 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 главе 13, как средство одновременного обхода сразу нескольких последовательностей в цикле. Как вы наверняка помните, в той главе вместо функции мы передавали объект `None`, чтобы выполнить попарное объединение элементов последовательностей. Ниже приводится более полезный пример, где используется настоящая функция, применяемая ко всем элементам списка, — функция `map` вызывает функцию `inc` для каждого элемента списка и собирает полученные результаты в новый список.

Функция `map` ожидает получить в первом аргументе функцию, поэтому здесь часто можно встретить `lambda`-выражения:

```
>>> 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 функции `map` представляет собой эквивалент цикла `for`, поэтому такую утилиту в общем виде можно представить так:

```
>>> def mymap(func, seq):
...     res = []
...     for x in seq: res.append(func(x))
...     return res
...
>>> map(inc, [1, 2, 3])
[11, 12, 13]
>>> mymap(inc, [1, 2, 3])
[11, 12, 13]
```

Однако функция `map` является встроенной функцией, поэтому она доступна всегда, всегда работает одним и тем же способом и обладает некоторыми преимуществами производительности (проще говоря – она выполняется быстрее, чем любой цикл `for`). Кроме того, функция `map` может использоваться в более сложных ситуациях, чем показано здесь. Например, в данном случае имеется несколько аргументов с последовательностями, а функция `map` извлекает их параллельно и передает как отдельные аргументы в функцию:

```
>>> pow(3, 4)
81
>>> map(pow, [1, 2, 3], [2, 3, 4]) # 1**2, 2**3, 3**4
[1, 8, 81]
```

Здесь функция `pow` при каждом вызове принимает от функции `map` два аргумента – по одному из каждой последовательности. Мы могли реализовать свою собственную функцию, имитирующую это действие, но вполне очевидно, что в этом нет никакой необходимости, так как имеется высокопроизводительная встроенная функция.



Этот вызов функции `map` напоминает генераторы списков, которые рассматривались в главе 13 и с которыми мы встретимся еще раз далее в этой главе. Основное отличие состоит в том, что `map` применяет к каждому элементу последовательности не произвольное выражение, а функцию. Вследствие этого ограничения она обладает меньшей гибкостью. Однако современная реализация `map` в некоторых случаях обладает более высокой производительностью, чем генераторы списков (например, когда отображается встроенная функция), и использовать ее проще. Благодаря этому функция `map` скорее всего останется доступной в версии Python 3.0. Однако в недавнем описании Python 3.0 предлагалось убрать из встроенного пространства имен (и, возможно, перенести в модули) эту функцию, а также функции `reduce` и `filter`, которые будут рассматриваться в следующем разделе.

В то время как функция `map` может остаться в языке Python, функции `reduce` и `filter` в версии 3.0 будут изъяты, отчасти потому, что они избыточны и легко могут быть реализованы с помощью генераторов списков (функция `filter` легко замещается оператором `if` в генераторах списков), а отчасти из-за своей сложности (функция `reduce` — одна из самых сложных функций в языке и одна из наименее понятных). Впрочем, я не могу предсказывать будущее, поэтому за дополнительной информацией об этих и других изменениях обращайтесь к примечаниям к выпуску Python 3.0. Эти функции по-прежнему рассматриваются в данном издании книги потому, что они входят в состав текущей версии Python, и почти наверняка будут встречаться в программном коде еще какое-то время.

Средства функционального программирования: filter и reduce

Функция `map` — это простейший представитель класса встроенных функций в языке Python, используемых в *функциональном программировании*, т. е. функций, которые применяют другие функции к последовательностям. Родственные ей функции отфильтровывают элементы с помощью функций, выполняющих проверку (`filter`), и применяют функции к парам элементов, накапливая результаты (`reduce`). Например, следующий вызов функции `filter` отбирает элементы последовательности больше нуля:

```
>>> range(-5, 5)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> filter((lambda x: x > 0), range(-5, 5))
[1, 2, 3, 4]
```

Элементы последовательности, для которых применяемая функция возвращает истину, добавляются в список результатов. Как и функция `map`, `filter` является 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]
```

`reduce` — более сложная функция. Ниже приводятся два вызова функции `reduce`, которые вычисляют сумму и произведение элементов списка:

```
>>> 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 первый элемент последовательности принимается в качестве начального значения. Ниже приводится цикл `for`, эквивалентный первому вызову, с жестко заданной операцией сложения внутри цикла:

```
>>> L = [1, 2, 3, 4]
>>> res = L[0]
>>> for x in L[1:]:
...     res = res + x
...
>>> res
10
```

Написать свою версию функции `reduce` (на тот случай, если она действительно будет убрана из Python 3.0) достаточно просто:

```
>>> 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
```

Если этот пример разжег ваш интерес, загляните также во встроенный модуль `operator`, который содержит функции, соответствующие встроенным выражениям, которые могут пригодиться при использовании некоторых функциональных инструментов:

```
>>> import operator
>>> reduce(operator.add, [2, 4, 6])    # Оператор сложения в виде функции
12
>>> reduce((lambda x, y: x + y), [2, 4, 6])
12
```

Как и функция `map`, `filter` и `reduce` поддерживают мощные приемы функционального программирования. Некоторые программисты могут дополнить комплект средств функционального программирования языка Python также `lambda`-выражениями в комплексе с функцией `apply` и генераторами списков, которые рассматриваются в следующем разделе.

Еще раз о генераторах списков: отображения

Отображение операций на последовательности и сбор результатов являются настолько распространенной задачей в программировании на языке Python, что в версии Python 2.0 появилась новая особенность – *генераторы списков*, которые упрощают решение задач еще больше, чем только что рассмотренные функции. Мы уже встречались с гене-

раторах списков в главе 13, но, так как они относятся к средствам функционального программирования, 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 функциональными альтернативами.

Основы генераторов списков

Рассмотрим несколько примеров, демонстрирующих самые основы. Как было показано в главе 7, встроенная функция `ord` в языке Python возвращает целочисленный код ASCII единственного символа (обратной к ней является встроенная 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 цикла `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 результатов мы можем достичь с помощью единственного вызова функции без необходимости заботиться о создании и заполнении списка:

```
>>> res = map(ord, 'spam')    # Применить функцию к последовательности
>>> res
[115, 112, 97, 109]
```

Но, начиная с версии Python 2.0, те же результаты можно получить с помощью генератора списка:

```
>>> res = [ord(x) for x in 'spam'] # Применит выражение к последовательности
>>> res
[115, 112, 97, 109]
```

Генераторы списков собирают результаты применения произвольного выражения 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 результаты 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]
```

Здесь создается список квадратов чисел от 0 до 9 (здесь мы позволили интерактивной оболочке автоматически вывести список – если вам необходимо сохранить список, присвойте его переменной). Чтобы выполнить аналогичные действия с помощью функции `map`, потребовалось бы написать отдельную функцию, реализующую операцию возведения в квадрат. Так как эта функция 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 ненамного сложнее (по крайней мере, для тех, кто разбирается в `lambda`-выражениях). Однако в случае более сложных выражений генераторы списков часто выглядят проще. В следующем разделе будет показано почему.

Добавление проверок и вложенных циклов

Генераторы списков обладают даже еще большей гибкостью, чем было показано до сих пор. Например, после цикла `for` можно добавить оператор `if` для реализации логики выбора. Генераторы списков с оператором `if` можно представить как аналог встроенной функции `filter`, представленной 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 сравнения здесь также показана реализация на основе цикла `for`:

```
>>> [x for x in range(5) if x % 2 == 0]
[0, 2, 4]

>>> 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 `filter` 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 единственном выражении:

```
>>> [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, присоединенное справа, возвращает ложь, а выражение слева вычисляет квадраты. Эквивалентный 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:

```
>>> 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 следующим 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 выполняться 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 иллюстрирует 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 каждой из последовательностей. Ниже приводится эквивалентная реализация на базе инструкций:

```
>>> 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. Программный 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, поэтому я даже не буду пытаться продемонстрировать их. Оставьте эту задачу в качестве упражнения мастерам Дзен, бывшим программистам на языке LISP и просто безумцам.

Генераторы списков и матрицы

Рассмотрим еще одно, более сложное применение генераторов списков, чтобы поупражнять мозги. Основной способ реализации матриц (они же – многомерные массивы) в языке Python заключается в использовании вложенных списков. В следующем примере определяют 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 индексирования для обращения к строкам и элементам внутри строк:

```
>>> M[1]
[4, 5, 6]

>>> M[1][2]
6
```

Генераторы списков являются мощным средством обработки таких структур данных, потому что они позволяют автоматически сканировать строки и столбцы матриц. Например, несмотря на то, что при такой структуре матрицы 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]

>>> [M[row][1] for row in (0, 1, 2)]
[2, 5, 8]
```

Пользуясь позициями, мы также легко можем извлечь элементы, лежащие на диагонали. В следующем примере используется функция `range` — она создает 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 т. д. (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 использовать для объединения нескольких матриц. Первый пример ниже создает простой список, содержащий результаты умножения соответствующих элементов 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 цикле: для каждой строки запускается итерация по столбцам, которая создает одну строку в матрице с результатами. Это выражение эквивалентно следующему 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, 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 непостижимыми, особенно при наличии вложенных конструкций. Поэтому начинающим осваивать язык Python я рекомендую в большинстве случаев использовать простые циклы `for` и функцию `map` (если они не становятся 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 более высокую его производительность: проведенные тесты свидетельствуют, 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 рекомендую использовать их для обеспечения большей простоты. Однако, функция `map` и генераторы списков стоят того, чтобы знать и применять их для реализации простых итераций, а также в случаях,

¹ Производительность этих тестов может зависеть от вида решаемой задачи, а также от изменений и оптимизаций в самом интерпретаторе языка Python. Например, в последних 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 встроженных функций). Чтобы проверить 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 альтернатив» далее в этой главе.

когда скорость работы приложения имеет критически важное значение. Кроме того, функция `map` и генераторы 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.

Еще раз об итераторах: генераторы

В этой части книги мы уже познакомились с обычными функциями, которые получают входные параметры и возвращают результат. Однако точно так же возможно написать функцию, которая может возвращать значение, а позднее продолжить свою работу с того места, где она была приостановлена. Такие функции известны как *генераторы*, потому что они генерируют последовательность значений с течением времени.

Функции-генераторы во многом похожи на обычные функции, единственное отличие состоит в том, что они автоматически поддерживают итерационный протокол и могут использоваться в контексте итераций. Мы рассмотрели итераторы в главе 13, а здесь мы взглянем на них еще раз, чтобы увидеть, какое отношение они имеют к генераторам.

Придется держать в уме: генераторы списков и `map`

Ниже приводится более реалистичный пример использования генераторов списков и функции `map` (мы решали эту задачу с помощью генераторов списков в главе 13, а здесь мы снова вернемся к ней, чтобы продемонстрировать 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 сразу во всех строках за одно действие с помощью генератора списков или функции `map`:

```
>>> [line.rstrip() for line in open('myfile').readlines()]
['aaa', 'bbb', 'ccc']

>>> [line.rstrip() for line in open('myfile')]
['aaa', 'bbb', 'ccc']

>>> 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 читать строки из файла). Вызов функции `map` выглядит немного длиннее, чем генератор списков, но ни в одном из этих двух случаев не требуется явно управлять списком 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 элементы кортежей – это значения столбцов:

```
listoftuple = [('bob', 35, 'mgr'), ('mel', 40, 'dev')]
```

Выбрать все значения из определенного столбца можно и вручную, с помощью цикла `for`, но функция `map` и генераторы списков сделают это быстрее и за один шаг:

```
>>> [age for (name, age, job) in listoftuple]
[35, 40]

>>> map((lambda (name, age, job): age), listoftuple)
[35, 40]
```

В обоих случаях используется операция присваивания кортежа, чтобы извлечь значения в список.

За дополнительной информацией о прикладных интерфейсах языка Python обращайтесь к другим книгам и источникам информации.

В отличие от обычных функций, которые возвращают значение и завершают работу, функции-генераторы автоматически приостанавливают и возобновляют свое выполнение, при этом сохраняя информацию, необходимую для генерации значений.

Вследствие этого часто они представляют собой удобную альтернативу вычислению всей серии значений заранее и ручному сохранению и восстановлению состояния в классах. Функции-генераторы при приостановке автоматически сохраняют информацию о своем состоянии, под которым понимается вся локальная область видимости, со всеми локальными переменными, которая становится доступной сразу же, как только функция возобновляет работу.

Главное отличие функций-генераторов от обычных функций состоит в том, что 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 бы-

ла приостановлена. Это позволяет функциям воспроизводить последовательности значений в течение долгого времени, вместо того чтобы создавать всю последовательность сразу и возвращать ее в виде некоторой конструкции, такой как список.

Функции-генераторы тесно связаны с понятием протокола итераций в языке Python. Проще говоря, функции, содержащие инструкцию `yield`, компилируются особым образом, как генераторы – при вызове они возвращают объект-генератор, поддерживающий интерфейс итераций. Функции-генераторы могут также содержать инструкцию `return`, которая завершает генерацию значений.

Объекты итераторов в свою очередь определяют метод `next`, который либо возвращает следующий элемент в итерации, либо возбуждает исключение (`StopIteration`) в конце итераций. Доступ к итератору можно получить с помощью встроенной функции `iter`. Циклы `for` в языке Python используют такой итерационный протокол, если он поддерживается, для обхода последовательностей (или генераторов последовательностей). Если протокол не поддерживается, инструкция `for` терпит неудачу и возвращается к операции индексирования последовательности.

Пример функции-генератора

Генераторы и итераторы – это достаточно сложные особенности языка, поэтому обязательно загляните в руководства по стандартной библиотеке языка Python, где найдете исчерпывающую информацию. Чтобы проиллюстрировать основные моменты, рассмотрим следующий фрагмент, где определяется функция-генератор, которая может использоваться для генерации серии квадратов чисел:¹

```
>>> def gensquares(N):
...     for i in range(N):
...         yield i ** 2          # Позднее продолжить работу с этого места
... 
```

Эта функция предоставляет значение и тем самым возвращает управление вызывающей программе на каждой итерации цикла – когда она возобновляет работу, восстанавливается ее предыдущее состояние и управление передается непосредственно в точку, находящуюся сразу же за инструкцией `yield`. Например, при использовании в заголовке цикла `for` управление возвращается функции на каждой итерации в точку, находящуюся сразу же за инструкцией `yield`:

```
>>> for i in gensquares(5):      # Возобновить работу функции
```

¹ Генераторы появились в языке Python, начиная с версии 2.2. В версии 2.2 для их использования необходимо было применять специальную инструкцию `import __future__` `import generators` (подробнее об этой форме инструкции рассказывается в главе 18). Генераторы стали доступны еще в версии 2.2 во многом благодаря тому, что лежащий в их основе протокол не требовал нового ключевого слова `yield`, нарушающего обратную совместимость.

```
...     print i, ': ',          # Вывести последнее полученное значение
...
0 : 1 : 4 : 9 : 16 :
>>>
```

Для завершения генерации 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 обратно был получен объект-генератор, который поддерживает протокол итераций (т. е. имеет метод `next`, который запускает функцию или возобновляет ее работу с места, откуда было поставлено последнее значение, а также возбуждает исключение `StopIteration` по достижении конца последовательности значений):

```
>>> x.next()
0
>>> x.next()
1
>>> x.next()
4
>>> x.next()
9
>>> x.next()

Traceback (most recent call last):
  File "<pyshell#453>", line 1, in <module>
    x.next()
StopIteration
```

Циклы `for` работают с генераторами точно так же – вызывают метод `next` в цикле, пока не будет перехвачено исключение. Если итерируемый объект не поддерживает этот протокол, вместо него цикл `for` использует протокол доступа к элементам по индексам.

Обратите внимание, что в этом примере мы могли бы просто сразу создать список всех значений:

```
>>> def buildsquares(n):
...     res = []
...     for i in range(n): res.append(i**2)
...     return res
...
>>> for x in buildsquares(5): print x, ': ',
...
0 : 1 : 4 : 9 : 16 :
```


В такой ситуации мы могли бы использовать любой из приемов: цикл `for`, функцию `map` или генератор списков:

```
>>> for x in [n**2 for n in range(5)]:
...     print x, ': ',
...
0 : 1 : 4 : 9 : 16 :
```

```
>>> for x in map((lambda x:x**2), range(5)):
...     print x, ': ',
...
0 : 1 : 4 : 9 : 16 :
```

Однако генераторы дают возможность избежать необходимости выполнять всю работу сразу, что особенно удобно, когда список результатов имеет значительный объем или когда вычисление каждого значения занимает продолжительное время. Генераторы распределяют время, необходимое на создание всей последовательности значений, по отдельным итерациям цикла. Кроме того, в более сложных случаях использования они обеспечивают простую альтернативу сохранению состояния вручную между вызовами в объектах классов (подробнее о классах рассказывается в шестой части книги) — в случае с генераторами переменные функций сохраняются и восстанавливаются автоматически.

Расширенный протокол функций-генераторов: `send` и `next`

В версии Python 2.5 в протокол функций-генераторов был добавлен метод `send`. Метод `send` не только перемещается к следующему элементу в последовательности результатов, как это делает метод `next`, но еще и обеспечивает для вызывающей программы способ взаимодействовать с генератором, влияя на его работу.

С технической точки зрения `yield` в настоящее время является не инструкцией, а выражением, которое возвращает элемент, передаваемый методу `send` (несмотря на то, что его можно использовать любым из двух способов, — как `yield X` или как `A = yield(X)`). Значения передаются генератору вызовом метода `send(value)`. После этого программный код генератора возобновляет работу, и выражение `yield` возвращает значение, полученное от метода `send`. Когда вызывается обычный метод `next()`, выражение `yield` возвращает `None`.

Метод `send` может использоваться, например, чтобы реализовать генератор, который можно будет завершать из вызывающей программы. Кроме того, генераторы в версии 2.5 поддерживают метод `throw(type)` для возбуждения исключения внутри генератора в последнем выражении `yield` и метод `close()`, который возбуждает 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.

Итераторы и встроенные типы

Как мы видели в главе 13, встроенные типы данных спроектированы так, чтобы воспроизводить объекты итераторов в ответ на вызов встроенной функции `iter`. Итераторы словарей, например, во время итераций воспроизводят список ключей:

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> x = iter(D)
>>> x.next()
'a'
>>> x.next()
'c'
```

Кроме того, все разновидности итераций (включая циклы `for`, функцию `map`, генераторы списков и других, с которыми мы встречались в главе 13) в свою очередь спроектированы так, чтобы для определения – поддерживается ли протокол автоматически вызывать встроенную функцию `iter`. Именно поэтому существует возможность выполнить обход ключей словаря, не прибегая к вызову метода `keys`, строк в файле – без вызова метода `readlines` или `xreadlines` и т. д.:

```
>>> 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,
...
Tis but
a flesh wound.
```

Кроме того, существует возможность реализовать произвольные объекты-генераторы с помощью классов, которые соответствуют протоколу итераторов и поэтому могут использоваться в циклах `for` и в других итерационных контекстах. Такие классы определяют специальный метод `__iter__`, возвращающий объект-итератор (что более предпочтительно, чем использование метода `__getitem__`, обеспечивающего доступ к элементам по индексу). Однако эта тема далеко выходит за рамки данной главы – обращайтесь к шестой части книги, где приводится информация о классах, и к главе 24 в частности, где приводятся примеры классов, реализующих протокол итераторов.

Выражения-генераторы: итераторы и генераторы списков

В 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
                               # итерируемый объект
<generator object at 0x011DC648>
```

Однако с функциональной точки зрения выражения-генераторы кардинально отличаются от генераторов списков – вместо того, чтобы создавать в памяти список с результатами, они возвращают объект-генератор, который в свою очередь поддерживает итерационный протокол, поставляя по одному элементу списка за раз в любом итерационном контексте:

```
>>> G = (x ** 2 for x in range(4))
>>> G.next()
0
>>> G.next()
1
>>> G.next()
4
>>> G.next()
9
>>> G.next()
Traceback (most recent call last):
  File "<pyshell#410>", line 1, in <module>
    G.next()
StopIteration
```

Как правило, нам не приходится наблюдать итерационную механику действий выражений-генераторов в виде вызовов метода `next`, как в данном примере, потому что циклы `for` вызывают его 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
```

Фактически именно таким образом работает любой итерационный контекст, включая встроенные функции `sum`, `map` и `sorted`, и другие итерационные инструменты, которые мы рассматривали в главе 13, 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 необходимы 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
>>> 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 памяти – они не требуют создания в памяти полного списка с результатами, как это делают генераторы списков в квадратных скобках. Кроме того, на практике они могут работать несколько медленнее, поэтому их лучше использовать, только когда объем результатов очень велик, – и мы естественным образом переходим к следующему разделу.

Хронометраж итерационных альтернатив

В этой книге нам встретилось несколько итерационных альтернатив. Чтобы подвести итог, коротко проанализируем ситуацию, соединив все, что мы узнали об итерациях и функциях.

Я уже упоминал, что генераторы списков обладают более высокой скоростью выполнения, чем циклы `for`, а скорость работы функции `map` может быть выше или ниже в зависимости от конкретной решаемой задачи. Выражения-генераторы, рассматривавшиеся в предыдущем разделе, обычно немного медленнее, чем генераторы списков, но при этом они минимизируют требования к объему используемой памяти.

Все это справедливо на сегодняшний день, но относительная производительность может измениться со временем (интерпретатор Python постоянно оптимизируется). Если вам захочется проверить это самим, попробуйте запустить следующий сценарий на своем компьютере, со своей версией интерпретатора:

```
# файл timerseqs.py

import time, sys
reps = 1000
size = 10000

def tester(func, *args):
    startTime = time.time()
    for i in range(reps):
```

```

    func(*args)
    elapsed = time.time() - startTime
    return elapsed

def forStatement():
    res = []
    for x in range(size):
        res.append(abs(x))

def listComprehension():
    res = [abs(x) for x in range(size)]

def mapFunction():
    res = map(abs, range(size))

def generatorExpression():
    res = list(abs(x) for x in range(size))

print sys.version
tests = (forStatement, listComprehension, mapFunction, generatorExpression)
for testfunc in tests:
    print testfunc.__name__.ljust(20), '=>', tester(testfunc)

```

Этот сценарий тестирует все альтернативные способы создания списков и, как видно из листинга, выполняет по 10 миллионов итераций каждым из способов, т. е. каждый из тестов создает список из 10 000 элементов 1000 раз.

Обратите внимание, как выражение-генератор вызывается через вызов встроенной функции `list`, чтобы вынудить его выдать все значения, — если бы этого не было сделано, мы бы просто создали генератор, который не выполняет никакой работы. Кроме того, заметьте, как программный код в самом конце сценария выполняет обход кортежа из четырех функций и выводит значение атрибута `__name__` для каждой из них: это встроенный атрибут, который возвращает имя функции.

Когда я запустил этот сценарий в среде IDLE в Windows XP, где установлен Python 2.5, я обнаружил следующее: генератор списков оказался почти в два раза быстрее эквивалентной инструкции цикла `for`, функция `map` оказалась немного быстрее генератора списков при отображении встроенной функции `abs` (возвращает абсолютное значение):

```

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

```

Но вот как изменилось положение дел, когда сценарий был изменен так, чтобы он выполнял настоящую операцию, такую как сложение:

```

...
...
def forStatement():
    res = []

```

```

    for x in range(size):
        res.append(x + 10)

def listComprehension():
    res = [x + 10 for x in range(size)]

def mapFunction():
    res = map((lambda x: x + 10), range(size))

def generatorExpression():
    res = list(x + 10 for x in range(size))

...

```

Присутствие вызова функции сделало вызов `map` таким же медленным, как и цикл `for`, несмотря на то, что инструкция цикла содержит больше программного кода:

```

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, анализ производительности, как в данном случае, становится очень непростым делом. В действительности невозможно заранее утверждать, какой метод лучше – лучшее, что можно сделать, это провести хронометраж своего программного кода, на своем компьютере, со своей версией Python. В этом случае все, что можно сказать наверняка, – это то, что в данной версии Python использование пользовательской функции в вызове `map` может привести к снижению производительности 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 главной целью при создании программ на языке Python – основное внимание должно уделяться удобочитаемости и простоте программного кода, и только потом код можно будет оптимизировать, если это действительно необходимо. Вполне возможно, что все четыре варианта обладают достаточной скоростью обработки имеющихся наборов данных – в этом случае основной целью должна быть ясность программного кода.

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 позволяет избежать проблем, связанных с используемой платформой (на некоторых платформах, к примеру, предпочтительнее использовать функцию `time.time`, а не `time.clock`). Кроме того, обратите внимание на модуль `profile` из стандартной библиотеки, где вы найдете полные исходные тексты инструментов профилирования программного кода.

Концепции проектирования функций

Когда начинают использоваться функции, возникает проблема выбора, как лучше связать элементы между собой, например, как разложить задачу на функции (связность), как должны взаимодействовать функции (взаимодействие) и т. д. Вы должны учитывать такие понятия, как слаженность, взаимодействие и размер функций, — часть которых относится к категории структурного анализа и проектирования. Некоторые понятия, имеющие отношение к взаимодействию функций и модулей, были представлены в предыдущей главе, а здесь мы коротко рассмотрим некоторые основные правила для тех, кто начинает осваивать язык Python:

- **Взаимодействие: для передачи значений функции используйте аргументы, для возврата результатов — инструкцию `return`.** Всегда следует стремиться сделать функцию максимально независимой от того, что происходит 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 (т. е. имена в объемлющем модуле) обычно далеко не самый лучший способ организации взаимодействий с функциями. Они могут порождать зависимости и проблемы согласованности, которые существенно осложняют отладку программ.
- **Взаимодействие: не воздействуйте на изменяемые аргументы, если вызывающая программа не предполагает этого.** Функции могут оказывать воздействие 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.** Хорошо спроектированная функция должна решать одну задачу, которую можно выразить в одном повествовательном предложении. Если это предложение допускает слишком широкое толкование (например: «эта функция 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 отдельные и более простые функции. В противном случае окажется невозможным повторно использовать программный код функции, в котором смешаны 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 вложенности часто свидетельствует о промахах в проектировании. Сохраняйте функции короткими и простыми.

- **Взаимодействие: избегайте непосредственного изменения переменных в другом модуле.** Мы рассматривали эту концепцию в предыдущей главе и еще вернемся к ней в следующей части книги, когда сконцентрируем свое внимание на модулях. И тем не менее напомним, что непосредственное изменение переменных в других модулях устанавливает тесную зависимость между модулями, так же как тесную зависимость устанавливает изменение глобальных переменных из функций – модули становятся сложными в понимании и малоприспособными для многократного использования. Всегда, когда это возможно, для изменения переменных модуля вместо прямых инструкций присваивания используйте функции доступа.

На рис. 17.1 приводится схема организации взаимодействий функций с внешним миром – входные данные поступают в функции из элементов слева, а результаты могут возвращаться в любой из форм справа. Многие программисты предпочитают использовать для ввода только аргументы, и для вывода – только инструкцию `return`.

Конечно, из приведенных выше правил проектирования есть свои исключения, включая те, что связаны с поддержкой ООП в языке Python. Как вы увидите в шестой части книги, классы в языке Python зависят от изменения передаваемого изменяемого объекта – функции воздействуют на атрибуты аргумента `self`, получаемого автоматически, из-

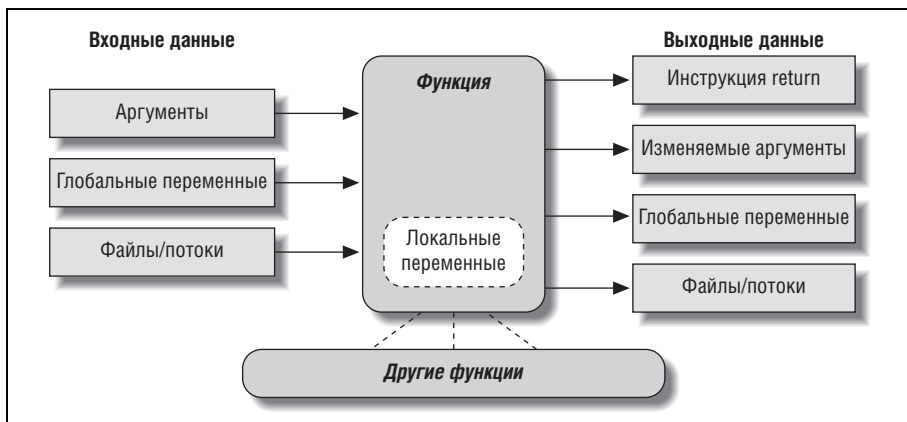


Рис. 17.1. Окружение функции времени выполнения. Функция может получать входные данные и возвращать результаты различными способами, однако функции проще в понимании и сопровождении, когда входные данные передаются в виде аргументов, а возврат результатов производится с помощью инструкции `return` или посредством воздействия на изменяемые аргументы, при условии, что последнее предполагается вызывающей программой

меняя информацию о его состоянии (например, `self.name = 'bob'`). Кроме того, когда классы не используются, глобальные переменные часто представляют для функций в модуле наилучший способ сохранения состояния между вызовами. Побочные эффекты в этом случае не опасны, потому что они ожидаемы.

Функции – это объекты: косвенный вызов

Так как функции в языке Python во время выполнения являются объектами, можно написать такую программу, которая будет обрабатывать их в общем виде. Объекты функций могут присваиваться, передаваться другим функциям, сохраняться в структурах данных и т. д., как если бы они были простыми числами или строками. Мы встречали уже такие способы использования в более ранних примерах. Кроме того, объекты функций поддерживают специальные операции: они могут вызываться перечислением аргументов в круглых скобках, следующих сразу же за выражением функции. И тем не менее, функции принадлежат к той же категории, что и другие объекты.

Например, в имени, которое используется в инструкции `def`, нет ничего уникального: это всего лишь переменная, которая создается в текущей области видимости, как если бы она 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
...
>>> x = echo                # Теперь на эту функцию ссылается еще и имя x
>>> x('Hello world!')      # Вызов объекта добавлением ()
Hello world!
```

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 добавлением списка аргументов в круглых скобках:

```
>>> def indirect(func, arg):
...     func(arg)           # Вызов объекта добавлением ()
...
>>> indirect(echo, 'Hello jello!') # Передача функции в функцию
Hello jello!
```

Существует даже возможность наполнять структуры данных функциями, как если бы они были простыми числами или строками. В этом нет ничего особенного, так как составные типы объектов могут содержать объекты любых типов:

```
>>> schedule = [ (echo, 'Spam!'), (echo, 'Ham!') ]
>>> for (func, arg) in schedule:
```

```
...     func(arg)
...
Spam!
Ham!
```

В этом фрагменте просто выполняется обход списка `schedule` и производится вызов функции `echo` с одним аргументом (обратите внимание на операцию присваивания кортежа в заголовке инструкции цикла `for`, которая была представлена в главе 13). Отсутствие описаний типов делает язык программирования Python невероятно гибким.

Типичные ошибки при работе с функциями

При работе с функциями вас поджидают подводные 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 операциями присваивания, производимыми во время выполнения. Эта особенность становится причиной появления самых причудливых сообщений в группе новостей, 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 присваивания 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):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in selector
UnboundLocalError: local variable 'X' referenced before assignment
(UnboundLocalError: обращение к локальной переменной 'X' до присваивания)
```

Было получено сообщение о том, что переменная не определена, но причина его появления неочевидна. Этот программный код компилируется интерпретатором 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` операция присваивания еще не производилась, интерпретатор сообщает о том, что имя не определено. Согласно этому правилу использования имен он говорит, что обращение к локальной переменной `X` произведено до того, как ей было присвоено значение. Фактически, любая операция присваивания внутри функции создает локальное имя. Операция импортирования, `=`, вложенные инструкции `def`, вложенные определения классов и т. д., – все трактуются именно таким образом.

Проблема возникает из-за того, что операция присваивания делает имена локальными для всей функции, а не только для той ее части, которая следует за инструкцией присваивания. На самом деле предыдущий пример далеко не однозначен: 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 переменной `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 использовать как локальную, так и глобальную версии одного и того же имени. Если вы действительно предполагаете вывести значение глобальной переменной, а затем присвоить значение локальной версии того же самого имени, импортируйте 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 (полному) имени (с `.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 непонятным, прочитайте пятую часть книги.¹

Значения по умолчанию и изменяемые объекты

Значения по умолчанию для аргументов функции вычисляются и запоминаются в момент выполнения инструкции `def`, а не при вызове функции. Внутренняя реализация Python сохраняет по одному объекту для каждого аргумента со значением по умолчанию, присоединенного к функции.

Вычисление значений по умолчанию в момент определения функции позволяет, в случае необходимости, сохранять значения из объемлющей области видимости. Но, так как значения по умолчанию сохраняются между вызовами функции, следует быть внимательным при воздействии на изменяемые значения по умолчанию. Например, следующая 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]

```

¹ Положение дел с локальными переменными в языке Python к настоящему моменту несколько улучшилось, потому что в данном случае выводится более определенное сообщение об ошибке «обращение к локальной переменной до присваивания», которое показано в листинге примера (теперь оно используется вместо более расплывчатого сообщения об ошибке, связанной с именем), впрочем, этот вид ошибки все еще встречается.

Некоторые воспринимают такое поведение как достоинство – изменяемые аргументы по умолчанию сохраняют свое состояние между вызовами функции, поэтому они могут играть роль, подобную роли *статических* локальных переменных в языке C. В некотором смысле они ведут себя как глобальные переменные за исключением того, что их имена являются локальными по отношению к функциям, вследствие чего исключается конфликт имен с переменными, определенными в другом месте.

Для большинства же это выглядит как недостаток, особенно для тех, кто впервые сталкивается с этой особенностью. В языке Python существует лучший способ сохранения состояния между вызовами функций (например, за счет использования 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 списка – 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, который выполняется при каждом вызове функции, вы всякий раз будете получать новый объект:

```
>>> def saver(x=None):
...     if x is None:      # Аргумент отсутствует?
...         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 оператор `or` в языке Python возвращает один из двух объектов: если аргумент отсутствует, 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 возвращает новый пустой список справа, – вер-

нитесь к разделу «Проверка истинности» в главе 12, если вам не понятно, почему так происходит). В разных программах могут предъявляться разные требования к такому поведению.

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 поток 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
...
>>> 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 «процедурами». Как правило, они вызываются как инструкции, а возвращаемое значение `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 возвращает. Например, присваивание результата метода списков `append` не вызывает появления 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 главе 14, действие таких функций проявляется как побочный эффект и они обычно вызываются как инструкции, а не как выражения.

Переменные цикла в объемлющей области видимости

Эта ошибка была описана в главе 16, когда мы рассматривали области видимости объемлющих функций, однако напомним еще раз: будьте внимательны при использовании переменных в области видимости объемлющей функции, которые 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, используйте аргу-

менты со значениями по умолчанию (дополнительные сведения по этой теме вы найдете в главе 16).

В заключение

В этой главе мы рассмотрели расширенные понятия, связанные с функциями – `lambda`-выражения; функции-генераторы и инструкцию `yield`; выражения-генераторы: `apply`-подобный синтаксис вызова; инструменты функционального программирования, такие как `map`, `filter` и `reduce`; и общие правила проектирования функций. Мы также повторно рассмотрели итераторы и генераторы списков, просто потому, что они так же связаны с функциональным программированием, как и инструкции циклов. В завершение изучения итерационных концепций мы произвели измерения производительности различных методов выполнения итераций. Наконец, мы коротко рассмотрели типичные ошибки, допускаемые при работе с функциями, чтобы помочь вам обойти потенциальные ловушки.

Эта глава завершает функциональную часть книги. В следующей части мы рассмотрим модули – вершину в иерархии структур языка Python; структуру, в которой всегда и располагаются наши функции. После этого мы займемся исследованием классов – инструментов, которые являются пакетами функций со специальным первым аргументом. Как вы увидите, все, что мы здесь узнали, пригодится везде, где далее в книге будут появляться функции.

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. Пусть имеются функция и кортеж аргументов, как можно было бы вызвать эту функцию?
6. Как связаны между собой функция `map` и генераторы списков? В чем их сходства и различия?
7. Как связаны между собой `lambda`-выражение и инструкция `def`? 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 похожее назначение, но не воспроизводят список результатов целиком. Вместо этого выражения-генераторы возвращают объект-генератор, который предоставляет по одному значению при использовании в итерационном контексте.
2. Генераторы – это объекты, поддерживающие итерационный протокол – они обладают методом `next`, который выполняет переход к следующему 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 круглые скобки, и объекты-генераторы с помощью классов, которые определяют специальный метод `__iter__` (обсуждается далее в этой книге).
3. Функции-генераторы имеют в своем теле инструкцию `yield`. Во всем остальном они ничем не отличаются от обычных функций.
4. При наличии этой инструкции интерпретатор Python компилирует функцию как генератор – при вызове она возвращает объект-генератор, который поддерживает итерационный протокол. Когда запускается инструкция `yield`, она возвращает результат вызывающей программе и приостанавливает работу функции, после этого в ответ на вызов метода `next` со стороны вызывающей программы функция возобновляет свою работу с позиции после последней выполненной инструкции `yield`. Функции-генераторы также могут содержать инструкцию `return`, которая завершает работу генератора.
5. Вызвать функцию можно с использованием `apply`-подобного синтаксиса: `function(*argstuple)`. Кроме того, можно использовать встроенную функцию `apply(function, args)`, но из будущих версий Python эта функция, скорее всего, будет исключена и потому такой способ нельзя считать универсальным.
6. Вызов функции `map` напоминает генератор списков тем, что обе конструкции создают новый список с результатами, применяя операцию к каждому элементу последовательности или другого итерируемого объекта, по одному за раз. Главное различие состоит в том, что `map` применяет к каждому элементу функцию, а генератор списков – произвольное выражение. Вследствие 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 списков поддерживают расширенный синтаксис. Например, вложенные циклы `for` и условные операторы `if`, что делает их похожими на встроенную функцию `filter`.

7. И `lambda`-выражения, и инструкция `def` создают объекты функций для последующего вызова. Однако `lambda`-выражения – это именно выражения, и поэтому они могут использоваться для вложения определений функций там, где инструкция `def` синтаксически недопустима. Нет таких случаев, когда нельзя было бы обойтись без `lambda`-выражений, – всегда можно определить идентичную инструкцию `def` и сослаться на функцию по имени. Однако `lambda`-выражения удобно использовать для встраивания небольших фрагментов кода, которые в программе больше нигде не используются. Синтаксически `lambda` позволяет возвращать значение единственного выражения; так как эта конструкция не поддерживает блоки инструкций, она плохо подходит для создания больших функций.

Упражнения к четвертой части

В этих упражнениях вам будет предложено написать более сложные программы. Обязательно проверьте решения в разделе «Часть IV. Функции» в приложении В; оформляйте свои решения в виде файлов модулей. Если будет допущена ошибка, будет очень сложно повторно ввести эти упражнения с клавиатуры в интерактивной оболочке.

1. *Основы.* В интерактивной оболочке интерпретатора Python напишите функцию, которая выводит на экран единственный аргумент, и попробуйте 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 конкатенацию). Затем добавьте в конец файла модуля вызовы функции `adder` с объектами различных типов (две строки, два списка, два числа с плавающей точкой) 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 аргументов, 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 16, где используется подобный прием.) Что произойдет, если функции передать аргументы разных типов? Что произойдет, если ей передать словари?
4. *Передача аргументов по ключу.* 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 интерактивной оболочке. Попробуйте передавать ей один, два, три и четыре аргумента. Попробуйте передавать аргументы по именам. Будет ли работать такой вызов: `adder(ugly=1, good=2)`? Почему? Наконец, обобщите новую версию функции `adder` так, чтобы принимала и вычисляла сумму/конкатенацию произвольного числа аргументов, передаваемых по ключу. Решение будет напоминать то, что было получено в упражнении 3, с той лишь разницей, что вам придется выполнить обход словаря, а не кортежа. (Подсказка: метод `dict.keys()` возвращает список, который можно обойти с помощью цикла `for` или `while`.)

5. Напишите функцию с именем `copyDict(dict)`, которая копирует словарь, получаемый в виде аргумента. Она должна возвращать новый словарь, содержащий 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 объединение двух словарей. Она должна возвращать новый словарь, содержащий все элементы обоих аргументов (которые, как предполагается, являются 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 определите 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 аргументов по ключу

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 со значениями
                                     # по умолчанию
```

Теперь протестируйте следующие вызовы в интерактивной оболочке и попробуйте объяснить полученные результаты – в некоторых слу-

чаях вам, возможно, придется вернуться к обсуждению алгоритмов сопоставления в главе 16. Как вы думаете, смешивание режимов сопоставления вообще можно считать удачным выбором? Можете ли вы придумать ситуации, когда 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)

>>> f6(1)
>>> f6(1, 3, 4)
```

8. *Снова простые числа.* Вспомните следующий фрагмент из главы 13, который 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 и добавьте несколько вызовов функции в конец этого файла. При этом замените оператор `/` на `//` в первой строке, чтобы функция могла также обрабатывать числа с плавающей точкой и была неязвима для истинной операции деления, которую, как планируется в Python 3.0, будет выполнять оператор `/` (описывается в главе 5). Что вы можете сказать об отрицательных значениях? Сумеете ли вы повысить скорость 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 создавать новый список, содержащий квадратные корни всех чисел из следующего 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 вычислений используйте функцию `sqrt` из модуля `math` (т. е. выполните `import math` и вызов `math.sqrt(x)`). Какой из трех вариантов на ваш взгляд является лучшим?



Модули

18

Модули: общая картина

Начиная с этой главы, мы приступаем к детальному изучению модуля в языке Python – самой крупной организационной программной единицы, которая 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 встроенной функцией:

`import`

Позволяет клиентам (импортерам) получать модуль целиком

`from`

Позволяет клиентам получать определенные имена из модуля

`reload`

Обеспечивает возможность повторной загрузки модуля без остановки интерпретатора Python

В главе 3 были представлены основные принципы, касающиеся модулей, и мы пользовались ими до сих пор. Часть V книги мы начнем с подробного описания базовых концепций, а затем перейдем к исследованию расширенных возможностей использования модулей. В этой первой главе вашему вниманию 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 т. д. Поскольку мо-

дули и классы – это всего лишь пространства имен, здесь мы также формализуем понятие пространства имен.

Зачем нужны модули?

В двух словах, модули обеспечивают простой способ организации компонентов в систему автономных пакетов переменных, известных как *пространства имен*. Все имена, определяемые на верхнем уровне модуля, становятся атрибутами объекта импортируемого модуля. Как мы видели в предыдущей части, операция импорта предоставляет доступ к именам в глобальной области видимости 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 интерактивной оболочке интерпретатора Python и исчезает безвозвратно после выхода из оболочки, программный код в файлах модулей хранится постоянно – его можно повторно загружать и запускать столько раз, сколько потребуется. Можно добавить, что модули – это место, где определяются имена, известные как *атрибуты*, на которые может 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. Модули 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 – выполняемый 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 требуется 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 импортироваться множеством клиентов.

Однако, чтобы понять действительную роль модулей в системе Python, нам необходимо 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 языке 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 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 исходными 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 файлов, содержащих *инструкции*. Программа организована как один *главный* файл, к которому могут подключаться дополнительные файлы, известные как *модули*.

Главный файл определяет, как будет двигаться основной поток выполнения программы, – это тот файл, который необходимо запустить, чтобы начать работу приложения. Файлы модулей – это библиотеки инструментальных 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 отдельно, – в них определяются инструментальные средства, используемые в других файлах. Чтобы получить доступ к определенным в модуле инструментам, именуемым *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. 18.1 схематически 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 инст-

рукций, который при запуске выполняется от начала и до конца. Файлы *b.py* и *c.py* – это модули, они также являются простыми текстовыми файлами, содержащими инструкции, но обычно они не запускаются как самостоятельные программы. Вместо этого они, как уже говорилось выше, обычно импортируются другими файлами, использующими инструментальные средства, определяемые в этих файлах.

Например, предположим, что файл *b.py* на рис. 18.1 определяет функцию с именем `spam`. Как мы уже знаем из четвертой части книги, чтобы определить функцию, 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')
```

В первой строке здесь располагается инструкция `import`, дающая файлу *a.py* доступ ко всему, что определено на верхнем уровне в файле *b.py*. Вкратце 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.

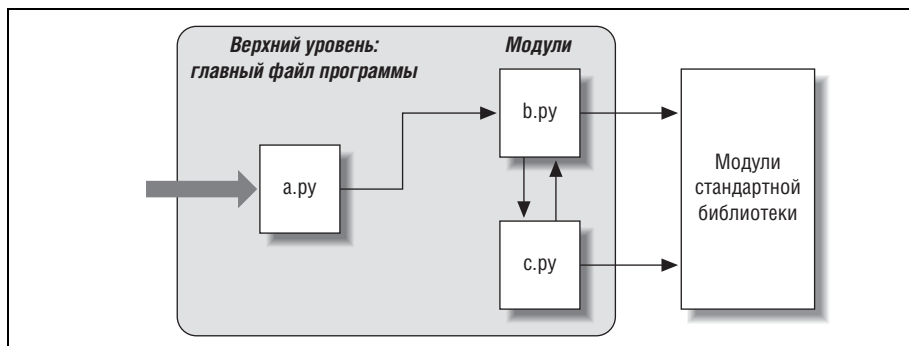


Рис. 18.1. Архитектура программы на языке Python. Программа – это система модулей. Она состоит из одного главного файла сценария (который требуется запустить, чтобы начать работу программы) и нескольких модулей (импортируемых библиотек инструментальных средств). И сценарии, и модули – это текстовые файлы, содержащие инструкции на языке Python, хотя инструкции в модулях обычно только создают объекты для последующего использования. Стандартная библиотека языка Python представляет собой коллекцию модулей, готовых к использованию

В языке Python невозможно обращаться к именам в других модулях, пока такие инструкции импорта не будут выполнены на этапе времени выполнения. Основная задача этих инструкций состоит в том, чтобы связать имена в модуле – 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: инструкция `import`, в действительности, последовательно выполняет инструкции в указанном файле, чтобы воссоздать его содержимое.

Вторая инструкция в файле *a.py* вызывает функцию `spam`, определенную в модуле `b`, используя форму записи атрибутов объекта. Запись `b.spam` означает следующее: «извлечь значение имени `spam`, расположенного в объекте `b`». В нашем примере – это вызываемая функция, поэтому далее ей передается строка в круглых скобках (`'gumby'`). Если вы создадите эти файлы, сохраните их и запустите 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 которым можно получить с помощью оператора `«.»`. Некоторые атрибуты – имена вызываемых функций, а другие – простые значения, которые представляют свойства объекта (например, имя персоны).

Импорт – широко используемое понятие в языке Python. Любой файл может импортировать функциональные возможности из любого другого файла. Например, файл *a.py* может импортировать файл *b.py*, чтобы иметь возможность вызывать его функцию, при этом файл *b.py* может в свою очередь импортировать файл *c.py*, чтобы получить доступ к другим функциональным возможностям, определенным в нем. Цепочка импортирования может уходить так глубоко, как это потребуются: в этом примере модуль `a` может импортировать модуль `b`, который импортирует модуль `c`, который в свою очередь может еще раз импортировать модуль `b`, и т. д.

Помимо самой крупной единицы в организационной структуре программы модули (и пакеты модулей, которые описываются в главе 20) также играют роль самой крупной единицы программного кода, доступного для повторного использования. Оформив программные компоненты в виде файлов модулей, вы сможете использовать их не только в оригинальной программе, но и в любых других программах, которые вам придется писать. Например, если после написания программы, структура которой изображена на рис. 18.1, вдруг обнаружится, что функция `b.spam` является универсальным инструментом, мы сможем использовать ее 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. 18.1. Некоторые из модулей, которые будут импортироваться вашими программами, входят непосредственно в состав языка Python, и вам не придется писать их.

Интерпретатор Python поставляется с обширной коллекцией дополнительных модулей, которая известна как *стандартная библиотека*. Эта коллекция насчитывает порядка 200 крупных модулей и содержит платформонезависимую поддержку распространенных задач программирования: интерфейсы операционных систем, организацию хранения 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 доступны и будут работать переносимым образом на большинстве платформ, на которых работает интерпретатор Python.

В примерах этой книги вы увидите несколько модулей стандартной библиотеки в действии, но за полной информацией вам следует обратиться к справочным руководствам по стандартной библиотеке языка Python, которые можно найти в инсталляции Python (в IDLE или в меню кнопки Пуск (Start) в операционной системе Windows) или в Интернете, по адресу: <http://www.python.org>.

При таком большом количестве модулей это действительно единственный способ получить представление о том, какие инструментальные средства имеются в наличии. Кроме этого, описание библиотеки инструментов Python можно найти в некоторых книгах, посвященных прикладному программированию, таких как «Programming Python», но, в отличие от книг, руководства распространяются бесплатно, их можно просматривать в любом веб-браузере (поскольку они распространяются в формате HTML) и к тому же они обновляются 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 инструкции импортирования составляют основу структуры программы, здесь мы подробнее рассмотрим этот вопрос, чтобы сделать представление о процессе импорта менее абстрактным.

Некоторые программисты на языке C любят сравнивать инструкцию `import` в языке Python с инструкцией `#include`, но они в корне неправы – импортирование в языке Python – это не просто включение текста одного файла в другой. Это самые настоящие операции времени выполнения, которые выполняют следующие 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 выполняются, только когда модуль впервые импортируется во время выполнения программы – все последующие операции импорта того же модуля пропускают эти действия и просто выбирают уже находящийся в памяти объект модуля.

1. Поиск

Прежде всего, интерпретатор должен определить местонахождение файла модуля, указанного в инструкции `import`. Обратите внимание, что имена файлов в инструкции `import` в примерах из предыдущих разделов указаны без расширения `.py` и без пути к каталогу: вместо записи в виде, например, `import c:\dir1\b.py` инструкция записывается просто – `import b`. Фактически, допускается указывать лишь простые имена – путь к каталогу и расширение файла должны быть опущены, потому что для поиска файла, соответствующего имени, указанному в инструкции `import`, интерпретатор использует стандартный *путь поиска модулей*.¹ Поскольку это основная часть операции импорта, которую необходимо знать программистам, мы рассмотрим ее более подробно.

Путь поиска модулей

В большинстве случаев можно положиться на автоматически организованные пути поиска модулей и вообще не настраивать этот путь. Однако, если вам потребуется импортировать модули из пользовательских каталогов, вам необходимо будет знать, как работает путь поиска файлов, чтобы настроить его соответствующим образом. В общих чертах пути поиска модулей в языке Python выбираются из объединенных

¹ В действительности синтаксис стандартной инструкции `import` не позволяет включать путь к файлу и его расширение. Для операции *импортирования пакетов*, которая будет рассматриваться в главе 20, 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 пакетам указываются относительно одного из каталогов, находящегося в пути поиска). 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.

данных следующих основных источников. Некоторые 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 следующем разделе. Первый и третий компоненты пути поиска определяются автоматически, 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 собственные каталоги с исходными текстами. Далее описывается, как интерпретатор Python использует эти компоненты пути:

Домашний каталог

В первую очередь интерпретатор ищет импортируемые файлы в домашнем каталоге. В зависимости от того, как была запущена программа, это будет либо каталог, где находится главный файл программы, либо каталог, из которого вы работаете в интерактивной оболочке. Поскольку поиск в первую очередь производится в этом каталоге, если программа целиком располагается в одном каталоге, все операции импорта будут выполнены автоматически, без необходимости настраивать путь поиска.

Каталоги в PYTHONPATH

После этого поиск производится во всех каталогах, перечисленных в переменной окружения `PYTHONPATH`, слева направо (если эта переменная вообще установлена). В двух словах, переменная окружения `PYTHONPATH` – это просто список имен каталогов, определяемых пользователем и системой, в которых 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 настройки при создании пути поиска модулей.

Поскольку интерпретатор пытается отыскать файлы сначала в домашнем каталоге, настройка этой переменной приобретает большое значение, только когда необходимо импортировать модули, размещающиеся в разных каталогах, – то есть когда импортируемый файл хранится в каталоге, отличном 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 необходимости 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 окружения `PYTHONPATH`.

Каталоги в файле .pth

Наконец, относительно новая особенность языка Python дает пользователям 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 представляют собой расширенную возможность, имеющую отношение к проблеме установки, и мы не будем здесь подробно их обсуждать.

Текстовый файл со списком каталогов помещается в соответствующий каталог и может играть примерно ту же роль, что и переменная окружения `PYTHONPATH`. Например, для расширения пути поиска модулей файл с именем *myconfig.pth* можно поместить в главный каталог, куда был установлен Python в Windows (например, *C:\Python25* или *C:\Python25\Lib\sitepackages*). Интерпретатор добавит в конец пути поиска модулей каталоги, перечисленные во всех строках файла, от первой до последней. Поскольку это файлы, а не параметры настройки командной оболочки, они могут применяться ко всем пользователям системы, а не только к одному пользователю или одной командной оболочке.

Эта особенность на практике более сложная, чем я описал. За дополнительной информацией обращайтесь к руководству по библиотеке языка Python (в частности, к описанию модуля *site*, входящего в стандартную библиотеку). Начинаящим я рекомендую использовать переменную окружения `PYTHONPATH` или единственный файл *.pth* и только в том случае, если возникает необходимость импортировать файлы из других каталогов. В приложении А вы найдете примеры наиболее распространенных способов расширения пути поиска файлов с помощью переменной окружения `PYTHONPATH` или файлов *.pth* на различных платформах.

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 могут добавляться дополнительные каталоги.

Например, в путь поиска вслед за каталогами из переменной окружения `PYTHONPATH` и перед каталогами стандартной библиотеки, интерпретатор может добавлять *текущий рабочий каталог* – каталог, откуда

была запущена программа. 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).¹ Так как от запуска к запуску программы текущий рабочий каталог может изменяться, при обычных условиях рабочий каталог не должен иметь значения для операций импорта.²

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 стандартной библиотеки). Этот список строк с именами каталогов представляет собой фактический путь поиска, используемый интерпретатором, – при выполнении 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 все каталоги, перечисленные в переменной окружения `PYTHONPATH` и в файлах `.pth`, и добавляя в начало списка домашний каталог главного файла программы (возможно как пустую строку).

Представление языком Python этого списка имеет два основных полезных результата. 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 проделали. Во-вторых, если вы понимаете, как формируется список, вы можете обеспечить 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; переменная окружения `PYTHONPATH` и файлы `.pth` обеспечивают возможность более долговременного хранения измененного пути.³

¹ Подробнее о запуске программ из командной строки говорится в главе 3.

² В главе 21 дополнительно обсуждается *относительно новый синтаксис инструкции импортирования*, добавленный в версии Python 2.5, – благодаря ему можно изменять путь поиска в инструкции `from` с помощью символов «.» (например, `from . import string`).

³ Некоторым программам действительно требуется изменять `sys.path`. Сценарии, которые выполняются на веб-сервере, например, обычно выполняются с привилегиями пользователя «nobody» с целью ограничить доступ к системе. Поскольку такие сценарии обычно не должны зависеть от значения переменной окружения `PYTHONPATH` для пользователя «nobody», они часто изменяют список `sys.path` вручную, чтобы включить в него необходимые каталоги до того, как будет выполнена какая-либо инструкция `import`. Обычно для этого бывает достаточно вызова `sys.path.append(dirname)`.

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, инструкция `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 главе 20).
- Скомпилированный модуль расширения, как правило, написанный на языке C или C++ и динамически присоединенный при импорте (например, *b.so* в Linux и *b.dll* или *b.pyd* в Cygwin и в Windows).
- Скомпилированный встроенный модуль, написанный на языке C и статически скомпонованный с интерпретатором Python.
- Файл ZIP-архива с компонентом, который автоматически извлекается при импорте.
- 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 импортирования 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 `import b` загружает некоторый модуль *b* в соответствии с настройками пути поиска модулей, а `b.attr` извлекает элемент модуля, будь то переменная 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 располагается раньше (левее) в пути поиска модулей, так как поиск в списке `sys.path` выполняется слева направо. Но что произойдет, если оба файла, *b.py* и *b.so*, находятся в одном и том же каталоге? В этом случае интерпретатор будет следовать 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 порядок выбора файлов.

Дополнительные возможности выбора модуля

Обычно операция импорта работает именно так, как описывается в данном разделе, — она отыскивает и загружает файлы, находящиеся на вашей машине. Однако вполне возможно переопределить большую часть того, что делает операция импорта, используя то, что называется *программными ловушками импорта*. Эти ловушки могут использоваться, чтобы придать операции импорта дополнительные полезные возможности, такие как загрузка файлов из архивов, расшифровывание и т. д. Фактически сам интерпретатор Python использует эти ловушки, чтобы обеспечить возможность извлечения импортируемых компонентов из ZIP-архивов — заархивированные файлы автоматически извлекаются во время импорта, когда в пути поиска выбирается файл с расширением *.zip*. За дополнительной информацией обращайтесь к описанию встроенной функции `__import__` в руководстве по стандартной библиотеке Python — настраиваемому инструменту, которым в действительности 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 используются достаточно редко. Система Psycopy (глава 2) обеспечивает куда более существенный прирост в скорости выполнения.

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 компиляции 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 исходным текстом, он просто загрузит байт-код. Другими словами, этап компиляции пропускается, если можно ускорить запуск программы. Если вы измените исходный программный код, Python автоматически скомпилирует байт-код при следующем запуске программы. Кроме того, вы можете распространять свою программу исключительно в виде файлов с байт-кодом, чтобы не передавать файлы с исходными текстами.

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 импортированных файлов сохраняется в файлах для ускорения будущих операций импорта.

Главные файлы программ часто планируется исполнять непосредственно и нигде 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 импортироваться, поэтому для них создаются соответствующие файлы *.pyc*. Чтобы разобраться с тем, как это получается, читайте обсуждение специальных атрибутов `__name__` и `__main__` в главе 21.

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, будут создавать атрибуты конечного объекта модуля. Таким образом, этот этап выполнения создает все инструменты, которые определяются модулем. Например, 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 можно будет наблюдать результат работы инструкций `print` на верхнем уровне модуля. Инструкции `def` просто определяют объекты для последующего использования.

Как видите, во время импорта выполняется достаточно много работы – производится 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 программы. При повторных попытках импортировать модуль все три шага просто пропускаются, и повторно используется модуль, уже загруженный в память.¹ Если вам потребуется еще раз импортировать файл, который уже был загружен (например, чтобы обеспечить поддержку настроек, выполняемых пользователем), воспользуйтесь функцией `reload`, с которой мы встретимся в следующей главе.

¹ В реальности интерпретатор хранит загруженные модули во встроенном словаре `sys.modules`, который проверяется в начале операции импортирования, чтобы определить, не был ли уже загружен указанный модуль. Если вам потребуется увидеть, какие модули были загружены, импортируйте модуль `sys` и выведите результат работы метода `sys.modules.keys()`. Подробнее об этой внутренней таблице рассказывается в главе 21.

Стороннее программное обеспечение: `distutils`

Настройка пути поиска модулей, описание которой приводится в этой главе, в первую очередь касается программного кода, который вы пишете самостоятельно. Сторонние расширения для Python обычно используют для автоматической установки самих себя такой инструмент, как `distutils`, входящий в состав стандартной библиотеки, поэтому для использования такого программного кода не требуется выполнять настройку пути поиска модулей.

Системы, использующие `distutils`, обычно поставляются со сценарием `setup.py`, который запускается для установки таких систем – этот сценарий 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 с помощью `distutils` обращайтесь к стандартному набору руководств 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 производится установка). Кроме того, обратите внимание на развивающуюся систему `eggs`, распространяемую с открытыми исходными текстами, которая добавляет возможность проверки зависимостей для установленного 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 поиск файла модуля в пути поиска модулей, компиляция в байт-код и выполнение 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 каталогов, отличных от домашнего каталога и от каталогов стандартной библиотеки.

Как показала эта глава, операция импортирования и модули являются основой 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 импортирования определяется с по-

мощью пути поиска модулей, а модули определяют атрибуты, использующиеся за пределами этих модулей.

Конечно, основное назначение операции импорта и модулей состоит в том, чтобы образовать структуру программы, логика которой подразделяется на самостоятельные программные компоненты. Программный код в одном модуле 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, поставьте себе задачу ответить на контрольные вопросы к главе.

Закрепление пройденного

Контрольные вопросы

1. Каким образом файл с исходным программным кодом модуля превращается в объект модуля?
2. Зачем может потребоваться настраивать значение переменной окружения `PYTHONPATH`?
3. Назовите четыре основных компонента, составляющих путь поиска модулей.
4. Назовите четыре типа файлов, которые могут загружаться операцией импортирования.
5. Что такое пространство имен, и что содержит пространство имен модуля?

Ответы

1. Файл с исходными текстами модуля автоматически превращается в объект модуля в результате выполнения операции импортирования. С технической 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 необходимости 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 модулей, являются: домашний каталог главного файла программы (каталог, в котором он находится), все каталоги, перечисленные

в переменной окружения `PYTHONPATH`, каталоги стандартной библиотеки и все каталоги в файлах с расширением `.pth`, размещенных в стандартных местах. Из них доступны для настройки переменная окружения `PYTHONPATH` и файлы с расширением `.pth`.

4. Интерпретатор Python может загружать файлы с исходными текстами (`.py`), файлы с байт-кодом (`.pyc`), файлы расширений, написанных на языке C (например, файлы с расширением `.so` в Linux или с расширением `.dll` в Windows), или каталог с указанным именем, в случае импортирования пакета. Операция импортирования может также загружать менее обычные файлы, такие как компоненты из архивов в формате ZIP, классы Java 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 содержит все имена, присваивание значений которым производится программным 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. Пространство имен модуля может изменяться с помощью операций присваивания из других файлов, которые импортируют данный модуль, хотя это и не приветствуется (подробнее об этом рассказывается в главе 16).

19

Основы программирования модулей

Теперь, когда мы рассмотрели общие идеи, лежащие в основе модулей, обратимся к простому примеру модулей в действии. Модули в языке Python создаются *очень просто* – это всего лишь файлы с программным кодом на языке Python, которые создаются с помощью текстового редактора. Вам не требуется 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, поэтому 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 присваивание на верхнем уровне модуля, станут его атрибутами (именами, ассоциированными 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 единственным 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 собираетесь импортировать их. Для главных файлов программ, которые будут запускаться, но не будут импортироваться, имена не обязательно должны иметь расширение *.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.

Поскольку имена модулей внутри программы превращаются в имена переменных (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 выполнить 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 соответствовать требованиям, предъявляемым к именам переменных и представленным в главе 11, — они могут, например, содержать только алфавитные символы, 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 поиска модулей и добавляя *.py* или другое расширение в конец. Например, в конечном 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 обычно используются для создания библиотек, используемых сценариями на языке Python. Когда модули расширений импортируются программным кодом на языке Python, они выглядят и ведут себя точно так же, как обычные модули, написанные на языке Python, — они импортируются инструкцией `import` и предоставляют 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 «Programming 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, ес-

ли он еще не был загружен. Главное различие этих инструкций заключается в том, что инструкция `import` загружает модуль целиком, поэтому при обращении к именам в модуле их необходимо дополнять именем модуля. Инструкция `from`, напротив, загружает (или копирует) из модуля отдельные имена.

Посмотрим, что это означает на примере программного кода. Все следующие примеры вызывают функцию `printer`, определенную во внешнем модуле `module1.py`, но делают это различными способами.

Инструкция `import`

В первом примере имя `module1` служит двум различным целям — оно идентифицирует 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!
```

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`

Напротив, инструкция `from` копирует имена из области видимости одного файла 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 импортируемое имя копируется в область видимости, в которой находится сама инструкция `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 присваиваются значения на верхнем уровне указанного модуля.

В этом случае точно так же можно использовать скопированное имя `printer`, не предвзяя 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 копирование всех имен в импортируемом модуле в область видимости, откуда производится импорт. По сути происходит совмещение пространств имен модулей, что позволяет нам меньше вводить с клавиатуры.

Как видите, модули действительно легко использовать. Чтобы еще лучше понять, что происходит в действительности, когда вы определяете и используете модули, рассмотрим некоторые их свойства более подробно.

Импорт выполняется только один раз

Один из самых типичных вопросов, которые задают начинающие программисты, начав использовать модули: «Почему операция импорта перестает работать?». 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 `import` или `from`. Реализовано такое поведение преднамеренно, потому что импортирование – это дорогостоящая операция и интерпретатор выполняет ее всего один раз за все время работы. Последующие операции импорта просто получают объект уже загруженного модуля.

Из этого следует: так как программный код на верхнем уровне модуля выполняется всего один раз, это обстоятельство можно использовать для инициализации переменных. Рассмотрим пример модуля *simple.py*:

```
print 'hello'
spam = 1          # Инициализировать переменную
```

В этом примере инструкции `print` и `=` выполняются при первой операции импортирования модуля, 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 перезапуску программного кода модуля – они просто получают объект модуля из внутренней таблицы модулей интерпретатора. В результате повторная инициализация 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 программный 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 выполняемыми инструкциями, а не объявлениями времени компиляции. Они могут вкладываться в условные инструкции `if`, присутствовать в объявлениях функций `def` и т. д., и они не имеют никакого эффекта, пока интерпретатор не достигнет их в ходе выполнения программы. Другими словами, импортируемые модули и имена в них не будут доступны, пока не будут выполнены соответствующие инструкции `import` или `from`. Кроме того, подобно инструкции `def`, `import` и `from` – это явные операции присваивания:

- Инструкция `import` присваивает ссылку на объект модуля единственной переменной.
- Инструкция `from` присваивает одно или более имен объектам с теми же именами в другом модуле.

Все, что уже обсуждалось ранее, в равной степени применимо и к модулям. Например, имена, копируемые инструкцией `from`, становятся ссылками на разделяемые объекты – как и в случае с аргументами функций, переприсваивание полученному имени не оказывает воздействия на модуль, откуда это имя было скопировано, но изменение *изменяемого объекта* может оказывать воздействие на объект в модуле, откуда он был импортирован. Для иллюстрации 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 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 инструкция `from`, вернитесь к рис. 16.2 (передача аргументов функциям) и мысленно замените слова «вызывающая программа» и «функция» на «импортируемый модуль» и «импортирующий модуль». Эффект тот же самый, за исключением 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 интерактивной 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 никакой 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 16. Поскольку изменение переменных в других модулях, как в данном случае, часто является источником проблем (и следствием неудачного проектирования), мы еще вернемся к этому приему позднее в этой части книги. Обратите внимание, что изменение элемента `y[0]` в предыдущем примере – это нечто иное; изменяется объект, а не имя.

Эквивалентность инструкций `import` и `from`

Обратите внимание: в предыдущем примере после инструкции `from` нам потребовалось 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 *)` эквивалентная последовательность действий та же самая, только при этом копируются 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, сколько 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 незначительным.

Потенциальные проблемы инструкции `from`

Инструкция `from` делает местоположение переменных менее явным и очевидным (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 библиотеки, например.

Суть проблемы состоит в том, что инструкция `from` способна повреждать пространства имен, по крайней мере, в принципе – если использовать ее для импортирования переменных, когда существуют одноименные переменные в имеющейся 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 (имя `module.attr` не конфликтует с именем `attr` в текущей области видимости). Пока вы понимаете и контролируете все, что может происходить при использовании 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 проблемы, когда используется в комбинации с функцией `reload`, т. е. импортированные имена могут ссылаться на предыдущие версии объектов. Кроме того, инструкция в форме `from module import *` действительно может повреждать пространства 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 несет возможность разделения пространств имен. Мы будем рассматривать эти проблемы более подробно в разделе «Типичные ошибки при работе с модулями» в конце этой части книги (глава 21).

Пожалуй, лучший совет, который можно дать, – в общем отдавать предпочтение инструкции `import` для простых модулей, явно перечислять необходимые переменные 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 находят ее удобной для организации доступа к модулям.

Когда необходимо использовать инструкцию `import`

Единственное, когда необходимо вместо инструкции `from` использовать инструкцию `import`, – когда требуется использовать одно и то же имя, присутствующее в двух разных модулях. Например, когда два файла по-разному определяют 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 инструкцию `from` использовать нельзя, потому что в результате вы получите единственное 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 вменяющего модуля делает имена уникальными:

```
# 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 сталкиваться с ним на практике.

Пространства имен модулей

Модули будут, вероятно, более понятны, если представлять их, как простые пакеты имен – то есть место, где определяются переменные, которые должны быть доступны остальной системе. С технической точки зрения каждому модулю соответствует отдельный файл, и интерпретатор создает объект модуля, 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 атрибутами. 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 пределов модуля можно как `M.X`. Кроме того, имя `X` становится глобальной переменной для программного кода внутри *M.py*, но нам необходимо более формально объяснить понятия загрузки модуля и областей видимости, чтобы понять почему:

- **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 атрибуты модуля.** Во время импортирования инструкции присваивания выполняемые на верхнем уровне файла и не вложенные в инструкции `def` или `class` (например, `=`, `def`) создают атрибуты объекта модуля – при присваивании имени сохраняются в пространстве имен модуля.
- **Доступ к пространствам имен модулей можно получить через атрибуты `__dict__` или `dir(M)`.** Пространства имен модулей, создаваемые операцией импортирования, представляют собой словари – доступ к ним можно получить через встроенный атрибут `__dict__`, ассоциированный с модулем, и с помощью функции `dir`. Функция `dir` – это примерный эквивалент отсортированного списка ключей атрибута `__dict__`, но она включает унаследованные имена классов, может возвращать не полный список и часто изменяется от версии к версии.
- **Модуль – это единая область видимости (локальная является глобальной).** Как мы видели в главе 16, имена на верхнем уровне модуля подчиняются тем же правилам обращения/присваивания, что имена в функциях, только в этом случае локальная область видимо-

сти совпадает с глобальной (точнее, они следуют тому же правилу LEGB поиска в областях видимости, с которым мы познакомились в главе 16, только без уровней поиска L и E). Но в модулях *область видимости* модуля после загрузки модуля превращается в атрибут-словарь *объекта* модуля. В отличие от функций (где локальное пространство имен существует только во время выполнения функции), область видимости файла модуля превращается в область видимости атрибутов объекта модуля и нигде не исчезает после выполнения операции импортирования.

Ниже эти понятия демонстрируются в программном коде. Предположим, что 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 пространстве 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 инструкция `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 0x012B1830>

>>> module2.klass
<class module2.klass at 0x011C0BA0>
```

Здесь именам `sys`, `name`, `func` и `klass` были присвоены значения во время выполнения инструкций модуля, поэтому они стали атрибутами после завершения операции импортирования. О классах мы будем говорить

в шестой части книги, но обратите внимание на атрибут `sys` – инструкции `import` действительно *присваивают* объекты модулей именам, а любая операция присваивания на верхнем уровне файла создает атрибут модуля.

Внутри интерпретатора пространства имен хранятся в виде объектов словарей. Это самые обычные объекты словарей с обычными методами. Обратиться к словарю пространства имен модуля можно через атрибут `__dict__` модуля:

```
>>> module2.__dict__.keys()
['_file_', 'name', '__name__', 'sys', '__doc__', '__builtins__',
'klass', 'func']
```

Имена, которые были определены в файле модуля, становятся ключами внутреннего словаря, таким образом, большинство имен здесь отражают операции присваивания на верхнем уровне в файле. Однако интерпретатор Python добавляет 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 каталогу).

Дополнение имен атрибутов

После ознакомления с модулями мы должны поближе рассмотреть понятие *дополнения* имен. В языке Python для доступа к атрибутам любого объекта используется синтаксис дополнения имени `object.attribute`.

Дополнение имени – это в действительности выражение, возвращающее значение, присвоенное имени атрибута, ассоциированного с объектом. Например, выражение `module2.sys` в предыдущем примере возвращает значение атрибута `sys` в объекте `module2`. Точно так же, если имеется встроенный объект списка `L`, выражение `L.append` вернет метод `append`, ассоциированный с этим списком.

Итак, какую роль играет дополнение имен атрибутов с точки зрения правил, рассмотренных нами в главе 16? В действительности – никакую: это совершенно независимые понятия. Когда вы обращаетесь к именам, используя дополнение, 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 произведен поиск этого имени в текущих областях видимости (следуя правилу **LEGB**).

Полные 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 областях видимости).

Полные пути

Имя $X.Y.Z$ означает, что будет произведен поиск имени Y в объекте X , а затем поиск имени Z в объекте $X.Y$.

Общий случай

Дополнение имен применимо ко всем объектам, имеющим атрибуты: модулям, классам, расширениям типов на языке C и т. д.

В шестой части книги мы увидим, что дополнение имен для классов имеет немного большее значение (здесь также имеет место то, что называется *наследованием*), 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 автоматически не получите доступ к именам в другом файле, независимо от вида импортируемого модуля и вызовов функций в вашей программе. Смысл переменной всегда определяется местоположением операции присваивания в программном коде, а для обращения к атрибутам всегда необходимо явно указывать объект.

Например, рассмотрим два следующих простых модуля. Первый, *moda.py*, определяет переменную X , которая является глобальной только для программного 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 переменную 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 является файл, вмещающий ее, независимо от того, из какого модуля она была вызвана:

```
% python modb.py
11 99
```

Другими словами, операция импортирования никогда не изменяет область видимости 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 функциях, если только они физически не вложены друг в друга.
- Программному коду модуля никогда не будут доступны имена, определенные в других модулях, если только они явно не были импортированы.

Это поведение является частью понятия *лексической области видимости* – в языке Python области видимости, доступные части программного кода, полностью 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 видимости, но она дает возможность обращаться к вложенным областям видимости. Используя полные 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* определяет единственное глобальное имя и атрибут операций присваивания:

```
X = 3
```

Файл *mod2.py* определяет свою переменную *X*, затем импортирует модуль *mod3* и использует дополнение имени, чтобы получить доступ к атрибуту импортированного модуля:

```
X = 2
import mod3

print X,          # Моя глобальная переменная X
print mod3.X      # Глобальная переменная X из модуля mod3
```

Файл *mod3.py* также определяет свою собственную переменную *X*, затем импортирует модуль *mod2* и получает значения атрибутов обоих модулей:

```
X = 1
import mod2

print X,          # Моя глобальная переменная X
print mod2.X,     # Переменная X из модуля mod2
print mod2.mod3.X # Переменная X из модуля mod3
```

¹ Некоторые языки программирования подразумевают иной порядок действий и реализуют *динамические области видимости*, когда области видимости в действительности могут зависеть от вызовов функций во время выполнения программы. Это усложняет программный код, потому что смысл переменной может изменяться с течением времени.

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 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 видимости 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 (инструкция `import` выполняет операцию присваивания). Для таких путей, как `mod2.mod3.X`, интерпретатор Python выполняет вычисления слева направо, извлекая атрибуты из объектов.

Обратите внимание: в `mod1` можно вставить инструкцию `import mod2` и затем использовать 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 всего один раз за все время работы программы. Чтобы принудительно повторно загрузить 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 динамичной. В двух словах:

- При вызове операции импортирования (с помощью как инструкции `import`, так 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 программного кода в этом случае не происходят.
- Функция `reload` принудительно выполняет повторную загрузку уже загруженного модуля и запускает его программный код. Инструкции присваивания, выполняемые при повторном запуске, будут изменять существующий объект модуля.

Для чего вся эта суета вокруг повторной загрузки модулей? Функция `reload` позволяет изменять части программы, не останавливая всю программу. Благодаря функции `reload` эффект от изменений в программном коде можно наблюдать сразу же после внесения этих изменений. Повторная загрузка модулей поможет не во всех ситуациях, но она позволит существенно сократить цикл разработки. 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.

Язык Python относится к языкам интерпретирующего типа (более или менее), поэтому в нем отсутствуют этапы компиляции/компоновки, необходимые, чтобы запустить программу, например, на языке C: модули загружаются динамически уже запущенной программой. Возможность повторной загрузки обеспечивает 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 Python, – скомпилированные модули расширений, написанные на таких языках, как C, тоже могут динамически загружаться во время работы программы, но их нельзя загрузить повторно.

Основы использования функции `reload`

В отличие от инструкций `import` и `from`:

- `reload` – это не инструкция, а встроенная функция.
- Функции `reload` передается существующий объект модуля, а не имя.

Функция `reload` ожидает получить объект, поэтому к моменту ее вызова модуль уже должен быть успешно импортирован (если операция импорта оказалась неудачной из-за синтаксических или каких-либо других ошибок, вам может потребоваться повторить ее, прежде чем можно будет повторно загрузить модуль). Кроме того, синтаксис инструкции `import` и функции `reload` отличается: аргумент должен передаваться функции `reload` в круглых скобках, а инструкции `import` – без них. Повторная загрузка модуля выполняется примерно следующим образом:

```
import module                # Первоначальное импортирование
...используются атрибуты модуля...
...                          # Теперь выполняются изменения в файле модуля
...
reload(module)               # Загрузить обновленный модуль
...используются атрибуты модуля...
```

Это типичный случай, когда вы импортируете модуль, затем изменяете исходный программный код в текстовом редакторе, а потом повторно загружаете его. Когда вы вызываете функцию `reload`, интерпретатор повторно читает файл с исходными текстами и выполняет инст-

рукции, 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 создает его повторно. Вследствие этого все ссылки на объект модуля, имеющиеся в программе, автоматически будут учитывать изменения, произошедшие в результате повторной загрузки. А теперь подробнее о том, как происходит повторная загрузка:

- **Функция `reload` запускает новый программный код в файле модуля в текущем 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 новыми 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, использовавших 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 только на тех клиентов, которые будут использовать инструкцию `from` в будущем.** Клиенты, которые 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, полученные 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. Повторная загрузка может использоваться в различных других случаях (смотрите врезку «Придется держать в уме: повторная загрузка модулей»), но мы рассмотрим лишь самый простой пример. Во первых, в текстовом редакторе создайте файл модуля с именем *changer.py* и добавьте в него следующее содержимое:

```
message = "First version"

def printer():
    print message
```

Этот модуль создает и экспортирует два имени – одно связано со строкой, а другое является функцией. Теперь запустите интерпретатор Python, импортируйте модуль и вызовите функцию, которую он экспортирует. Функция выведет значение глобальной переменной `message`:

```
% python
>>> import changer
>>> changer.printer()
First version
```

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 перезагрузки модулей являются системы, которые на запуске соединяются с серверами сети.

Эта возможность также может использоваться в приложениях с графическим 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 может вызывать повторную загрузку программного кода на языке Python без остановки всего приложения). За более подробным описанием повторной загрузки обработчиков событий в графическом интерфейсе и встроенном программном коде на языке Python обращайтесь к книге «Programming Python».

Как правило, повторная загрузка позволяет программам реализовать высокодинамичные интерфейсы. Например, Python часто используется как язык для *настройки* больших систем – пользователи могут настраивать программные продукты, изменяя программный код на языке Python без необходимости перекомпилировать весь продукт (и даже не имея исходных текстов этого продукта). В таких условиях программный код на языке Python уже сам по себе добавляет динамичности.

Тем не менее, чтобы обеспечить еще более высокую динамичность, такие системы могут автоматически выполнять повторную загрузку настроечного 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 останавливать 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 перезагрузка модулей является простым и удобным средством выполнения настроек.

Не закрывая интерактивную оболочку интерпретатора, отредактируйте файл модуля в другом окне:

```
...измените файл changer.py, не останавливая интерактивный сеанс...  
% vi changer.py
```

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, чтобы выполнить обновленный программный код. Обратите внимание: в следующем 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  
  
>>> reload(changer) # Принудительная загрузка/выполнение нового кода  
<module 'changer'>  
>>> changer.printer() # Теперь будет запущена новая версия  
reloaded: After editing
```

Обратите внимание, что функция `reload` в действительности возвращает объект – обычно ее результат игнорируется, но поскольку интерактивная оболочка автоматически выводит результат выражения, интерпретатор вывел результат в виде строки `<module 'name'>`.

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 рассмотрели 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 сведения о модели импортирования – об импортирова-

нии 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 систем и позволяющий избежать конфликтов между одинаковыми именами модулей. Но прежде чем двинуться дальше, постарайтесь ответить на контрольные вопросы по представленным здесь идеям.

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 инструкциями на языке Python; любой файл с исходным программным кодом автоматически становится модулем – нет никаких синтаксических конструкций для его объявления. Можно также создать модуль, написав программный 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 позволяет использовать импортированные имена напрямую (`name`), без дополнения их именем модуля (`module.name`).
3. По умолчанию модуль импортируется один раз за все время выполнения программы. Функция `reload` принудительно выполняет повторное импортирование. Она часто используется, чтобы загрузить новую версию исходного программного кода модуля в процессе разработки и в случаях динамической 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 имена вмещающих модулей, эти два имени будут уникальны.

5. Инструкция `from` может делать непонятным смысл переменной (в каком модуле она определена), вызывать проблемы при использовании функции `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 повреждениям пространств имен и скрывать смысл переменных – эту форму инструкции следует использовать с большой осторожностью.
6. А какая ласточка имеется в виду? Африканская или европейская?

20

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. Однако возможности импортирования модулей немного богаче, чем я предлагал вам считать до настоящего момента.

Помимо возможности импортировать имя модуля существует возможность импортировать имена каталогов. Каталог, как говорят на языке Python, является *пакетом*, поэтому такая операция импортирования называется *импортированием пакетов*. 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 системах 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 нескольких файлов программ с одинаковыми именами, установленных на одном компьютере.

Основы операции импортирования пакетов

Так как же работает импортирование пакетов? В инструкциях `import`, там, где вы указывали имя простого файла, можно указать список имен в пути к каталогу, разделяя их символами точки:

```
import dir1.dir2.mod
```

То же самое относится и к инструкции `from`:

```
from dir1.dir2.mod import x
```

Предполагается, что такой «точечный» путь в этих инструкциях соответствует 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.pyс, mod.so и т. д.
```

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.

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, специфичный для текущей платформы, как, например, *C:\dir1*, *My Documents.dir2* или *../dir1* – это недопустимый синтаксис. Вместо этого платформозависимый синтаксис используйте в настройках путей поиска модулей, именуя необходимые каталоги-контейнеры.

Так, в предыдущем примере *dir0* – это имя каталога, которое требуется добавить в путь поиска модулей и которое может иметь произвольную длину и путь, с учетом специфики используемой платформы, ведущий к каталогу *dir1*. Вместо 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 платформозависимые пути к каталогам, которые ведут к самым левым именам

в цепочках, представленных в инструкциях `import`, а сами инструкции `import` предоставляют окончание пути к каталогам платформонезависимым способом.¹

Файлы пакетов `__init__.py`

Если вы решили использовать импортирование пакетов, существует еще одно условие, которое необходимо будет соблюдать: каждый каталог в пути, указанном в инструкции импортирования пакета, должен содержать файл с именем `__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 говоря, 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 т. д.).

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
```

¹ Символ точки как разделитель имен каталогов был выбран не только для обеспечения независимости от используемой платформы, но и потому, что пути в инструкциях `import` в действительности становятся вложенными объектами пути. Этот синтаксис 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.

Файлы `__init__.py` могут содержать программный код на языке Python, как любые другие файлы модулей. Отчасти они являются объявлениями для интерпретатора и могут вообще ничего не содержать. Эти файлы, будучи объявлениями, 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 выбирать каталоги, которые не имеют никакого отношения к вашему программному 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 реализации поведения инструкций `from *` (то есть `from ... import *`), когда они используются для импортирования каталогов:

Инициализация пакета

Когда интерпретатор Python импортирует каталог в первый раз, он автоматически запускает программный код файла `__init__.py` этого каталога. По этой причине обычно в эти файлы помещается программный код, выполняющий 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 т. д. 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 завершения 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 каталогам, в которых отсутствуют настоящие файлы модулей.

Поведение инструкции `from *`

В качестве дополнительной особенности, в файлах `__init__.py` можно использовать списки `__all__`, чтобы определить, что будет импортироваться из каталога инструкцией `from *`. (Мы познакомимся со списком `__all__` в главе 21.) Список `__all__` в файлах `__init__.py` представляет собой список имен субмодулей, которые должны импортироваться, когда в инструкции `from *` указывается имя пакета (каталога). Если список `__all__` отсутствует, инструкция `from *` не будет автоматически загружать субмодули, вложенные в каталог, — она загрузит только имена, определяемые инструкциями присваивания в файле `__init__.py`, включая любые субмодули, явно импор-

тируемые программным кодом в этом файле. Например, инструкция `from submodule import X` в файле `__init__.py` создаст имя `X` в пространстве имен каталога.

Эти файлы можно оставить пустыми, если вам не требуется выполнять специальных действий. Однако для успешного выполнения операции импортирования каталогов они должны существовать обязательно.

Пример импортирования пакета

Рассмотрим практический пример программного кода, который демонстрирует, как используются файлы инициализации и пути к каталогам. Следующие три файла располагаются в каталоге *dir1* и в подкаталоге *dir2*:

```
# 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 каталога (то есть домашнего каталога программы) или подкаталогом одного из каталогов, перечисленных в пути поиска модулей (с точки зрения реализации: каталога, входящего в список `sys.path`). В любом из этих случаев для каталога, вмещающего подкаталог *dir1*, не требуется наличие файла `__init__.py`.

Инструкции `import` выполняют файл инициализации в каждом каталоге, который присутствует 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 `reload` 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
>>>
>>> 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`, становится *цепочкой вложенных объектов*. Здесь `mod` — это объект, вложенный в объект `dir2`, который в свою очередь вложен в объект `dir1`:

```

>>> dir1
<module 'dir1' from 'dir1\__init__.pyc'>
>>> dir1.dir2
<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 присваивается объект модуля, пространство имен которого инициализируется всеми инструкциями присваивания в файле `__init__.py`, находящемся в этом каталоге. 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` может 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 предыдущего раздела, например, приходилось каждый раз вводить полный путь от `dir1`, когда необходимо было обратиться к переменной `z`. Если попытаться 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 инструкцию `from`, чтобы избежать необходимости ввода полного имени при каждом обращении к нему. Пожалуй, еще более важно, что

если вы когда-нибудь выполните реструктуризацию дерева каталогов, то в случае использования инструкции `from` достаточно будет обновить только саму инструкцию, тогда как в случае использования инструкции `import` придется обновлять все обращения к именам в изменившемся пакете. Расширение `import as`, обсуждаемое 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
```

Зачем 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, особенно в крупных программах: они делают операцию импортирования более информативной, выступают в роли организационного инструмента, упрощают поиск файлов модулей и способны разрешать возникающие неоднозначности.

Так как операция импортирования пакетов содержит некоторые сведения о структуре каталогов, где находятся файлы программы, они, в первую очередь, упрощают поиск файлов и служат организационным инструментом. 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 пакетов делает более очевидной роль, которую играют пакеты, что обеспечивает более высокую удобочитаемость программного кода. Например, обычная операция 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 переменной 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 производится относительно общего корневого каталога, вам достаточно будет добавить единственную запись в путь поиска модулей: общий корневой каталог. Наконец, операция импортирования пакетов способна разрешать неоднозначности за счет явного и точного указания импортируемых файлов. В следующем разделе эта роль исследуется более подробно.

История о трех программах

Единственный случай, когда операция импортирования действительно необходима, – это разрешение неоднозначностей, которые могут возникать, когда на одной машине установлено множество программ, содержащих файлы с одинаковыми именами. В определенной степени это проблема установки программ, но она может стать источником беспокойств в обычной практике. Давайте рассмотрим гипотетическую ситуацию, чтобы проиллюстрировать эту проблему.

Предположим, что программист разработал программу на языке Python, которая содержит файл именем `utilities.py`, хранящий вспомогательный программный код, и файл `main.py`, используемый для запуска программы. Все файлы программы вызывают инструкцию `import utilities` для загрузки и использования общего программного кода. Программа распространяется в виде единого архива в формате `.tar` или `.zip`, содержащего все файлы программы, и при установке 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 самом компьютере, где уже была установлена первая программа, ее файлы были распакованы в новый каталог с именем `system2`, чтобы не перезаписать одноименные 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. Фактиче-

ски для этих программ вам даже не нужно настраивать путь поиска модулей, потому что интерпретатор 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 запускается 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 импортировать 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 программный код, а для программного кода на языке Python вполне естественно, когда он используется многократно. В такой ситуации вам необходима возможность из своего программного кода, хранящегося в третьем каталоге, загрузить один из двух файлов:

```
import utilities
utilities.func('spam')
```

Теперь проблема начинает вырисовываться. Чтобы вообще выполнить эту работу, 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 просматривается 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 импортирования, 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 устанавливать программы как плоские списки файлов в независимые каталоги, можно установить их в *подкаталоги* с общим корнем. Например, можно было бы организовать установку всего программного кода из этого примера в виде следующей 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 модулей. Если теперь выполнять импортирование относительно этого общего корня, можно будет с помощью операции импортирования пакетов импортировать любой файл из любой программы – использование имени вмещающего каталога делает путь (а значит, и ссылку на модуль) уникальным. Фактически можно даже импортировать *обе* утилиты сразу в одном и том же модуле, при условии, 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 ссылок на модули.

Обратите внимание, что вместо инструкции `from` необходимо использовать инструкцию `import`, только если вам необходимо получить доступ к двум или более одноименным атрибутам. Если бы имена вызываемых здесь функций различались, можно было бы использовать инструкцию `from`, чтобы избежать необходимости 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 программы импортировать каталог пакета.

С точки зрения реализации, каталог `system3` не обязательно должен находиться в каталоге `root` – здесь должны находиться только каталоги, откуда производится импорт 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 программах поиск модулей производится в первую очередь в их *домашних* каталогах, добавление общего корневого каталога в путь по-

Придется держать в уме: пакеты модулей

Теперь, когда пакеты стали стандартной частью Python, часто можно встретить крупные расширения сторонних разработчиков, распространяемые не как плоский список модулей, а как набор каталогов с пакетами. Например, пакет расширений *win32all* для Python в операционной системе Windows был одним из первых, кто перешел на сторону победителя. Многие вспомогательные модули этого пакета располагаются в пакетах, импортируемых посредством указания пути. Например, чтобы загрузить набор инструментальных средств для работы с технологией 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 организованы 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. Если вы создаете каталоги пакетов, в конечном итоге у вас будет возможность импортировать их.

иска модулей никак не отражается на программном коде в *system1* и *system2* – они по-прежнему могут использовать инструкции `import utilities` и ожидать получить в ответ свои собственные файлы. Кроме того, если вы предусмотрительно будете устанавливать все программы на языке Python в общий корневой каталог, как в данном примере, настройка пути поиска станет элементарной: вам достаточно будет один раз добавить в него общий корневой каталог.

В заключение

В этой главе была представлена модель импортирования пакетов – обязательный, но удобный способ явно указать путь к каталогам с модулями. В инструкциях 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 поиска (если все каталоги, откуда производится импорт, вложены

в один общий корневой каталог), а также позволяют разрешать возникающие неоднозначности в тех случаях, когда существует более одного модуля с одним и тем же именем (наличие имен каталогов в операциях импортирования позволяет обеспечить уникальность имен модулей).

В следующей главе мы исследуем несколько более сложных тем, имеющих отношение к модулям, таких как синтаксис относительного импорта и переменная режима использования `__name__`. Как обычно, мы заканчиваем эту главу серией контрольных вопросов, чтобы проверить, насколько хорошо вы запомнили 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 инструкцию `import`?

Ответы

1. Файлы `__init__.py` служат для объявления и инициализации пакета, – интерпретатор 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 каталогах пакетов обязательно – вы не сможете импортировать пакеты при отсутствии 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 присутствовать только в одном месте – в инструкции `from` или `import`.
3. Каждый каталог, перечисленный в инструкции `import` или `from`, должен содержать 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.

21

Дополнительные возможности модулей

Эта глава завершает пятую часть книги, представляя коллекцию более сложных тем, имеющих отношение к модулям, – синтаксис относительного импорта, сокрытие данных, модуль `__future__`, переменная `__name__`, изменение списка `sys.path` и т. д. и, кроме того, содержит перечень наиболее типичных ошибок и упражнения, завершающие эту часть книги. Как и функции, модули наиболее эффективны, когда должным образом определены их интерфейсы, поэтому в данной главе представлен краткий обзор концепций проектирования модулей, некоторые из которых мы исследовали в предыдущих главах.

Несмотря на слово «дополнительные» в заглавии этой главы, некоторые из обсуждаемых здесь возможностей (такие, как приемы работы с переменной `__name__`) используются достаточно широко, поэтому обязательно ознакомьтесь с ними, прежде чем переходить к классам в следующей части книги.

Соккрытие 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 предотвратить возможность изменения имен в модуле извне, если в этом появится необходимость.

Соккрытие 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 пуристы возражают против такого либерального отношения к сокрытию данных, утверждая в связи с этим, что в языке

Python отсутствует возможность инкапсуляции. Инкапсуляция в языке Python скорее относится к организации пакетов, чем к возможности накладывать ограничения.

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 перезаписывание, когда клиент выполняет импортирование модуля инструкцией `from *`. Этот прием на самом деле предназначен только для минимизации загрязнения пространства имен — так как инструкция `from *` копирует все имена, импортирующий модуль может получить больше, чем предполагал (включая имена, которые перезапишут имена импортирующего модуля). Символы подчеркивания не являются объявлением «частных» данных: вы по-прежнему можете видеть эти имена и изменять их с помощью других форм импортирования, таких как инструкция `import`.

Альтернативный способ достижения эффекта сокрытия данных, напоминающий соглашение об именовании `_X`, заключается в присвоении на верхнем уровне 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, доступные для копирования, тогда как соглашение `_X` идентифицирует имена, недоступные 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 использовать 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 20 — там эти списки объявляют субмодули, которые могут быть загружены инструкцией `from *`.)

Включение будущих возможностей языка

В языке периодически появляются изменения, которые могут повлиять на работоспособность существующего программного кода. Сначала они появляются в виде расширений, которые по умолчанию отключены. Чтобы включить такие расширения, используется инструкция импорта специального вида:

```
from __future__ import имя_функциональной_особенности
```

Эта инструкция вообще должна появляться в самом начале файла модуля (возможно, сразу же вслед за строкой документирования), потому что она включает специальный режим компиляции программного кода для каждого отдельно взятого модуля. Возможно также выполнить эту инструкцию в интерактивной оболочке, что позволит поэкспериментировать с грядущими изменениями в языке; включенная особенность будет после этого доступна в течение всего интерактивного сеанса.

Например, в предыдущем издании этой книги мы использовали эту форму инструкции для демонстрации функций-генераторов, в которых используется ключевое слово, еще недоступное по умолчанию в то время (в качестве *имя_функциональной_особенности* указывалось `generators`). Мы уже использовали эту инструкцию для включения операции истинного деления чисел в главе 5 и будем использовать ее и в этой главе для включения возможности абсолютного импорта, а также в седьмой части книги – для демонстрации менеджеров контекста.

Все эти изменения могут отрицательно сказаться на работоспособности существующего программного кода, 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 он запущен как самостоятельная программа или импортирован другим модулем. Например, предположим, что мы создаем файл модуля с именем *runme.py*, который экспортирует единственную функцию с именем `tester`:

```
def tester():
    print "It's Christmas in Heaven..."

if __name__ == '__main__':
    tester()                                # Только когда запускается,
                                           # а не импортируется
```

Этот модуль определяет функцию для клиентов и может импортироваться как обычный модуль:

```
% python
>>> import runme
>>> runme.test()
It's Christmas in Heaven...
```

Но в самом конце модуля имеется программный код, который вызывает функцию, когда этот файл запускается как самостоятельная программа:

```
% python runme.py
It's Christmas in Heaven...
```

Пожалуй, чаще всего проверка атрибута `__name__` выполняется в программном коде для самопроверки модуля. Проще говоря, вы можете добавить в конец модуля 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, запуская его из командной строки или каким-либо другим способом. На практике программный код самопроверки в конце файла, заключенный в условную инструкцию, проверяющую атрибут `__name__`, является, пожалуй, самым распространенным и удобным способом модульного тестирования в языке Python. (В главе 29 обсуждаются другие часто используемые способы тестирования программного кода на языке Python – как будет показано, в стандартной библиотеке существуют модули `unittest` и `doctest`, которые реализуют более совершенные средства тестирования.)

Прием, основанный на проверке атрибута `__name__`, также часто используется при создании файлов, которые могут использоваться и как утилиты командной 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 использоваться в составе других программ.

Тестирование модулей с помощью `__name__`

Мы уже видели в этой книге один хороший пример, применительно к которому проверка атрибута `__name__` могла бы быть полезной. В разделе, рассказывающем об аргументах, в главе 16, 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 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 исправить 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)

```

Здесь 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 запускается 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, главный результат – что наш программный код может использоваться и как библиотека инструментов, и как самостоятельная программа.

Изменение пути поиска модулей

В главе 18 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 после этого допускается удалять, добавлять и изменять компоненты списка по своему усмотрению:

```
>>> import sys
>>> sys.path
['', 'D:\\PP3ECD-Partial\\Examples', 'C:\\Python25', ...далее опущено...]

>>> sys.path.append('C:\\sourcedir') # Дополнение пути поиска модулей
>>> import string                  # Новый каталог будет участвовать
                                   # в поиске
```

Как только будут внесены изменения, они будут воздействовать на все последующие инструкции импорта, выполняемые в программе, так как все инструкции 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:\\lp3e\\examples') # Только для этой программы
>>> sys.path
['d:\\temp', 'c:\\lp3e\\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 настройки пути поиска внутри программ на языке Python. Однако будьте внимательны: если убрать из пути критически важный каталог, можно потерять доступ к критически важным утилитам. В предыдущем примере, например, был потерян доступ к модулю `string`, потому что из пути был удален исходный каталог библиотеки.

Кроме того, не забывайте, что такие изменения списка `sys.path` действуют только в рамках интерактивного сеанса или внутри программы (тех-

тически – в рамках процесса), где были выполнены эти изменения, – они не сохраняются после завершения работы интерпретатора. Настройки в переменной окружения 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 машине, и продолжают существовать по завершении программы.

Расширение import as

Обе инструкции, `import` и `from`, были расширены так, чтобы позволить дать модулю в вашем сценарии другое имя. 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 сценарии, которые в противном случае были бы просто перезаписаны инструкцией импортирования. Кроме того, это расширение может пригодиться с целью создания коротких и простых имен для длинных путей, состоящих из цепочки каталогов, при импортировании пакетов, которое описывалось в главе 20.

Синтаксис относительного импорта

В версии Python 2.5 была изменена семантика пути поиска в некоторых инструкциях `from`, когда они применяются к пакетам модулей, которые мы рассматривали в предыдущей главе. Некоторые аспекты этого изменения не будут очевидны до выхода более поздних версий Python (в настоящее время их включение запланировано для версий 2.7 и 3.0), однако некоторые из них доступны уже сегодня.

Говоря в двух словах, инструкции `from` теперь могут использовать точки («.»), чтобы указать, что поиск модулей в первую очередь должен производиться в том же самом пакете (эта особенность известна также как *относительный импорт внутри пакета*), и только потом может

выполняться где-то в другом месте, в пути поиска импортируемых модулей (эта особенность называется *абсолютным импортом*). То есть:

- На сегодняшний день можно использовать точки, чтобы указать, что импорт должен производиться *относительно* вмещающего пакета – при таком способе импортирования предпочтение будет отдаваться модулям, расположенным внутри пакета, а не одноименным модулям, находящимся где-то в пути поиска, в `sys.path`.
- Обычная операция импортирования в программном коде пакета (без точек) в настоящее время по умолчанию выполняется в порядке «сначала поиск относительно пакета, потом – абсолютный поиск». Однако в будущем интерпретатор будет по умолчанию использовать абсолютный импорт – при отсутствии точек операции импортирования будут пропускать вмещающий пакет и искать импортируемые модули в пути поиска `sys.path`.

Например, в настоящее время инструкция:

```
from .spam import name
```

означает: «из модуля с именем `spam`, расположенного в том же пакете, что и сама инструкция, импортировать переменную `name`». Похожая инструкция без начальной точки по умолчанию также будет использовать порядок поиска «сначала относительно пакета, а затем – абсолютный поиск», если только в импортирующий файл не будет включена следующая инструкция:

```
from __future__ import absolute_import    # До выхода версии 2.7?
```

Если эта инструкция присутствует, она включает использование абсолютного пути поиска, которое станет поведением по умолчанию в будущем. В результате этого все операции импортирования без дополнительных точек будут пропускать компоненты в том же пакете и производить поиск по абсолютному пути, хранящемуся в списке `sys.path`. Например, когда абсолютный импорт включен таким способом, следующая инструкция всегда будет находить не модуль `string` в текущем пакете, а одноименный модуль в стандартной библиотеке:

```
import string    # Всегда будет находить версию модуля в стандартной библиотеке
```

Без инструкции `from __future__` всегда будет импортироваться модуль `string`, если он присутствует в пакете. Чтобы получить то же самое поведение в будущем, когда по умолчанию будет выполняться абсолютный импорт, для выполнения относительного импорта можно будет использовать следующую форму инструкции (которая работает и в настоящее время):

```
from . import string    # Сначала поиск производится внутри пакета
```

Обратите внимание: ведущий символ точки может использоваться только в инструкции `from`, в инструкции `import` он недопустим. Инструкция `import modname` в настоящее время по-прежнему выполняет от-

носительный импорт, но в Python 2.7 она будет выполнять абсолютный импорт.

Возможны также и другие варианты точечной нотации для ссылки на модули в пакете. Допустим, что имеется пакет `mypkg`, тогда следующие альтернативные 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 разобраться с обоснованием этого грядущего изменения.

Зачем необходим относительный импорт

Эта возможность предназначена, чтобы дать сценариям возможность ликвидировать возникающие неоднозначности, которые могут возникать, когда в разных местах в пути поиска присутствует несколько одноименных модулей. Рассмотрим 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 2.4 и в более ранних версиях интерпретатор будет сначала искать модуль в каталоге *mypkg*, выполняя *относительный* импорт. 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 импортировать модуль `string` из стандартной библиотеки языка Python. К сожалению, в этих версиях Python нет достаточно простого способа проигнорировать модуль `mypkg.string` и импортировать модуль `string` из стандартной библиотеки, расположенной в пути поиска модулей правее. Мы не можем зависеть от структуры каталогов пакета, описанных выше стандартной библиотеки, присутствующей на любом компьютере.

Другими словами, инструкции импортирования в пакетах могут быть неоднозначными – внутри пакета может быть непонятно, какой модуль пытается импортировать инструкция `import spam`, – внутри пакета или за его пределами. Если говорить более точно, локальный модуль или пакет могут сделать невозможным 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 требует-

ся 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 новые модули – с теми же именами, которые присвоены вашим уже существующим модулям. Программный код, использующий особенности относительного импорта, сложнее понять, потому что бывает трудно выяснить, какой модуль импортируется. Гораздо лучше, если решение явно описывается в программном коде.

В Python 2.5 мы можем управлять поведением операций импорта, вынуждая их использовать абсолютный режим импортирования инструкцией `from __future__`, которая была описана ранее. Учтите, что режим абсолютного импортирования будет использоваться по умолчанию в будущих версиях (планируется ввести его в Python 2.7). Когда режим абсолютного импортирования будет включен, следующая инструкция в нашем примере файла `mypkg\main.py` всегда будет находить версию модуля `string` из стандартной библиотеки благодаря абсолютному поиску:

```
import string      # Импорт модуля string из стандартной библиотеки
```

Вы должны уже сейчас привыкать к использованию абсолютного режима импорта, чтобы быть готовыми, когда изменения вступят в силу. Таким образом, если вы действительно хотите импортировать модуль из своего пакета, то, чтобы сделать это явным и абсолютным, вы должны начать писать инструкции, подобные этой (`mypkg` будет найден в режиме абсолютного поиска по `sys.path`):

```
from mypkg import string      # Импортирует mypkg.string (по абсолютному пути)
```

Режим относительного импорта по-прежнему остается доступным – при использовании точки в инструкции `from`:

```
from . import string      # Импортирует mypkg.string (по относительному пути)
```

Данная форма инструкции импортирует модуль `string` относительно текущего пакета и является примерным эквивалентом абсолютной формы из предыдущего примера (поиск в каталоге пакета автоматически выполняется в первую очередь).

С помощью относительного синтаксиса можно также копировать заданные имена из модуля:

```
from .string import name1, name2      # Импортирует имена из mypkg.string
```

Эта инструкция также ссылается на модуль `string` относительно текущего пакета. Если поместить эту инструкцию в модуль `mypkg.main`, например, она будет импортировать имена `name1` и `name2` из модуля `mypkg.string`. Дополнительная начальная точка предписывает выполнить относительный 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 иерархии каталогов, что и пакет `mypkg`, – то есть модуль `spam`, находящийся в каталоге, родительском по отношению к пакету `mypkg`. В общем случае действия программного кода в модуле `A.B.C` будут следующими:

```
from . import D      # Импортирует A.B.D
from .. import E     # Импортирует A.E
from ...F import G   # Импортирует A.F.G
```

Синтаксис относительного импорта и предложенные изменения по использованию «абсолютного импорта по умолчанию» – это достаточно сложные концепции, которые лишь частично реализованы в Python 2.4. Поэтому мы больше не будем углубляться в эту тему, и за более подробной информацией обращайтесь к набору стандартных руководств по языку Python.

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 т. д. Все это станет более понятным, когда вы начнете создавать крупные программы на языке Python, а пока ознакомьтесь с несколькими основными положениями:

- **В языке Python вы всегда находитесь в модуле.** Нет никакого способа написать программный код, который не находился бы в каком-нибудь модуле. Фактически даже программный код, который вводится в интерактивной 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 самостоятельные закрытые компоненты. Следуйте правилу: модули должны быть максимально независимы от глобальных имен в других модулях.
- **Максимизируйте согласованность внутри модуля: общая цель.** Уменьшить взаимозависимость модулей можно за счет увеличения согласованности 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 демонстрировали справедливость этого правила на примере программного кода в главе 16. Но будет совсем нелишним повторить его: использование глобальных переменных из других модулей (в конце концов, это один из способов, каким клиенты импортируют

службы) – совершенно нормальное 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 данными через такие механизмы, как аргументы и возвращаемые значения функций, 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. 21.1 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 находятся в модулях, – вы познакомитесь в главе 22.

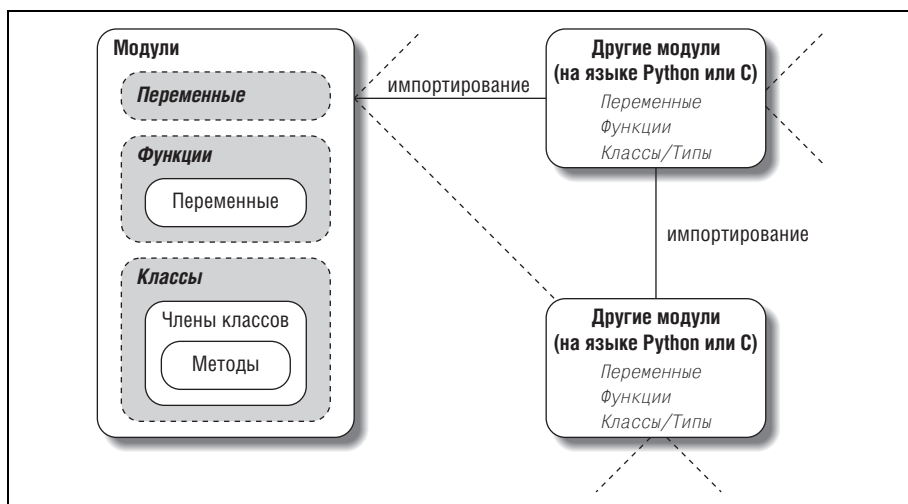


Рис. 21.1. Среда выполнения модуля. Модули импортируются и сами могут импортировать и использовать другие модули, которые могут быть написаны на языке Python или на других языках программирования, таких как C. Модули содержат переменные, функции и классы, с помощью которых решают возложенные на них задачи. 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

Модули – это объекты: метапрограммы

Поскольку модули экспортируют большую часть своих свойств в виде встроенных атрибутов, это позволяет легко создавать программы, которые управляют другими программами. Такие менеджеры программ

мы обычно называем *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 помощью словаря атрибутов модуля (экспортируется в виде встроенного атрибута `__dict__`). Кроме того, интерпретатор экспортирует список всех загруженных модулей в виде словаря `sys.modules` (то есть в виде атрибута `modules` модуля `sys`) и предоставляет встроенную функцию `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 помогает создавать программы, управляющие другими программами.¹ Например, ниже приводится модуль с именем *mydir.py*, в котором использованы эти идеи для реализации измененной версии встроенной функции `dir`. Этот модуль определяет и экспортирует функцию с именем `listing`, которая принимает объект модуля в качестве аргумента и выводит отформатированный листинг пространства имен модуля:

```
# Модуль, который выводит содержимое пространства имен других модулей

verbose = 1

def listing(module):
    if verbose:
        print "-"*30
        print "name:", module.__name__, "file:", module.__file__
        print "-"*30

    count = 0
```

¹ Как мы видели в главе 16, 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 символов!) так: `import sys; glob=sys.modules[__name__]; glob.X=0`. Не забывайте, что каждый 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.

```

for attr in module.__dict__.keys():      # Сканировать пространство имен
    print "%02d) %s" % (count, attr),
    if attr[0:2] == "__":
        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 "-"*30
    print module.__name__, "has %d names" % count
    print "-"*30

if __name__ == "__main__":
    import mydir
    listing(mydir)                        # Код самопроверки: вывести свое
                                         # пространство имен

```

В модуле, в самом конце, реализована логика самопроверки, которая заставляет модуль импортировать самого себя и вывести содержимое своего пространства имен. Ниже показан результат работы этого модуля:

```

C:\python> python mydir.py
-----
name: mydir file: mydir.py
-----
00) __file__ <built-in name>
01) __name__ <built-in name>
02) listing <function listing at 885450>
03) __doc__ <built-in name>
04) __builtins__ <built-in name>
05) verbose 1
-----
mydir has 6 names
-----

```

С функцией `getattr` и родственными ей мы встретимся еще раз позднее. Самое важное здесь, что `mydir` — это программа, которая позволяет исследовать другие программы. Так как интерпретатор не скрывает внутреннее устройство модулей, вы можете реализовать обработку любых объектов единообразно.¹

¹ Инструменты, такие как `mydir.listing`, могут быть предварительно загружены в пространство имен интерактивной оболочки импортированием их в файле, указанном в переменной окружения `PYTHONSTARTUP`. Так как программный код этого файла выполняется в интерактивном пространстве имен (модуль `__main__`), такой способ импортирования часто используемых инструментов позволит вам сэкономить время на вводе инструкций вручную. Дополнительная информация приводится в приложении А.

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.

Порядок следования инструкций на верхнем уровне имеет значение

Когда модуль впервые импортируется (или загружается повторно), интерпретатор выполняет инструкции в нем одну за другой, сверху вниз. Из этого следует несколько замечаний, которые касаются опережающих ссылок на переменные, которые следует подчеркнуть особо:

- Инструкции программного кода на верхнем уровне в файле модуля (не вложенные в функцию) выполняются, как только интерпретатор достигает их в процессе импортирования. По этой причине он не может сослаться 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 программном коде верхнего уровня, который выполняется немедленно; функции могут ссылаться на любые имена. Ниже приводится пример, демонстрирующий опережающие ссылки:

```
func1()                # Ошибка: имя "func1" еще не существует

def func1():
    print func2()       # ОК: поиск имени "func2" будет выполнен позднее

func1()                # Ошибка: имя "func2" еще не существует

def func2():
    return "Hello"

func1()                # ОК: "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 успешно, 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`, возьмите за правило помещать инструкции `def` в начало файла, а программный код верхнего уровня – в конец файла. При таком подходе ваши функции гарантированно будут определены к моменту выполнения программного кода, который их использует.

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 сожалению, невозможно напрямую использовать инструкции импорта для загрузки модуля, имя которого задано в виде строки, – в этих инструкциях интерпретатор 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 строку переменной:

```
x = "string"
import x
```

Чтобы решить эту проблему, необходим специальный инструмент, выполняющий 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 инструкции `exec` для исполнения:

```
>>> modname = "string"
>>> exec "import " + modname      # Выполняется как строка программного кода
>>> string                        # Модуль был импортирован в пространство имен
<module 'string'>
```

Инструкция `exec` (и родственная ей функция `eval`, используемая для вычисления значений выражений) скомпилирует строку в код и передаст его интерпретатору для исполнения. В языке Python компилятор байт-кода доступен непосредственно во время выполнения, поэтому можно писать программы, которые конструируют и выполняют другие программы, как в этом случае. По умолчанию инструкция `exec` выполняет программный код в текущей области видимости, но существует возможность передавать ей необязательные словари пространств имен.

Единственный настоящий недостаток инструкции `exec` состоит в том, что она должна компилировать инструкцию `import` всякий раз, когда она запускается, – если импортировать приходится достаточно часто, программный код может работать немного быстрее при использовании встроенной функции `__import__`, 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'>
```

Инструкция `from` создает копии, а не ссылки

Несмотря на то что инструкция `from` широко применяется, она часто становится источником самых разных проблем. При выполнении присваивания именам в области видимости импортирующего модуля она не создает 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 следующий модуль (*nested1.py*):

```
X = 99
def printer(): print X
```

Если импортировать эти два имени с помощью инструкции `from` в другом модуле (*nested2.py*), будут получены копии этих имен, а не ссылки на них. Изменение имени в импортирующем модуле приведет к изменению только локальной версии этого имени, а имя в модуле *nested1.py* будет иметь прежнее значение:

```
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 изменению имени в файле *nested1.py*. Полное имя направляет интерпретатор к имени в указанном объекте модуля, а не к имени в импортирующем модуле (*nested3.py*):

```
import nested1                  # Импортировать модуль целиком
nested1.X = 88                  # ОК: изменяется имя X в nested1
nested1.printer()

% python nested3.py
88
```


Инструкция `from *` может затушевывать смысл переменных

Я упоминал об этом в главе 19, но оставил подробности до этого момента. Поскольку в инструкции `from module import *` не указываются необходимые имена переменных, она может непреднамеренно перезаписать имена, уже используемые в области видимости импортирующего модуля. Хуже того, это может осложнить 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 которых может находиться в отдельном каталоге):

```
>>> from module1 import *   # Плохо: может незаметно перезаписать мои имена
>>> from module2 import *   # Еще хуже: нет никакого способа понять,
>>> 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 применение формы `from *` одним модулем на файл. В этом случае любые неопределенные имена, согласно дедуктивному методу, должны находиться в модуле, который 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.

Функция `reload` может не оказывать влияния, если импорт осуществлялся инструкцией `from`

Это еще одна ловушка, связанная с инструкцией `from`: как уже говорилось ранее, инструкция `from` копирует (присваивает) имена при выполнении, поэтому нет никакой обратной связи с модулем, откуда были скопированы имена. Имена, скопированные инструкцией `from`, просто становятся ссылками на объекты, на которые ссылались по тем же именам в импортируемом модуле, когда была выполнена инструкция `from`.

Вследствие этого повторная загрузка импортируемого модуля может не оказывать воздействия на клиентов, которые импортировали его имена с помощью инструкции `from`. То есть имена в модулях-клиентах будут по-прежнему ссылаться 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!
...

```

```
reload(module)      # Изменится модуль, но не мои имена
X                   # По-прежнему ссылается на старый объект
```

Чтобы сделать повторную загрузку более эффективной, используйте инструкцию `import` и полные имена переменных вместо инструкции `from`. Поскольку полные имена всегда ведут обратно в импортированный модуль, после повторной загрузки они автоматически будут связаны с новыми именами в модуле.

```
import module      # Получить объект модуля, а не имена
...
reload(module)     # Изменит непосредственно сам объект модуля
module.X           # Текущее значение X: отражает результат перезагрузки
```

reload, from и тестирование в интерактивной оболочке

В главе 3 уже говорилось, что из-за возникающих сложностей лучше не использовать операции импорта и повторной загрузки для запуска программ. Дело еще больше усложняется, когда в игру вступает инструкция `from`. Начиная осваивать язык Python часто сталкиваются с проблемой, описываемой 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 возвращаетесь в окно редактирования, исправляете ее и пытаетесь повторно загрузить модуль следующим способом:

```
reload(module)
```

Но вы не получите ожидаемого эффекта – инструкция `from` создала имя `function`, но не создала имя `module`. Чтобы сослаться на модуль в функции `reload`, его сначала необходимо импортировать инструкцией `import`:

```
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, такие 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, после перезагрузки модуля ее необходимо вызывать как `module.function` или повторно запустить инструкцию `from`:

```
import module
reload(module)
from module import function
function(1, 2, 3)
```

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 запускать инструкции `from` после перезагрузки модуля. Это достаточно сложно, чтобы время от времени сбивать с толку даже опытного программиста.

Вы не должны ожидать, что функция `reload` и инструкция `from` будут безукоризненно 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→Run Module (Запустить→Запустить модуль) в меню среды разработки IDLE щелчком мыши на ярлыке файла или из командной строки системы).

Функция `reload` не обладает транзитивностью

Когда выполняется повторная загрузка модуля, интерпретатор перезагружает только данный конкретный файл модуля – он не выполняет повторную загрузку модулей, которые были импортированы перезагружаемым модулем. Например, если выполняется перезагрузка некоторого модуля *A*, и *A* импортирует модули *B* и *C*, перезагружен будет только модуль *A*, но не *B* и *C*. Инструкции внутри модуля *A*, которые импортируют модули *B* и *C*, будут перезапущены в процессе перезагрузки, но они просто вернут объекты уже загруженных модулей *B* и *C* (предполагается, что к этому моменту они уже были импортированы). 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
>>> . . .
>>> reload(A)
```

Не следует полагаться на транзитивную перезагрузку модулей – лучше несколько раз вызывать функцию `reload` для непосредственного обновления субкомпонентов. При желании можно предусмотреть в программе автоматическую перезагрузку ее компонентов, добавив вызовы `reload` в родительские модули, каким здесь является *A*.

Еще лучше было бы написать универсальный инструмент для автоматического выполнения рекурсивной перезагрузки модулей, сканируя содержимое атрибутов `__dict__` модулей (смотрите раздел «Модули – это объекты: метапрограммы», выше в этой главе) и проверяя атрибут `type` в каждом элементе (глава 9), чтобы отыскать вложенные модули. Такие вспомогательные функции могут рекурсивно вызывать сами себя и обходить произвольные цепочки импортирования.

Например, модуль *reloadall.py*, в листинге ниже, содержит функцию *reload_all*, автоматически выполняющую перезагрузку модуля, каждого импортируемого им модуля, и т. д., до самого конца каждой цепочки импортирования. Она использует словарь, с помощью которого отыскивает уже загруженные модули, рекурсию – для обхода цепочек импорта и модуль *types* из стандартной библиотеки (был представлен в конце главы 9), в котором просто предопределены значения атрибута *type* для всех встроенных типов.

Чтобы воспользоваться этой утилитой, импортируйте функцию *reload_all* и передайте ей имя уже загруженного модуля (как если бы это была встроенная функция *reload*). Когда файл запускается как самостоятельная программа, его код самопроверки выполнит проверку самого модуля – он должен импортировать самого себя, потому что его собственное имя не будет определено в файле без инструкции импортирования. Я рекомендую поэкспериментировать с этим примером самостоятельно:

```
import types

def status(module):
    print 'reloading', module.__name__

def transitive_reload(module, visited):
    if not visited.has_key(module):      # 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
```

Рекурсивный импорт с инструкцией *from* может не работать

Напоследок я оставил самую странную (и, к счастью, малоизвестную) проблему. 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 импортом*). 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 инструкция `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` создает имя `X` и затем импортирует `recur2` до того, как присвоит значение имени `Y`. В этом месте модуль `recur2` может импортировать модуль `recur1` целиком с помощью 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 инструкции `import` в `recur1`, еще не существует, поэтому возникнет ошибка:

```
# Файл: recur1.py
X = 1
import recur2          # Запустить recur2, если он еще не был импортирован
Y = 2

# Файл: recur2.py
from recur1 import X    # OK: "X" уже имеет значение
from recur1 import Y    # Ошибка: "Y" еще не существует

>>> 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 будет 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 заикнется, если вы все-таки сделаете это, но ваша программа попадет в зависимость от порядка следования инструкций в модулях.

Существует два способа решения этой проблемы:

- Обычно можно ликвидировать рекурсивный импорт, подобный приведенному, правильно проектируя модули: увеличить согласованность внутри модуля и уменьшить взаимозависимость между модулями – это самое первое, что стоит попробовать сделать.

- Если от циклов не удастся избавиться полностью, попробуйте отсрочить обращение к именам модуля, используя инструкции `import` и полные имена (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 модулями. Мы изучили приемы сокрытия данных, включение новых особенностей языка из модуля `__future__`, возможности использования переменной `__name__`, синтаксис относительного импортирования в пакетах и многое другое. Мы также исследовали проблемы проектирования модулей и познакомились с типичными ошибками при работе с модулями, 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, рассмотренных в последних нескольких главах, применима и здесь – классы располагаются в модулях и также являются пространствами имен, но они добавляют еще один элемент в поиск атрибутов, который называется «поиск в цепочке наследования». Поскольку это последняя глава в этой части книги, прежде чем углубиться в объектно-ориентированное программирование, обязательно выполните упражнения к этой части книги. Но перед этим попробуйте ответить на контрольные вопросы главы, чтобы освежить в памяти темы, рассматривавшиеся здесь.

Закрепление пройденного

Контрольные вопросы

1. Что важно знать о переменных в программном коде верхнего уровня модуля, имена которых начинаются с одного символа подчеркивания?
2. Что означает, когда переменная `__name__` модуля имеет значение `"__main__"`?
3. В чем разница между инструкциями `from mypkg import spam` и `from . import spam`?
4. Если пользователь вводит имя модуля в ответ на запрос программы, как импортировать этот модуль?
5. Чем отличается изменение списка `sys.path` от изменения значения переменной окружения `PYTHONPATH`?
6. Импорт будущих изменений в языке возможен с помощью модуля `__future__`, а возможен ли импорт из прошлого?

Ответы

1. Переменные в программном коде верхнего уровня модуля, чьи имена начинаются с одного символа подчеркивания, не копируются при импортировании 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 импортирован в качестве модуля другим файлом в программе. То есть файл используется как программа, а не как библиотека.
3. Инструкция `from mypkg import spam` выполняет импортирование в абсолютном режиме, используя путь поиска в `sys.path`. Инструкция `from . import spam`, напротив, выполняет импорт в относительном режиме – поиск модуля `spam` осуществляется сначала в пакете, где находится инструкция, и только потом в `sys.path`.
4. Как правило, ввод пользователя поступает в сценарий в виде строки – чтобы импортировать модуль по имени, заданному в виде строки, можно собрать и выполнить инструкцию `import` с помощью инструкции `exec` или передать строку с именем функции `__import__`.
5. Изменения в `sys.path` воздействуют только на работающую программу и носят временный характер – изменения будут утеряны сразу же после завершения программы. Содержимое переменной окружения `PYTHONPATH` хранится в операционной системе – оно оказывает воздействие на все программы, выполняемые на этом компьютере, а изменения сохраняются после завершения программ.
6. 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 А.

1. *Основы импортирования.* Напишите программу, которая подсчитывает количество строк и символов в файле (в духе утилиты `wc` в операционной системе UNIX). В своем текстовом редакторе создайте модуль с именем `mytod.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 функции `len`).

¹ Последняя стабильная версия. – Примеч. науч. ред.

- Функцию `countChars(name)`, которая читает входной файл и подсчитывает число символов в нем (подсказка: метод `file.read` возвращает единую строку).
- Функцию `test(name)`, которая вызывает две предыдущие функции с заданным именем файла. Вообще говоря, имя файла можно жестко определить в программном коде, принимать ввод имени от пользователя или принимать его 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 строки. Если размер любой из функций превысит две-три строки, это значит, что вы делаете лишнюю работу, – используйте подсказки, которые я вам дал!

Затем проверьте свой модуль в интерактивной оболочке, используя инструкцию `import` и полные имена экспортируемых функций. Следует ли добавить 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 честолобивы, 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 оболочке, 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 производился 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 импортируйте модуль и проверьте работу функций в интерактивном режиме. Будут ли работать функции в обоих режимах?
4. *Вложенное импортирование*. Напишите второй модуль `myclient.py`, который импортирует модуль `mymod` и проверяет работу его функций, затем запустите `myclient` из системной командной строки. Будут ли доступны функции 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 интерактивном 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 модулей. 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 платформах Python (это одна из причин, почему в языке Python в качестве разделителя 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 программный код в `__init__.py` и посмотрите, будет ли он выполняться при каждой операции импортирования.

6. *Повторная загрузка.* Поэкспериментируйте с возможностью повторной загрузки модуля: выполните тесты, которые приводятся в примере *changer.py* в главе 19, 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 редактировать в другом окне или, приостановив интерпретатор, редактировать модуль в том же окне (в UNIX комбинация клавиш Ctrl-Z обычно приводит к приостановке текущего процесса, а команда fg – возобновляет его работу).
7. *Циклический импорт.*¹ В разделе, где описываются проблемы, связанные с рекурсивным импортом, импортирование модуля `recur1` вызывает появление ошибки. Но если перезапустить интерактивный сеанс работы с интерпретатором и предварительно импортировать модуль `recur2`, ошибка не возникает – проверьте этот факт сами. Как вы думаете, почему импортирование `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, «завершен» модуль или нет.) Теперь попробуйте запустить `recur1` как самостоятельный 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 программы, они не импортируются, поэтому здесь возникает тот же эффект, как и при импортировании `recur2` в интерактивной оболочке, – `recur2` является первым импортируемым модулем.) Что произойдет, если запустить `recur2` как самостоятельный сценарий? Почему?

¹ Обратите внимание, что циклическое импортирование чрезвычайно редко встречается на практике. Автор этой книги за десять лет работы с языком Python на практике сам не использовал и не сталкивался с циклическим импортом. 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.

VI

Классы и ООП

22

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 передавали объекты, использовали их в выражениях, вызывали их методы и т. д. Однако, чтобы наш программный код стал по-настоящему объектно-ориентированным (*object-oriented*, ОО), наши объекты должны участвовать в *иерархии наследования*.

Начиная с этой главы, мы приступаем к исследованию *классов* в языке Python – программных компонентов, используемых для реализации новых типов объектов, поддерживающих наследование. Классы – это основные инструменты объектно-ориентированного программирования (ООП) в языке Python, поэтому 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 снизить его избыточность, и пишем новые программы, *адаптируя* имеющийся программный код, а не изменяя существующий или создавая новый код.

Классы в языке Python создаются с помощью новой для нас инструкции: инструкции `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 манипулировать новыми объектами, а кроме того, они поддерживают механизм наследования – совсем иной способ адаптации программного кода и повторного его использования, чем мы рассматривали ранее.

Одно предварительное замечание: ООП в языке Python является не обязательным к применению, и на первых порах вам не обязательно использовать классы. Многие задачи можно реализовать с помощью более простых конструкций, таких как функции, или даже просто программируя на верхнем уровне в сценарии. Поскольку использование классов требует некоторого предварительного планирования, они представляют больший интерес для тех, кто работает в *стратегическом* режиме (участвует в долгосрочной разработке программных продуктов), чем для тех, кто работает в *тактическом* режиме (где на разработку 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 классы 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, знание основ работы с классами будет как минимум полезно.

Зачем нужны классы?

Помните, как я говорил вам, что «программы выполняют некоторые действия над чем-то»? Проще говоря, классы – это всего лишь способ определить новое «что-то», они являются отражением реальных объектов в мире программ. Например, предположим, что мы решили реализовать гипотетическую машину по изготовлению пиццы, которую мы использовали в качестве примера в главе 15. 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 машин. 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, которые все вместе работают как единое целое. Например, нашей машине необходимы манипуляторы, чтобы раскатывать тесто, двигатели, чтобы перемещаться к духовке, и т. д. На языке ООП наша машина – это пример композиции; она содержит другие объекты, которые активизируются для выполнения определенных дейст-

вий. 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 создается как набор визуальных элементов управления – кнопок, меток и т. д., 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 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 пространство имен для своих собственных данных, отличных от данных других объектов.

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 операций, которые мы знаем по работе со встроенными типами. Например, объекты, созданные из классов, могут подвергаться операции извлечения среза, конкатенации, извлечения элементов по их индексам и т. д. Язык Python предоставляет специальные обработчики, которые могут использоваться 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 показаться немного сложными. Кроме того, смысл этих терминов может ускользать от вас, пока вы не получите возможность познакомиться с тем, как программисты 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 использовании, чем в других языках программирования, таких как C++ и Java. Будучи языком сценариев с динамической типизацией, Python ликвидирует синтаксическую 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 т. д. Однако, когда подобное выражение применяется к объекту, полученному с помощью инструкции `class`, интерпретатор начинает *поиск* – поиск в дереве связанных объектов, который заканчивается, как только будет встречено первое появление 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, а потом все классы выше него, снизу вверх и слева направо.

Другими словами, выборка атрибутов производится в результате простого поиска 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 происходит 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 действительно поднимается вверх по дереву, пытаясь во время выполнения отыскать атрибуты всякий раз, когда мы используем выражение `object.attribute`. Чтобы было более понятно, на рис. 22.1 приводится пример одного из таких деревьев.

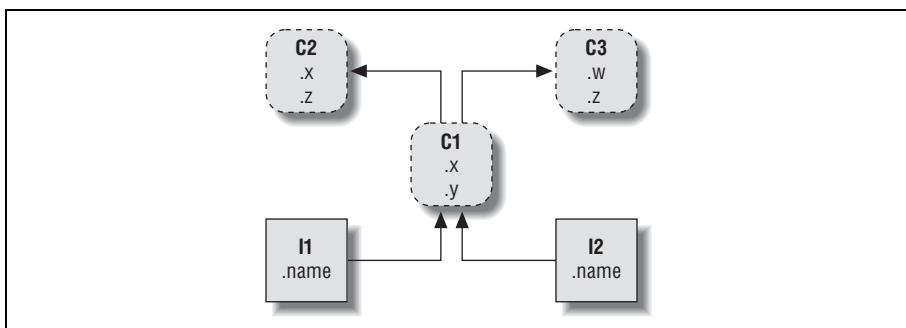


Рис. 22.1. Дерево классов с двумя экземплярами внизу (I1 и I2), классом выше (C1) и двумя суперклассами в самом верху (C2 и C3). Все эти объекты являются пространствами имен (пакетами переменных), а наследование означает просто поиск в дереве снизу вверх, цель которого – найти самое нижнее вхождение 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 говорить более точно, это дерево связывает вместе три объекта классов (в овалах, C1, C2 и C3) и два объекта экземпляра (в прямоугольниках, I1 и I2) в иерархию наследования. Обратите внимание, что в модели объектов в языке Python классы и экземпляры порождаются от двух разных типов объектов:

Классы

Играют роль фабрик экземпляров. Их атрибуты – данные и функции – обеспечивают поведение, то есть наследуются всеми экземплярами, созданными от них (например, функция, вычисляющая зарплату служащего, исходя из часового тарифа).

Экземпляры

Представляют конкретные элементы программы. Их атрибуты хранят данные, которые могут отличаться в конкретных объектах (например, номер 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. 22.1 можно продолжить категоризацию классов по относительным положениям 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

¹ В других книгах можно также встретить такие термины, как *базовые классы* и *дочерние классы*, которые используются для обозначения суперклассов и подклассов соответственно.

термины обозначают относительное положение в дереве и исполняемые роли. Суперклассы реализуют поведение, общее для всех их подклассов, но из-за того, что поиск производится снизу вверх, подклассы могут переопределять поведение, определяемое их суперклассами, переопределяя имена суперклассов ниже в дереве.

Так как эти последние несколько слов отражают основную суть адаптации программного обеспечения в ООП, давайте подробнее рассмотрим эту концепцию. Предположим, что мы создали дерево, приведенное на рис. 22.1, и затем пишем:

```
I2.w
```

Этот программный код демонстрирует использование механизма наследования. Так как это выражение вида `object.attribute`, оно приводит к запуску поиска в дереве с рис. 22.1 – интерпретатор приступает к поиску 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 атрибут `w` имеется только в этом объекте. Другими словами, имя `I2.w` в терминах автоматического поиска ищется 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 путях в дереве. Например:

- 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`.
- Для `I2.name` атрибут `name` будет найден в `I2`, в этом случае поиск по дереву вообще осуществляться не будет.

Проследите эти пути поиска в дереве на рис. 22.1, чтобы понять, как выполняется 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 обеспечения 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 экземпляры, которые мы объединили в иерархические деревья, выполняют практически одну и ту же роль: каждый из этих типов служит некоторым представлением *пространства имен* – пакета переменных и места, где определяются 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 классы представляют собой своего рода *фабрики* по производству экземпляров. Например, в реалистичном приложении у нас мог бы быть класс `Employee`, которые определяет характеристики служащего, – из этого класса мы можем создавать фактические экземпляры класса `Employee`. Это еще одно различие между классами 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, присоединенные к ним (например, `computeSalary`), а у экземпляров – элементы данных, используемые 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, которая обеспечивает более широкие возможности адаптации программного обеспечения, 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 функции, присоединенные к классам в виде атрибутов).

Если ссылка `I2.w` – это вызов функции, тогда в действительности это выражение означает: «вызвать функцию `C3.w` для обработки `I2`». То есть интерпретатор Python автоматически отобразит вызов `I2.w()` на вызов `C3.w()`, передав унаследованной функции экземпляр в виде первого аргумента.

Фактически всякий раз, когда вызывается функция, присоединенная к классу, подразумевается не класс целиком, а экземпляр класса. Этот подразумеваемый 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 служащего, которому дается надбавка к зарплате.

Как мы увидим позднее, Python передает методам подразумеваемый экземпляр в виде специального первого аргумента, в соответствии с соглашением именуемого `self`. Мы также узнаем, что методы могут вызываться как через экземпляры (например, `bob.giveRaise()`), так и через классы (например, `Employee.giveRaise(bob)`), причем обе формы играют одну и ту же роль в наших сценариях. Чтобы увидеть, как методы принимают свои подразумеваемые экземпляры, нам необходимо рассмотреть примеры программного кода.

Создание деревьев классов

Несмотря на всю отвлеченность наших разговоров, тем не менее за кулисами всех этих идей стоит реальный программный код. Мы создаем деревья и объекты в них с помощью инструкций `class` и вызовов классов, которые позднее мы рассмотрим 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 определяет порядок расположения в дереве.

Чтобы создать дерево, изображенное на рис. 22.1, например, мы могли бы использовать 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 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 более одного суперкласса над собой в дереве классов.

В языке Python, если в инструкции `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 имеет большое значение, к какому из объектов присоединяется тот или иной атрибут, – тем самым определяется его область видимости. Атрибуты, присоединяемые к экземплярам, принадлежат только этим конкретным экземплярам, но атрибуты, присоединяемые к классам, совместно используются всеми подклассами и экземплярами. Позднее мы подробно изучим программный код, выполняющий присоединение атрибутов к этим объектам. Мы увидим, что:

- атрибуты обычно присоединяются к классам с помощью инструкций присваивания внутри инструкции `class`, а не во вложенных инструкциях `def`, определяющих функции.
- атрибуты обычно присоединяются к экземплярам с помощью присваивания значений специальному аргументу с именем `self`, передаваемому функциям внутри классов.

Например, классы определяют поведение своих экземпляров с помощью функций, 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 унаследованы 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 функциональной 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 экземпляров, их методы обычно используют этот, получаемый

¹ Если когда-нибудь вам приходилось использовать язык C++ или Java, вы без труда 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 атрибутам более очевидными.

автоматически, аргумент `self` для получения или изменения значений атрибутов конкретного 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 объявляются заранее, а появляются, когда им впервые выполняется присваивание значений. 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` автоматически 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 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 ошибки, сообщающей о неопределенном имени. Если в классе требуется гарантировать, что атрибут, такой как `name`, всегда будет присутствовать в экземплярах, то такой 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 экземпляр будет передаваться методу `__init__` в виде первого аргумента `self`, а любые значения, перечисленные в круглых скобках при вызове класса, будут передаваться во втором и последующих за ним аргументах. В результате инициализация 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 операторов*. Более подробно эти методы будут рассматриваться в последующих главах. Такие методы наследуются в дереве классов как обычно, а их имена начинаются и заканчиваются двумя

символами подчеркивания, чтобы подчеркнуть их особенное назначение. Интерпретатор Python вызывает их автоматически, когда экземпляры, поддерживающие их, участвуют в соответствующих операциях, и они главным образом являются альтернативой вызовам простых методов. Кроме того, они являются необязательными: при их отсутствии соответствующие операции экземплярами не поддерживаются.

Например, чтобы реализовать пересечение множеств, класс может предусмотреть реализацию метода `intersect` или перегрузить оператор `&`, описав логику его работы в методе с именем `__and__`. Поскольку использование операторов делает экземпляры более похожими на встроенные типы, это позволяет определенным классам обеспечивать непротиворечивый и естественный интерфейс и быть совместимыми с программным кодом, который предполагает выполнение операций над объектами встроенных типов.

ООП – это многократное использование программного кода

Вот, в основном, и все описание ООП в языке Python (за исключением некоторых 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, получение значений атрибутов, вывод и многие другие. Но вообще говоря, ООП реализует поиск атрибутов в деревьях.

Тогда зачем нам погружаться в тонкости создания деревьев объектов и выполнения 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, настраивая написанное программное обеспечение, вместо того чтобы изменять существующий программный код или писать новый код в каждом новом проекте.

С фундаментальной точки зрения, классы – это действительно всего лишь пакеты функций и других имен, которые во многом напоминают модули. Однако автоматический поиск атрибутов в дереве наследования, который мы получаем при использовании классов, обеспечивает возможности по адаптации программного обеспечения более широкие, чем это возможно с помощью модулей и функций. Кроме того, классы представляют собой удобную структуру, обеспечивающую компактное размещение выполняемого кода и переменных, что помогает в отладке.

Например, методы – это обычные функции со специальным первым аргументом, поэтому мы можем подражать некоторым чертам их поведения, вручая передавая объекты для обработки обычным функ-

циям. Однако участие методов в наследовании классов позволяет нам естественным образом адаптировать существующее программное обеспечение, создавая новые подклассы, определяющие новые методы, вместо того чтобы изменять существующий программный код. Подобное невозможно в случае с модулями и функциями.

В качестве примера предположим, что вас привлекли к реализации приложения базы данных, где хранится информация о служащих. Как программист, использующий объектно-ориентированные особенности языка Python, вы могли бы начать работу с реализации общего суперкласса, который определяет поведение, 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 категории 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 поведение будет унаследовано от общего класса. Например, если зарплата инженеров 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 будет замещать (переопределять) общую версию метода в классе `Employee`. Затем можно создать экземпляры разновидностей классов служащих в соответствии с принадлежностью имеющихся служащих классам, чтобы обеспечить корректное поведение:

```
bob = Employee()    # Поведение по умолчанию
mel = Engineer()    # Особые правила начисления зарплаты
```

Обратите внимание, что существует возможность создавать экземпляры любых классов в дереве, а не только тех, что находятся внизу, — класс, экземпляр которого создается, определяет уровень, откуда будет начинаться поиск атрибутов. 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 композиции, 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 представлена в главе 4 и повторно рассматривалась в главе 15. Вспомните, полиморфизм означает, что смысл операции зависит от объекта, над которым она выполняется. 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 инкапсуляции) различий интерфейсов. Например, программа, которая обрабатывает потоки данных, может работать с объектами, имеющими методы ввода и вывода, 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 можем использовать одну и ту же функцию `processor` для работы с любыми источниками данных, как уже существующими, так и с теми, что появятся в будущем:

```
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):
    def read(self): ...          # Чтение из сокета
...
processor(FileReader(...), Converter, FileWriter(...))
processor(SocketReader(...), Converter, TapeWriter(...))
processor(FtpReader(...), Converter, XmlWriter(...))
```

¹ Обратите внимание, что список `company` из этого примера мог бы сохраняться в файле, чтобы обеспечить постоянное хранение базы данных с информацией о служащих (с помощью модуля `pickle`, представленного в главе 9 в разделе «Использование файлов»). Кроме того, в состав Python входит модуль `shelve`, который мог бы позволить сохранять экземпляры классов в файлах с доступом по ключу, – сторонняя разработка, система ZODB, позволяет реализовать то же самое, но имеет более качественную поддержку объектно-ориентированных баз данных для промышленного использования.

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 логику 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 программного обеспечения), то обнаружите, начиная писать новую программу, что большая часть вашей задачи уже реализована, и ваша задача в основном сводится к тому, чтобы подобрать уже имеющиеся суперклассы, которые реализуют поведение, требуемое вашей программой. 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 программным кодом «за так».

На практике во многих прикладных областях вы можете получить или купить коллекции суперклассов, известных как *платформы*, в которых реализованы наиболее часто встречающиеся задачи программирования на основе классов, готовые к использованию в ваших приложениях. Такие платформы могут предоставлять интерфейсы к базам данных, протоколы тестирования, средства создания графического интерфейса и т. д. В среде такой платформы вам часто будет достаточно создать свой подкласс, добавив в него один-два своих метода, а основная работа будет выполняться классами платформы, расположенными 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, на то чтобы понять, как использовать классы для достижения такого объектно-ориентированного идеала, потребуется время. На практике ООП влечет за собой большой объем предварительного проектирования, на этапе которого осмысливаются преимущества, получаемые от использования классов; с этой целью программисты начали составлять список наиболее часто встречающихся решений в ООП, известных как *шаблоны проектирования* – помогающих решать проблемы проектирования. При этом объектно-ориентированный программный код на языке настолько прост, что он сам по себе не будет препятствием в освоении ООП. Чтобы убедиться в этом, вам следует перейти к главе 23.

В заключение

В этой главе мы в общих чертах рассмотрели ООП и классы, получив представление о них, прежде чем углубиться в детали синтаксиса. Как мы узнали, основу ООП составляет поиск атрибутов в дереве связанных объектов; мы называем его поиском в дереве наследования. Объекты в нижней части дерева наследуют атрибуты объектов, расположенных в дереве выше, – эта особенность позволяет создавать программы за счет адаптации программного кода, а не за 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, рассмотренной 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 поиску атрибутов в деревьях объектов. Прежде чем двинуться дальше, ответьте на контрольные вопросы, чтобы освежить в памяти все то, о чем рассказывалось 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 составляющие, чтобы снизить его избыточность, и при создании новых программ выполняется адаптация имеющегося программного кода, а не изменение существующего или создание нового кода.
2. Поиск унаследованных атрибутов выполняется сначала в объекте экземпляра, затем в классе, из которого был создан экземпляр, затем во всех суперклассах, в направлении снизу вверх и слева направо.

во в дереве объектов (по умолчанию). Как только будет найдено первое вхождение атрибута, поиск прекращается. Поскольку результатом поиска является самая нижняя версия имени из имеющихся в дереве, иерархия классов естественным образом поддерживает возможность адаптации за счет создания подклассов.

3. И классы, и экземпляры – это пространства имен (пакеты переменных, которые играют роль атрибутов). Главное различие между ними состоит в том, что классы представляют собой своего рода фабрики, позволяющие производить множество экземпляров. Кроме того, классы поддерживают методы перегрузки операторов, которые наследуются экземплярами, а функции, вложенные в классы, интерпретируются как специальные методы обработки экземпляров.
4. Первый аргумент в методах классов играет особую роль, так как через него всегда передается ссылка на объект экземпляра, который подразумевается при вызове метода. Согласно общепринятым соглашениям он обычно называется `self`. Так как по умолчанию методы всегда принимают этот подразумеваемый 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, интерпретатор автоматически будет вызывать его при создании каждого экземпляра этого класса. Этот метод также называют конструктором – он неявно получает новый экземпляр, а также все аргументы, которые были указаны при вызове имени класса. Кроме того, он является типичным представителем методов перегрузки операторов. В отсутствии 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 какого 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 следующей главе).
8. Суперклассы для заданного класса определяются в виде списка в круглых скобках в инструкции `class` после имени нового класса. Порядок следования суперклассов в списке определяет порядок поиска в дереве наследования.

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 модели классов в языке Python.

Если ранее вам не приходилось сталкиваться с ООП, будет сложно сразу усвоить 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 самом простом уровне они представляют собой всего лишь пространства имен, во многом похожие на модули, которые мы изучали в пятой части книги. Но, в отличие 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 особенностей.

Классы генерируют множество экземпляров объектов

Чтобы понять, каким образом обеспечивается возможность создания множества объектов, для начала нужно понять, что в объектно-ориентированной модели 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 экземпляров – это настоящие объекты, обрабатываемые программой, – каждый представляет собой самостоятельное пространство имен, но наследует имена (то есть автомати-

чески имеет доступ к ним) класса, из которого он был создан. Объекты классов создаются инструкциями, а объекты экземпляров – вызовами. Каждый раз, когда вы вызываете класс, вы получаете новый экземпляр этого класса.

Эта объектная концепция существенно отличается от любых других программных конструкций, которые мы видели до сих пор в этой книге. В действительности, классы – это фабрики, способные производить множество экземпляров. В противоположность этому каждый модуль в программе может присутствовать в единственном экземпляре (фактически это одна из причин, почему необходимо использовать функцию `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 приводятся 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* инструкцией. Когда она выполняется, она создает новый объект класса и присваивает его имени, указанному в заголовке инструкции `class`. Кроме того, как и инструкции `def`, инструкции `class` обычно выполняются при первом импортировании содержащих их файлов.
- **Операции присваивания внутри инструкции `class` создают атрибуты класса.** Так же как и в модулях, инструкции присваивания на верхнем уровне внутри инструкции `class` (не вложенные в инструкции `def`) создают атрибуты объекта класса. С технической точки зрения инструкция `class` преобразует свою область видимости в пространство имен атрибутов объекта класса, 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 приводятся несколько отличительных характеристик экземпляров классов:

- **Вызов объекта класса как функции создает новый объект экземпляра.** Всякий раз, когда вызывается класс, создается и возвращается новый объект 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 первоначально пустые, но наследуют атрибуты классов, из которых были созданы.
- **Операции присваивания значений атрибутам через ссылку `self` в методах создают атрибуты в каждом отдельном экземпляре.** Методы класса получают в первом аргументе (с именем `self` в соответствии с соглашениями) ссылки 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 практике. Для начала определим класс с именем `FirstClass`, выполнив инструкцию `class` в интерактивной оболочке интерпретатора Python:

```
>>> class FirstClass:                # Определить объект класса
...     def setdata(self, value):      # Определить методы 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 заголовка, содержащей имя класса, за которой следует тело класса из одной или более вложенных инструкций (обычно) с отступами. Здесь вложенными инструкциями являются инструкции `def` – они определяют функции, реализующие 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 инструкции `def`, и для них верно все то, что мы уже знаем о функциях (они могут иметь аргументы со значениями по умолчанию, возвращать значения и т. д.). Но в первом аргументе методы автоматически получают подразумеваемый 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 настоящему моменту у нас имеется три объекта – два экземпляра и один класс. В действительности у нас имеется три связанных пространства имен, как показано на рис. 23.1. В терминах ООП мы говорим, что `x` «наследует» классу `FirstClass`, как и `y`.

Изначально оба экземпляра пустые, но они связаны с классом, из которого 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 одноименный 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 заключается суть наследования в языке Python: механизм наследования привлекается в момент разрешения имени атрибута, и вся

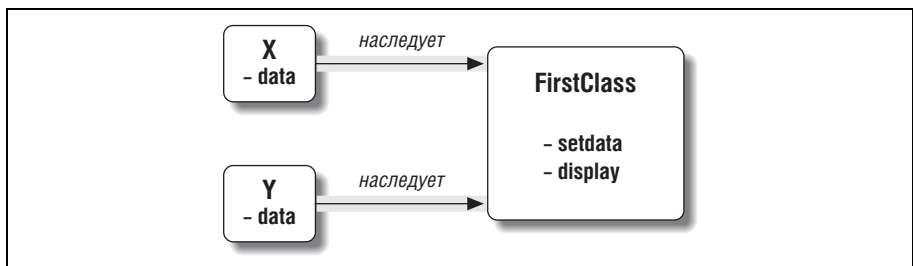


Рис. 23.1. Экземпляры и классы – это связанные между собой объекты пространств имен в дереве наследования классов, внутри которого выполняется поиск. Здесь атрибут «data» обнаруживается в экземплярах, а «setdata» и «display» – в классе, расположенном выше их

его работа заключается лишь в поиске имен в связанных объектах (например, следуя по ссылкам «наследует», как показано на рис. 23.1).

В функции `setdata` внутри класса `FirstClass` значение аргумента записывается в `self.data`. Имя `self` внутри метода – имя самого первого аргумента, в соответствии с общепринятыми соглашениями, – автоматически ссылается на обрабатываемый экземпляр (x или y), поэтому операция присваивания сохраняет значения в пространстве имен экземпляра, а не класса (так создаются имена `data`, показанные на рис. 23.1).

Поскольку классы способны генерировать множество экземпляров, методы должны использовать аргумент `self`, чтобы получить доступ к обрабатываемому экземпляру. Когда мы вызываем метод класса `display`, чтобы вывести значения 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 пришло (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 число с плавающей точкой). Как и повсюду в языке Python, атрибуты экземпляров (иногда называются *членами*) никак не объявляются – они появляются, как только будет выполнена первая операция присваивания, 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 пределами 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 создания 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 затем сможет использоваться любыми методами класса в объекте экземпляра x . Обычно классы создают все атрибуты экземпляров за счет присваивания значений аргументу `self`, но это не обязательно –

программы могут получать, изменять или создавать атрибуты в любых объектах, к которым они имеют доступ.

Классы адаптируются посредством наследования

Помимо роли фабрик по созданию объектов экземпляров, классы также позволяют нам вносить изменения за счет введения новых компонентов (которые называются подклассами), а не за счет изменения существующего программного 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 специализируется за счет переопределения существующих атрибутов ниже в иерархии. Здесь также нет никакого сходства с модулями: их атрибуты находятся в едином плоском пространстве имен.

В языке Python экземпляры наследуют классам, а классы наследуют суперклассам. Ниже приводятся основные идеи, лежащие в основе механизма наследования 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`. Класс, который наследует, называется подклассом, а класс, которому наследуют, называется его *суперклассом*.
- **Классы наследуют атрибуты своих суперклассов.** Как экземпляры наследуют имена атрибутов, определяемых их классами, так же и классы наследуют все имена атрибутов, определяемые в их суперклассах, — интерпретатор автоматически отыскивает их, когда к ним выполняется обращение, если эти атрибуты отсутствуют в подклассах.
- **Экземпляры наследуют атрибуты всех доступных классов.** Каждый экземпляр наследует имена из своего класса, а также из всех его суперклассов. Во время поиска имен интерпретатор проверяет сначала экземпляр, потом 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` (например, `X.attr`), а также ссылки на атрибуты аргумента экземпляра `self` в методах 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) изменяет подклассы 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 рассматривали ранее. С одной стороны, классы позволяют минимизировать избыточность программного кода (и тем самым снизить стоимость обслуживания) за счет создания единой и общей реализации операций, а с другой стороны, они обеспечивают возможность адаптировать уже существующий программный код вместо того, чтобы изменять его или писать заново.

Второй пример

Этот второй пример основан на предыдущем. Для начала мы определим новый класс, `SecondClass`, который наследует 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 экземпляров 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 классе `FirstClass`. В этом случае мы говорим, что класс `SecondClass` *переопределяет* метод `display` класса `FirstClass`. Иногда такая замена атрибутов за счет их переопределения 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 класса `FirstClass` метод `setdata`. Создадим экземпляр, чтобы продемонстрировать 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`, которая выводит измененный текст сообщения. Схема пространств имен, вовлеченных в действие, изображена на рис. 23.2.

Здесь очень важно заметить следующее: специализация, выполненная в классе `SecondClass`, находится полностью *за пределами* класса `First-`

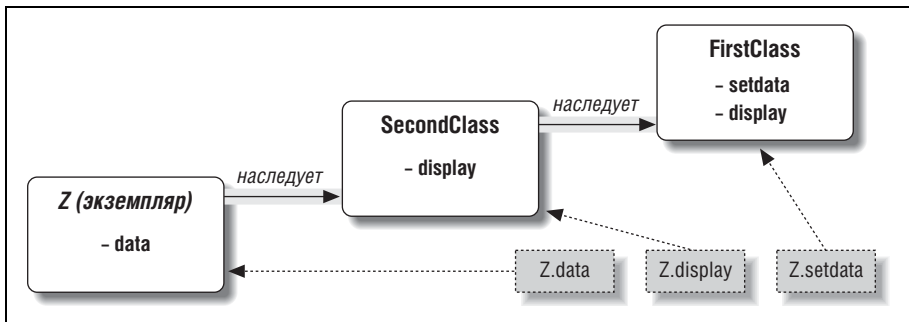


Рис. 23.2. Специализация путем переопределения унаследованных имен посредством повторного их определения ниже в дереве классов. Здесь класс *SecondClass* переопределяет и тем самым адаптирует метод «display» для экземпляров этого класса

Class. Как следствие она не оказывает влияния на уже созданные или создаваемые впоследствии 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. Безусловно, это достаточно искусственный пример, но он демонстрирует, как наследование позволяет вносить изменения с помощью внешних компонентов (то есть с помощью подклассов). Часто классы поддерживают возможность расширения и повторного использования гораздо лучше, чем функции или модули.

Классы – это атрибуты в модулях

Прежде чем двинуться дальше, вспомним, что в именах классов 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 *FirstClass* находилось в файле модуля, а не было введено в интерактивной оболочке, мы могли бы импортировать этот модуль и использовать имя в строке заголовка инструкции `class`:

```
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 рассматривали в пятой части книги. Например, в одном файле модуля могут находиться определения сразу нескольких классов – подобно другим инструкциям в модуле, инструкции `class` выполняются в ходе операции импортирования и определяют имена, которые в свою очередь становятся атрибутами модуля. Вообще любой модуль может содержать самые произвольные сочетания из любого числа 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 одинаковые 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 использовать 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 вещающего файла. Если вам это кажется сложным, то не используйте одинаковые имена для модулей и классов в них.

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: перегрузку операторов. Говоря простым языком, перегрузка операторов позволяет объектам, созданным из классов, перехватывать и участвовать в операциях, которые применяются к встроенным типам: сложение, получение среза, вывод, дополнение имен и т. д. По большей части это автоматический механизм: при выполнении выражений и других встроенных операций интерпретатор передает управление реализации классов. В модулях нет ничего подобного: модули могут реализовать функции, но не операторы выражений.

Мы можем полностью реализовать поведение класса в виде методов, однако перегрузка операторов позволяет объектам теснее интегрироваться в объектную модель языка Python. Кроме того, перегрузка операторов помогает нашим объектам действовать так же, как действуют встроенные объекты, потому что она позволяет создавать менее противоречивые и более простые в изучении интерфейсы объектов и обеспечивает возможность обрабатывать объекты, созданные из классов, программным кодом, который предполагает взаимодействие со встроенными типами.

- **Имена методов, начинающиеся и заканчивающиеся двумя символами подчеркивания (`__X__`), имеют специальное назначение.** Перегрузка операторов в языке Python реализуется за счет создания методов со специальными 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 становится результатом соответствующей операции.
- **Классы могут переопределять большинство встроенных операторов.** Существует множество специальных имен методов для реализации перегрузки почти всех операторов, доступных для встроенных типов. Сюда входят операторы выражений, а также такие базовые операции, как вывод и создание объекта.
- **В методах перегрузки операторов не существует аргументов со значениями по умолчанию, и ни один из таких методов не является обязательным для реализации.** Если класс не определяет и не наследует методы перегрузки операторов, это означает лишь то, что экземпляры класса не поддерживают эти операции. Например, если отсутствует метод `__add__`, попытка выполнить 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 особенностью – она используется в основном специалистами, создающими инструментальные средства для других программистов, а не разработчиками прикладных программ. И честно говоря, вам не стоит использовать ее только потому, что это «круто». Если не требуется, чтобы класс имитировал поведение встроенных типов, лучше ограничиться использованием простых методов. Например, зачем приложению, работающему с базой данных служащих, поддержка таких операторов, как `*` и `+`? Методы с обычными именами, такими как `giveRaise` и `promote`, обычно более уместны.

Вследствие этого мы не будем в этой книге углубляться в подробности реализации методов перегрузки каждого оператора, доступного в языке Python. Однако имеется один метод перегрузки оператора, который можно встретить практически в любом классе: метод `__init__`, который известен как *конструктор* и используется 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 от `SecondClass`, реализующий три специальных метода, которые будут вызываться интерпретатором автоматически: метод `__init__` вызывается, когда создается новый объект экземпляра (аргумент `self` представляет новый объект `ThirdClass`), и методы `__add__` и `__mul__` вызываются, когда экземпляр `ThirdClass` участвует в операциях `+` или `*`, соответственно. Ниже приводится определение нашего класса:

```
>>> class ThirdClass(SecondClass):           # Наследует SecondClass
...     def __init__(self, value):           # Вызов "ThirdClass(value)"
...         self.data = value
...     def __add__(self, other):             # Для выражения "self + other"
...         return ThirdClass(self.data + other)
...     def __mul__(self, other):
...         self.data = self.data * other    # Для выражения "self * other"
...
>>> a = ThirdClass("abc")                   # Вызывается новый метод __init__
>>> a.display()                             # Унаследованный метод
Current value = "abc"

>>> b = a + 'xyz'                           # Новый __add__: создается новый экземпляр
>>> b.display()
Current value = "abcxyz"
```

```
>>> a * 3                                     # Новый __mul__: заменяется сам экземпляр
>>> a.display()
Current value = "abcbcabcb"
```

Класс `ThirdClass` «наследует» классу `SecondClass`, поэтому его экземпляры наследуют метод `display` от `SecondClass`. Но теперь при создании экземпляра класса `ThirdClass` ему передается дополнительный аргумент (например, «abc») – в виде аргумента `value` в конструкторе `__init__`, значение которого присваивается атрибуту `self.data`. Далее, объекты `ThirdClass` могут участвовать в операциях `+` и `*` – объект экземпляра слева от оператора передается в виде аргумента `self`, а значение справа в виде аргумента `other`, как показано на рис. 23.3.

Методы со специальными именами, такими как `__init__` и `__add__`, наследуются 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 (подробнее об этом мы поговорим позднее – метод `__init__`, например, часто вызывается вручную, с целью запустить конструктор суперкласса).

Обратите внимание, что метод `__add__` создает и возвращает *новый* объект экземпляра этого класса (вызывая `ThirdClass`, которому передается значение результата), а метод `__mul__` изменяет *текущий* объект экземпляра (выполняя присваивание атрибуту аргумента `self`). Такое поведение отличается от поведения встроенных типов, таких как числа и строки, которые всегда создают новые объекты при выполнении оператора `*`. Поскольку перегрузка операторов – это в действительности всего лишь механизм отображения выражений на методы, вы мо-

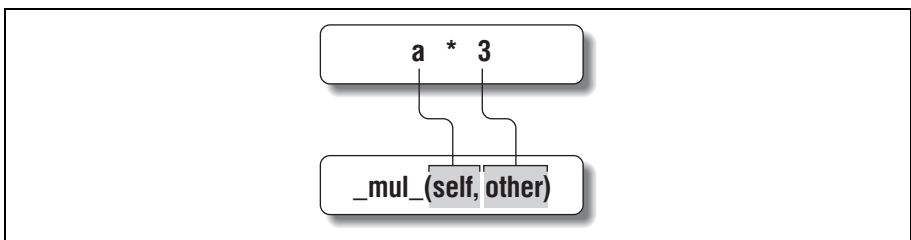


Рис. 23.3. Механизм перегрузки отображает операторы выражений и другие встроенные операции, выполняемые над экземплярами классов, на специальные имена методов в классе. Эти специальные методы являются необязательными и могут наследоваться, как обычные методы. Здесь использование оператора «*» приводит к вызову метода «`__mul__`»

жете интерпретировать операторы в своих объектах классов, как вам будет угодно.¹

С другой стороны, вы могли бы использовать перегрузку операторов, чтобы иметь возможность передавать объекты, определяемые пользователем, в функцию, 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 самых интерфейсов и, как следствие, совместимость с используемой функцией.

Один из методов перегрузки присутствует практически во всех реалистичных классах: метод-конструктор `__init__`. Он позволяет классам немедленно заполнять атрибуты вновь созданных экземпляров, поэтому конструктор полезно использовать практически во всех разновидностях ваших классов. Фактически даже при том, что в языке Python атрибуты не объявляются, вы без труда сможете определить, какие атрибуты принадлежат экземплярам, просмотрев программный код метода `__init__`. В этой книге мы не будем углубляться в изучение этой расширенной особенности, но мы еще будем рассматривать некоторые приемы использования механизма наследования и перегрузки операторов в главе 24.

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 вам, что сама модель наследования, на которой основаны классы, очень проста – суть ее заключается всего лишь в организации 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 потому, что здесь у нас нет никаких методов с программным кодом. После создания класса, выполнив инструкцию в интерактивной оболочке, мы можем приступать к присоединению атрибутов, выполняя операции присваивания из-за пределов класса:

```
>>> rec.name = 'Bob'      # Так же для объектов с атрибутами
>>> rec.age = 40
```

¹ Но лучше этого не делать. Обычная практика показывает, что перегруженные операторы должны работать так же, как и встроенные реализации. В данном случае это означает, что метод `__mul__` должен был бы возвращать новый объект в виде результата, а не изменять существующий экземпляр (`self`); для непосредственного изменения объекта лучше предусмотреть метод `mul`, чем выполнять перегрузку оператора `*` (например, `mul(3)`, вместо `a*3`).

После того как атрибуты будут созданы, мы можем обращаться к ним с помощью обычного синтаксиса. Когда класс используется таким способом, он напоминает структуры в языке C или записи в языке Pascal – объект с присоединенными 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 объекты, даже если нет ни одного экземпляра. Фактически они всего лишь самостоятельные пространства имен, поэтому, пока у нас имеется ссылка на класс, мы можем в любое время добавлять или изменять его атрибуты по своему усмотрению. 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 получают атрибуты, которые мы присоединили к классу:

```
>>> x.name, y.name           # Сейчас имена хранятся только в классе
('Bob', 'Bob')
```

В действительности у этих экземпляров нет своих собственных атрибутов – они просто получают атрибут `name` из класса. Тем не менее если выполнить присваивание атрибуту экземпляра, будет создан (или изменен) атрибут этого объекта, а не другого – атрибуты обнаруживаются в результате поиска по дереву наследования, 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 следующей главе, атрибуты объекта пространства имен обычно реализованы в виде словарей, и деревья наследования классов (вообще говоря) тоже всего лишь словари со ссылками на другие словари. Если знать, куда смотреть, в этом можно убедиться явно. В большинстве объектов, созданных на базе классов, имеется атрибут `__dict__`, который является словарем пространства имен:

```
>>> rec.__dict__.keys()
['age', '__module__', '__doc__', 'name']
```

```
>>> x.__dict__.keys()
['name']

>>> y.__dict__.keys()
[]
```

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 называется `__class__`, если вам захочется проверить ее:

```
>>> x.__class__
<class '__main__.rec at 0x00BAFF60>
```

Классы также имеют атрибут `__base__`, который представляет собой кортеж его суперклассов – эти два атрибута описывают, как деревья классов размещаются в памяти.

Главное, что следует из этого взгляда на внутреннее устройство, это то, что модель 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 выполняются 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 класса. Например, ниже определяется простая функция вне какого-либо класса, 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 особого смысла). 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

¹ Фактически это одна из причин, почему в языке Python аргумент `self` всегда должен явно объявляться в методах, – потому что методы могут создаваться как простые функции, независимо от классов, и им необходим явный аргумент со ссылкой на подразумеваемый экземпляр. В противном случае интерпретатор не смог бы обеспечить превращение простой функции в метод класса. Однако основная причина, по которой аргумент `self` объявляется явно, заключается в том, чтобы сделать назначение имен более очевидным. Имена, к которым обращаются не через аргумент `self`, являются простыми переменными, тогда как имена, обращение к которым происходит через аргумент `self`, очевидно являются атрибутами экземпляра.

```
>>> 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 атрибуты экземпляров создаются в результате присваивания значений атрибутам аргумента `self` в методах. Однако отметим снова, что все это не является обязательным, поскольку ООП в языке Python – это в основном поиск атрибутов во взаимосвязанных объектах пространств имен.

В заключение

В этой главе были представлены основы программирования классов в языке Python. Мы изучили синтаксис инструкции `class` и увидели, как ее можно использовать для построения дерева классов. Мы также узнали, что интерпретатор 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).

В следующей главе мы продолжим изучение программирования классов и повторно рассмотрим модель, чтобы восполнить недостающие подробности, которые были опущены здесь для простоты. Мы также приступим к исследованию нескольких больших классов, обладающих большей практической ценностью. Однако для начала ответьте на контрольные вопросы, чтобы освежить в памяти основы, которые были рассмотрены 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 объекта 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 операторов. В некотором смысле модули напоминают классы с единственным экземпляром, 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 присваивание атрибутам класса в любом месте, где доступна ссылка на объект класса, то есть даже за пределами инструкции `class`.
4. Атрибуты экземпляра создаются присваиванием атрибутам объекта экземпляра. 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`. 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 аргумент не обязательно должен носить имя `self`, главное – это положение аргумента. (Бывшие программисты на C++ или Java могут предпочесть использовать имя `this`, потому что в этих языках программирования это имя является отра-

жением 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 символами подчеркивания. Эти имена не являются встроенными или зарезервированными именами – интерпретатор Python просто автоматически вызывает методы с этими именами, когда экземпляр появляется в составе соответствующей операции. Язык Python определяет порядок отображения операций на специальные имена методов.
7. Перегрузка операторов может использоваться для реализации объектов, которые имитируют поведение встроенных типов (например, последовательностей или числовых объектов, таких как матрицы), и для реализации интерфейсов 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 использовать 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 начальных операций.
9. Наиболее важными концепциями ООП в языке Python являются аргумент `self` в методах и конструктор `__init__`.

24

Подробнее о программировании классов

Если что-то в главе 23 осталось для вас непонятным, не волнуйтесь – теперь, совершив краткий тур, мы начнем копать немного глубже и подробно изучим понятия, представленные ранее. В этой главе мы с другой стороны посмотрим на классы и методы, наследование и перегрузку операторов, формализуем и дополним некоторые идеи программирования классов, представленные в главе 23. Поскольку классы являются нашим последним инструментом пространств имен, то здесь мы также сведем вместе концепции пространств имен в языке Python. Кроме того, в этой главе будут представлены примеры более крупных и более реалистичных классов, чем те, что мы видели до сих пор, в том числе завершающий пример, который связывает воедино многое из того, что мы узнали об ООП.

Инструкция `class`

Несмотря на то что на первый взгляд инструкция `class` в языке Python напоминает 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 языке C++, инструкция `class` в языке Python является основным инструментом ООП, но в отличие от инструкции в C++, в языке Python она не является объявлением. Подобно инструкции `def`, инструкция `class` создает объект и является неявной инструкцией присваивания – когда она выполняется, создается объект класса, ссылка на который сохраняется в имени, использованном в заголовке инструкции. Кроме того, как и инструкция `def`, инструкция `class` является 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).

Общая форма

Инструкция `class` – это составная инструкция, с блоком операторов, обычно под строкой заголовка. В заголовке после имени в круглых скобках через запятую перечисляются суперклассы. Наличие более одного суперкласса в списке означает множественное наследование (которое будет обсуждаться в следующей главе). 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 класса, а методы со специальными именами перегружают операторы. Например, функция с именем `__init__`, если она определена, вызывается во время создания объекта экземпляра.

Пример

Как мы уже видели, классы – это всего лишь пространства имен, то есть инструменты, 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 течение 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`, становятся атрибутами объекта класса.

Основное отличие классов состоит в том, что их пространства имен также составляют основу механизма наследования в языке Python – ссылки на атрибут, 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 т. д. Все инст-

рукции внутри инструкции `class` выполняются, когда выполняется сама инструкция `class` (а не когда позднее класс вызывается для создания экземпляра). Операции 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 создает *атрибуты данных*, совместно используемых всеми экземплярами:

```
>>> class SharedData:
...     spam = 42          # Создает атрибут данных класса
...
>>> 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 инструкции `class`, оно присоединяется к классу и поэтому совместно используется 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, доступной всем экземплярам, например для хранения счетчика количества созданных экземпляров (эту идею мы рассмотрим в главе 26). 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 наследования производится только при попытке *чтения* атрибута, но не при присваивании: операция присваивания атрибуту

¹ Если у вас есть опыт работы с языком C++, вы можете заметить в этом некоторое сходство со «статическими» членами данных в языке C++ – членами, которые хранятся в классе независимо от экземпляров. В языке Python в этом нет ничего особенного: 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 другое («члены» в C++).

объекта всегда изменяет сам объект, а не что-то другое.¹ Например, атрибут `y.spam` будет найден в наследуемом классе, а операция присваивания атрибуту `x.spam` присоединит имя непосредственно 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
```

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 объекта класса. Как и все атрибуты класса, атрибут `data` наследуется и используется всеми экземплярами класса, которые не имеют собственного атрибута `data`.

Когда создаются экземпляры этого класса, имя `data` присоединяется к этим экземплярам 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                           # a Subclass.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 присваиванием 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 определяем их области видимости. Атрибуты классов совместно используются всеми экземплярами, а атрибуты экземпляров уникальны для каждого экземпляра – ни данные, ни поведение экземпляра недоступны для совместного использования. Несмотря на то что операция поиска в дереве наследования позволяет отыскивать имена, мы

¹ При условии, что класс не переопределил операцию присваивания с помощью метода перегрузки оператора `__setattr__` с целью выполнять какие-то особые действия.

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. Однако 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 обычные объекты функций, которые создаются инструкциями `def` в теле инструкции `class`. Говоря вкратце, методы реализуют поведение, наследуемое объектами экземпляров. С точки зрения программирования методы работают точно так же, как и обычные функции, с одним важным исключением: в первом аргументе методам всегда передается подразумеваемый объект экземпляра.

Другими словами, интерпретатор автоматически отображает вызов метода экземпляра 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 наследования. Фактически в языке Python обе формы вызова метода являются допустимыми.

Помимо обычного наследования имен методов, первый специальный аргумент – 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 экземпляру, то есть к субъекту вызова – поскольку из классов может создаваться 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 использовать имя `self` для получения или изменения атрибутов экземпляра, обрабатываемого текущим вызовом метода. Такая явная природа аргумента `self` предусмотрена намеренно – присутствие этого имени делает очевидным использование имен атрибутов экземпляра.

Пример

Чтобы пояснить эти концепции, обратимся к примеру. Предположим, что имеется 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, поэтому 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 составного имени экземпляра, как в данном случае, то сначала определяется местонахождение метода `printer`, а затем его аргументу `self` автоматически присваивается объект экземпляра (`x`). В аргумент `text` записывается строка, переданная в вызов метода (`'instance call'`). Обратите внимание, что Python автоматически передает в первом 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 текущий 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, явно передавая при этом экземпляр в аргументе `self`:

```
>>> NextClass.printer(x, 'class call') # Прямой вызов метода класса
class call

>>> x.message                    # Экземпляр снова изменился
'class call'
```

Вызов метода, который производится через экземпляр и через имя класса, оказывает 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 вызовом конструктора. Метод `__init__` наследуется точно так же, как и любые другие атрибуты. Это означает, что во время создания экземпляра интерпретатор отыскивает только один метод `__init__`. Если в конструкторе подкласса необходимо гарантировать выполнение действий, предусматриваемых конструктором 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 конструктор суперкласса таким способом следует, только если это действительно необходимо – без этого вызова подкласс полностью переопределяет данный метод. Более реалистичный случай применения этого приема приводится в заключительном примере этой главы.¹

Другие возможности методов

Такой способ вызова методов через имя класса представляет собой основу для расширения (без полной замены) поведения унаследованных методов. В главе 26 мы познакомимся с еще одной возможностью, добавленной в Python 2.2, *статическими методами*, которые не предполагают наличие объекта экземпляра в первом аргументе. Такие методы могут действовать как обычные функции, имена которых являются локальными по отношению к классам, где они были определены. Однако это дополнительное расширение не является обязательным – обычно нам всегда бывает необходимо передавать экземпляр методам, вызываемым либо через сам экземпляр, либо через имя класса.

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 25.

Наследование

Основное назначение такого инструмента пространств имен, как инструкция `class`, заключается в обеспечении поддержки наследования имен. В этом разделе мы подробно остановимся на вопросах, связанных с механизмами наследования атрибутов в языке Python.

В языке Python наследование включается в игру после того, как объект квалифицирован, и заключается в операции поиска в дереве определенных 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 атрибута `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 находятся выше, механизм наследования составляет основу специализации программного кода.

Создание дерева атрибутов

На рис. 24.1 приводятся способы, которыми создаются и заполняются именами деревья пространств имен. Вообще:

- Атрибуты экземпляров создаются посредством присваивания атрибутам аргумента `self` в методах.

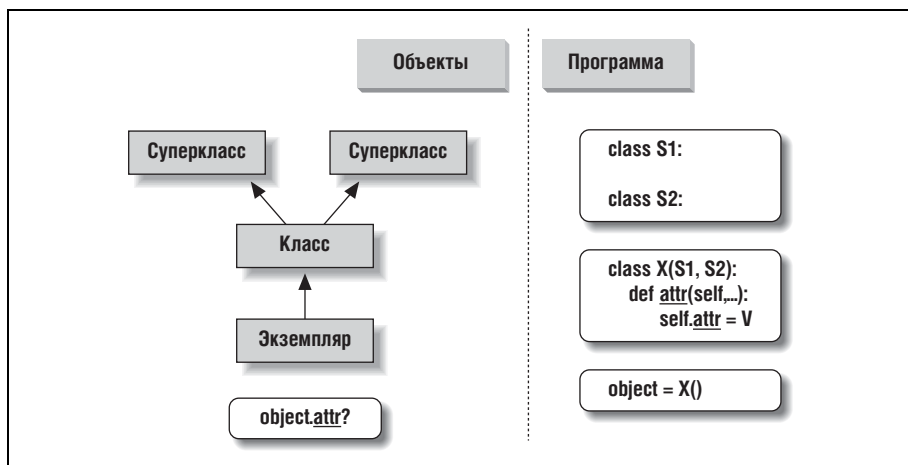


Рис. 24.1. Программный код создает дерево наследования объектов в памяти, в котором будет выполняться поиск атрибутов. Вызов класса создает новый экземпляр, который помнит, к какому классу он принадлежит. Инструкция `class`, в заголовке которой в круглых скобках перечислены суперклассы, создает новый класс. Каждое обращение к атрибуту вызывает новую процедуру поиска в дереве наследования снизу вверх – даже обращения к атрибутам аргумента `self` в методах класса

- Атрибуты классов создаются инструкциями (присваивания), расположенными 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`. Интерпретатор выполняет поиск в дереве в направлении снизу вверх, от экземпляров к суперклассам, всякий раз, когда используется имя подразумевает атрибут объекта экземпляра.¹

Специализация унаследованных методов

Только что описанная модель поиска в дереве наследования представляет собой прекрасный способ специализации программ. Поскольку механизм наследования сначала пытается отыскать имена в подклассах и только потом в их суперклассах, подклассы могут изменять поведение по умолчанию, предусматриваемое 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* унаследованные атрибуты, *предоставлять* атрибуты, которые ожидаются 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):
...         print 'starting Sub.method' # Переопределить метод
...         Super.method(self)        # Дополнительное действие
...         print 'ending Sub.method'  # Выполнить действие по умолчанию
...
```

Главное здесь – это прямые вызовы методов суперкласса. Класс `Sub` замещает метод `method` класса `Super` своей собственной, специализирован-

¹ Это описание далеко не полное, потому что точно так же возможно создавать атрибуты экземпляров и классов с помощью инструкций присваивания за пределами инструкций `class` – но этот прием используется существенно реже и зачастую более подвержен 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 26.

ной версией. Но внутри замещающего метода в классе `Sub` производится вызов версии, экспортируемой 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 примерами обращайтесь к предыдущему разделу «Методы».

Приемы организации взаимодействия классов

Расширение – это лишь один из способов организации взаимодействий с суперклассом. В файле ниже, *specialize.py*, определяется несколько классов, которые 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 определено 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 способах, 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

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. Когда через экземпляр класса `Provider` вызывается метод `delegate`, иницируются *две* независимые процедуры поиска:

1. При вызове `x.delegate` интерпретатор отыскивает метод `delegate` в классе `Super`, начиная поиск от экземпляра класса `Provider` и двигаясь вверх по дереву наследования. Экземпляр `x` передается методу в виде аргумента `self`, как обычно.
2. Внутри метода `Super.delegate` выражение `self.action` приводит к запуску нового, независимого поиска в дереве наследования, начиная от экземпляра `self` и дальше вверх по дереву. Поскольку аргумент `self` ссылается на экземпляр класса `Provider`, метод `action` будет найден в подклассе `Provider`.

Такой способ «восполнения пробелов» в реализации – обычное дело для платформ ООП. По крайней мере, в терминах метода `delegate` такие суперклассы, как в этом примере, иногда называют *абстрактными суперклассами* – классы, которые предполагают, что часть их функциональности будет реализована их подклассами. Если ожидаемый метод не определен в подклассе, интерпретатор возбудит исключение с сообщением о неопределенном имени, когда поиск в дереве наследования завершится неудачей. Разработчики классов иногда делают такие требования к подклассам более очевидными с помощью инструкций `assert` или возбуждая встроенное исключение `NotImplementedError`:

```
class Super:
    def method(self):
        print 'in Super.method'
    def delegate(self):
        self.action()
    def action(self):
        assert 0, 'action must be defined!'
```

Мы познакомимся с инструкцией `assert` в главе 27, а пока лишь замечу, что если выражение возвращает ложь, она возбуждает исключение с сообщением об ошибке. В данном случае выражение всегда возвращает ложь (0), чтобы вызвать появление об ошибке, если метод не будет переопределен и поиск по дереву наследования остановится на этой версии. В некоторых классах, напротив, в таких методах-заглушках исключение `NotImplementedError` возбуждается напрямую. Инструкцию возбуждения исключений `raise` мы рассмотрим в главе 27.

Более реалистичный пример использования концепций, представленный в этом разделе, вы найдете в упражнении 8 в конце главы 26 и в решении этого упражнения в разделе «Часть VI, Классы и ООП» (приложение В). Такое частичное наследование является традиционным способом введения в ООП, но оно постепенно исчезает из арсенала многих разработчиков.

Перегрузка операторов

Перегрузку операторов мы коротко рассмотрели в предыдущей главе, а здесь мы обсудим все более детально и рассмотрим несколько наиболее

часто используемых методов перегрузки. Ниже приводится краткий обзор ключевых идей, лежащих в основе механизма 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 функций, обращение к атрибутам и т. д.
- Перегрузка делает экземпляры классов более похожими на встроенные типы.
- Перегрузка заключается в реализации в классах методов со специальными именами.

Рассмотрим простой пример перегрузки в действии. Если в классе присутствуют определенные методы со специальными именами, интерпретатор автоматически будет вызывать их при появлении экземпляров класса в выражениях, выполняющих перегруженные операции. Например, класс `Number` в следующем файле `number.py` реализует метод перегрузки операции создания экземпляра (`__init__`), а также метод реализации 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 - другой
        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 оператора в языке Python, потому что он присутствует в большинстве классов. В этом разделе мы изучим некоторые другие инструменты, связанные с перегрузкой, и рассмотрим 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 методов перегрузки операторов. В табл. 24.1 перечислены наиболее часто используемые, но на самом деле их намного больше. На деле многие методы перегрузки существуют в нескольких версиях (например, `__add__`, `__radd__` и `__iadd__` для сложения). Исчерпывающий

список имен специальных методов вы найдете в других книгах, посвященных языку Python, и в справочных руководствах.

Таблица 24.1. Общие методы перегрузки операторов

Метод	Перегружает	Вызывается
<code>__init__</code>	Конструктор	При создании объекта: <code>X = Class()</code>
<code>__del__</code>	Деструктор	При уничтожении объекта
<code>__add__</code>	Оператор +	<code>X + Y</code> , <code>X += Y</code>
<code>__or__</code>	Оператор (побитовое ИЛИ)	<code>X Y</code> , <code>X = Y</code>
<code>__repr__</code> , <code>__str__</code>	Вывод, преобразование	<code>print X</code> , <code>repr(X)</code> , <code>str(X)</code>
<code>__call__</code>	Вызовы функций	<code>X()</code>
<code>__getattr__</code>	Обращение к атрибуту	<code>X.undefined</code>
<code>__setattr__</code>	Присваивание атрибутам	<code>X.any = value</code>
<code>__getitem__</code>	Доступ к элементу по индексу	<code>X[key]</code> , циклы <code>for</code> и другие конструкции итерации, при отсутствии метода <code>__iter__</code>
<code>__setitem__</code>	Присваивание элементу по индексу	<code>X[key] = value</code>
<code>__len__</code>	Длина	<code>len(X)</code> , проверка истинности
<code>__cmp__</code>	Сравнение	<code>X == Y</code> , <code>X < Y</code>
<code>__lt__</code>	Специальное сравнение	<code>X < Y</code> (в ином случае <code>__cmp__</code>)
<code>__eq__</code>	Специальное сравнение	<code>X == Y</code> (в ином случае <code>__cmp__</code>)
<code>__radd__</code>	Правосторонний оператор +	<code>Не_экземпляр + X</code>
<code>__iadd__</code>	Добавление (увеличение)	<code>X += Y</code> (в ином случае <code>__add__</code>)
<code>__iter__</code>	Итерационный контекст	Циклы <code>for</code> , оператор <code>in</code> , генераторы списков, <code>map</code> и другие

Все методы перегрузки имеют имена, начинающиеся и заканчивающиеся двумя 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 Python (и описывается в стандартном руководстве по языку). Например, по определению языка оператор + всегда отображается на имя `__add__` независимо от того, что в действительности делает метод `__add__`.

Все методы перегрузки операторов являются необязательными – если какой-то метод не реализован, это лишь означает, что соответствующая ему операция не поддерживается классом (а при попытке применить такую операцию возбуждается исключение). Большинство методов перегрузки операторов используются только при решении специальных

задач, когда необходимо, чтобы объекты имитировали поведение встроенных типов, однако конструктор `__init__` присутствует в большинстве классов. Мы уже познакомились с конструктором `__init__`, который вызывается на этапе инициализации, и с несколькими другими, перечисленными в табл. 24.1. Теперь мы исследуем примеры использования некоторых других методов из таблицы.

`__getitem__` реализует доступ к элементам по индексу

Метод `__getitem__` реализует операции доступа к элементам по индексу. Когда экземпляр `X` появляется в выражении извлечения элемента по индексу, таком как `X[i]`, интерпретатор Python вызывает метод `__getitem__`, наследуемый этим экземпляром (если он имеется), передавая методу объект `X` в первом аргументе и индекс в квадратных скобках во втором аргументе. Например, следующий класс возвращает квадрат значения индекса:

```
>>> class indexer:
...     def __getitem__(self, index):
...         return index ** 2
...
>>> X = indexer()
>>> X[2]                                # Выражение X[i] вызывает __getitem__(X, i).4
>>> for i in range(5):
...     print X[i],
...
0 1 4 9 16
```

`__getitem__` и `__iter__` реализуют итерации

Здесь описывается прием, который не всегда очевиден для начинающих программистов, но на практике может оказаться необычайно полезным. Инструкция `for` многократно применяет операцию индексирования к последовательности, 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 реализован, инструкции циклов `for` будут вызывать его на каждом шаге цикла, с постоянно 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 индексирования, 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,                     # for индексирует элементы 0..N
...
S p a m
```

Фактически это случай, когда «купив один предмет, в подарок получаешь целую связку». Любой класс, поддерживающий циклы `for`, автоматически поддерживает все итерационные контексты, имеющиеся в языке Python, многие из которых мы видели в более ранних главах (другие итерационные контексты описываются в главе 13). Например, оператор проверки на принадлежность `in`, генераторы списков, встроенная функция `map`, присваивание списков и кортежей и конструкторы типов также автоматически вызывают метод `__getitem__`, если он определен:

```
>>> 'p' in X                            # Во всех этих случаях вызывается __getitem__
1

>>> [c for c in X]                      # Генератор списков
['S', 'p', 'a', 'm']

>>> map(None, X)                        # Функция map
['S', 'p', 'a', 'm']

>>> (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, которые реализуют интерфейс последовательностей, и для добавления логики к операциям над встроенными типами – мы рассмотрим эту идею, когда будем расширять встроенные типы в главе 26.

Итераторы, определяемые пользователями

В настоящее время все итерационные контексты в языке Python пытаются сначала использовать метод `__iter__` и только потом – метод `__getitem__`. То есть при выполнении обхода элементов объекта предпочтение отдается итерационному протоколу, с которым мы познакомились в главе 13, – если итерационный протокол не поддерживается объектом, вместо него используется операция индексирования.

С технической точки зрения итерационные контексты вызывают встроенную функцию `iter`, чтобы определить наличие метода `__iter__`, который должен возвращать объект итератора. Если он предоставля-

ется, 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 использованием метода `__getitem__` и начинает извлекать элементы по индексам, пока не будет возбуждено исключение `IndexError`.

В новой схеме классы реализуют итераторы простой реализацией итерационного протокола, представленного в главах 13 и 17 (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:
            raise StopIteration
        self.value += 1
        return self.value ** 2

% python
>>> from iters import Squares
>>> for i in Squares(1, 5):        # for вызывает iter(),
                                   # который вызывает __iter__()
...     print i,                  # 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 информацией 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).

Эквивалентная реализация с использованием `__getitem__` может оказаться менее 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, возвращаемые методом `__iter__`, явно манипулируют информацией о своем состоянии и сохраняют ее между вызовами метода `next`, такая реализация может быть более универсальной, чем использование метода `__getitem__`.

С 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 индексации – 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 итерационных контекстах, к которым применим метод `__getitem__` (проверка на вхождение, конструкторы, присваивание последовательностей и т. д.). Однако, 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 итерации необходимо будет создавать новый объект итератора:

```
>>> 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 представлены в главе 17):

```
>>> from __future__ import generators # Требуется только в Python 2.2
>>>
>>> def gsquares(start, stop):
...     for i in range(start, stop+1):
...         yield i ** 2
...
>>> for i in gsquares(1, 5):
...     print i,
...
1 4 9 16 25
```

В отличие от класса, функция автоматически сохраняет информацию о своем состоянии между итерациями. Конечно, для реализации такого искусственного примера можно было бы вообще не использовать ни один из этих приемов, а просто использовать цикл `for`, функцию `map` или генератор списков, чтобы создать сразу весь список. Нередко самый лучший и самый быстрый способ в языке 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 моделировании более сложных итераций, особенно когда возможность сохранения информации 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 поддержку протекания нескольких итерационных процессов с одним и тем же набором 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,
...
aa ac ae ca cc ce ea ec ee
```

Здесь внешний цикл получает итератор строки вызовом функции `iter`, и каждый вложенный цикл делает то же самое, чтобы получить независимый итератор. Так как каждый итератор хранит свою собственную информацию о состоянии, каждый цикл управляет своим собственным положением в строке, независимо от любых других активных циклов. Чтобы добиться того же эффекта в итераторах, определяемых пользователем, метод `__iter__` должен не просто возвращать аргумент `self`, а создавать новый объект итератора со своей информацией о состоянии.

Например, в следующем примере определяется класс итератора, который пропускает каждый второй элемент. Поскольку объект итератора создается заново для каждой итерации, он обеспечивает поддержку нескольких активных циклов 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 элемент,
        self.wrapped = wrapped           # который будет использоваться
```



```

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 I.next(), I.next(), I.next()   # 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,                 # Каждый итератор помнит свое состояние, смещение

```

Этот пример работает подобно вложенным циклам со строками – каждый активный цикл запоминает свое положение в строке, потому что каждый 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

```

Наш более ранний пример класса `Squares`, напротив, поддерживал всего одну активную итерацию, нужно было во вложенных циклах вызывать `Squares` снова, 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 использованием 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,
...
aa ac ae ca cc ce ea ec ee

```

Однако это далеко не то же самое по двум причинам. Во-первых, каждое выражение извлечения среза физически сохраняет весь список с результатами в памяти, тогда как итераторы воспроизводят по одному значению за раз, что позволяет существенно экономить память в случае большого объема результатов. Во-вторых, операции извлечения среза создают новые объекты, поэтому в действительности 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 извлечения 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,
...
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 памяти (на сегодняшний день не существует генераторов, способных формировать 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 предполагается в данном примере. Независимо от того, требуется ли такая широта возможностей в наших приложениях, итераторы, определяемые пользователем, представляют собой мощный инструмент – они позволяют создавать произвольные объекты, которые выглядят и ведут себя подобно другим последовательностям и итерируемым объектам, с которыми мы встречались в этой книге. 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.

__getattr__ и __setattr__ перехватывают обращения к атрибутам

Метод `__getattr__` выполняет операцию получения ссылки на атрибут. Если говорить более определенно, он вызывается с именем атрибута в виде строки всякий раз, когда обнаруживается попытка получить ссылку на *неопределенный* (несуществующий) атрибут. Этот метод не вызывается, если интерпретатор может обнаружить атрибут посредством выполнения процедуры поиска в дереве наследования. Вследствие этого метод `__getattr__` удобно использовать для универсальной 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 и его экземпляр `X` не имеют своих собственных атрибутов, поэтому при обращении к атрибуту `X.age` вызывается метод `__getattr__` – 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, возбуждается встроенное исключение `AttributeError`, чтобы сообщить интерпретатору, что это действительно неопределенные имена, – попытка обращения к имени `X.name` приводит к появлению ошибки. Вы еще раз встретитесь с методом `__getattr__`, когда мы будем рассматривать делегирование и свойства в действии в следующих двух главах, а об исключениях я подробно буду рассказывать в седьмой части книги.

Родственный ему метод перегрузки `__setattr__` перехватывает *все* попытки присваивания 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 выполнить 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 посредством 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 контролировать или специализировать доступ к атрибутам в ваших объектах. Они могут играть весьма специфические роли, часть из которых мы рассмотрим далее в этой книге.

Имитация частных атрибутов экземпляра

Следующий фрагмент является обобщением предыдущего примера и позволяет каждому подклассу иметь свой перечень частных имен атрибутов, которым нельзя присваивать значения:

```
class PrivateExc(Exception): pass          # Подробнее об исключениях позднее

class Privacy:
    def __setattr__(self, attrname, value): # Вызывает self.attrname = value
        if attrname in self.privates:
            raise PrivateExc(attrname, self)
        else:
            self.__dict__[attrname] = value # Self.attrname = value
                                           # вызовет заикливание!

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.

Полную реализацию я оставляю как упражнение для самостоятельного решения, потому что хотя таким способом можно имитировать сокрытие атрибутов, это практически никогда не используется. Программисты, использующие язык Python, способны писать крупные объектно-ориентированные платформы без частных объявлений, но существующие интересные решения по управлению доступом далеко выходят за рамки нашего обсуждения.

Перехват операций обращения к атрибутам и присваивания им значений – вообще очень полезный прием. Он обеспечивает возможность *делегирования* – способ, позволяющий обертывать встроенные объекты объектами-контроллерами, добавлять новое поведение и делегировать

выполнение операций обернутым объектам (подробнее о делегировании и классах-обертках рассказывается в следующей главе).

`__repr__` и `__str__` возвращают строковое представление

В следующем примере реализованы конструктор `__init__` и метод перегрузки `__add__`, которые мы уже видели, но также в нем реализован метод `__repr__`, который возвращает строковое представление экземпляров. Здесь этот метод используется для преобразования объекта `self.data` в строку. Если метод `__repr__` (или родственный ему метод `__str__`) определен, он автоматически будет вызываться при попытках вывести экземпляр класса или преобразовать его в строку. Эти методы позволяют определить более удобочитаемый формат вывода ваших объектов:

```
>>> class adder:
...     def __init__(self, value=0):
...         self.data = value           # Инициализировать атрибут data
...     def __add__(self, other):
...         self.data += other          # Прибавить другое значение
...
>>> class addrepr(adder):
...     def __repr__(self):              # Добавляет строковое представление
...         return 'addrepr(%s)' % self.data # Преобразует в строку
...                                         # программного кода
>>> x = addrepr(2)                      # Вызывает __init__
>>> x + 1                               # Вызывает __add__
>>> x                                  # Вызывает __repr__
addrepr(3)
>>> print x                            # Вызывает __repr__
addrepr(3)
>>> str(x), repr(x)                    # Вызывает __repr__
('addrepr(3)', 'addrepr(3)')
```

Почему имеется два метода вывода? Дело в том, что сначала выполняется попытка использовать метод `__str__`, чтобы вывести объект в удобном для пользователя виде, как это делают инструкция `print` и встроенная функция `str`. Метод `__repr__` должен возвращать строку, которая могла бы использоваться как программный код для воссоздания объекта, — он используется при автоматическом выводе результатов выражений в интерактивной оболочке и в функции `repr`. Если метод `__str__` отсутствует, интерпретатор использует метод `__repr__` (но не наоборот):

```
>>> class addstr(adder):
...     def __str__(self):               # есть __str__, но нет метода __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                            # Вызывает __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 метода, 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(addstr):
...     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                                           # Вызывает __repr__
addboth(5)
>>> print x                                    # Вызывает __str__
[Value: 5]
>>> str(x), repr(x)
('[Value: 5]', 'addboth(5)')
```

Похоже, что метод `__str__` (и его низкоуровневый родственник `__repr__`) является 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 скорее всего был использован один из этих методов.

`__radd__` обрабатывает правостороннее сложение

С технической точки зрения метод `__add__`, который использовался в примерах выше, не поддерживает использование объектов экземпляров справа от оператора `+`. Чтобы реализовать поддержку таких выражений и тем самым обеспечить допустимость перестановки операндов, необходимо создать метод `__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):
...         print 'add', self.val, other
...     def __radd__(self, other):
...         print 'radd', self.val, other
... 
```

```

>>> x = Commuter(88)
>>> y = Commuter(99)
>>> x + 1          # __add__: экземпляр + не_экземпляр
add 88 1
>>> 1 + y          # __radd__: не_экземпляр + экземпляр
radd 99 1
>>> x + y          # __add__: экземпляр + не_экземпляр
add 88 <__main__.Commuter instance at 0x0086C3D8>

```

Обратите внимание на изменение порядка следования операндов в вызове метода `__radd__`: аргумент `self` в действительности находится справа от оператора `+`, а аргумент `other` — слева. Любой двухместный оператор имеет похожий правосторонний метод перегрузки (например, `__mul__` и `__rmul__`). Обычно правосторонний метод, такой как `__radd__`, просто изменяет порядок следования операндов и повторно выполняет операцию сложения, чтобы вызвать метод `__add__`, в котором находится основная реализация операции. Кроме того, следует заметить, что здесь `x` и `y` — это экземпляры одного и того же класса — когда в выражении участвуют экземпляры разных классов, интерпретатор предпочитает вызывать метод экземпляра, расположенного слева.

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 реализация поддержки этого оператора. Например, эти методы могут использоваться в классе `Vector`, но в таких классах, как `Employee` или `Button`, скорее всего, они не нужны.

`__call__` обрабатывает вызовы

Метод `__call__` вызывается при вызове вашего экземпляра. Это не повторяющееся определение — если метод `__call__` присутствует, интерпретатор будет вызывать его, когда экземпляр вызывается как функция. Это позволяет экземплярам классов имитировать поведение функций:

```

>>> class Prod:
...     def __init__(self, value):
...         self.value = value
...     def __call__(self, other):
...         return self.value * other
...
>>> x = Prod(2)
>>> x(3)
6
>>> x(4)
8

```

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 прикладными интерфейсами, где ожидается функция, – это позволяет создавать объекты, совместимые с ожидающими получить функцию интерфейсами, которые 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, с которым мы познакомимся далее в этой книге, позволяет регистрировать функции как обработчики событий (они же – функции обратного вызова); когда возникают какие-либо события, Tkinter вызывает зарегистрированные объекты. Если вам необходимо реализовать обработчик событий, способный сохранять свое состояние между вызовами, вы можете зарегистрировать либо связанный метод класса, либо экземпляр класса, который с помощью метода `__call__` обеспечивает совместимость с функциональным интерфейсом. В программном коде этого раздела оба варианта – `x.comp` из второго примера и экземпляр `x` из первого – могут передаваться в виде объектов функций.

В следующей главе я более подробно расскажу о связанных методах, а пока разберем гипотетический пример использования метода `__call__` для построения графического интерфейса. Следующий класс определяет объект, поддерживающий 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 графического интерфейса как обработчики событий для кнопок,

Этот прием является более простым, но менее универсальным, чем перегрузка операции вызова с помощью метода `__call__`. Еще раз напомним, что подробнее о связанных методах будет рассказываться в следующей главе.

Кроме того, в главе 26 будет представлен еще один пример использования метода `__call__`, который будет использоваться для реализации так называемого *декоратора функции* – вызываемого объекта, добавляющего уровень логики поверх внедренной функции. Поскольку метод `__call__` позволяет присоединять информацию о состоянии к вызываемым объектам, этот прием является естественным для реализации функций, которые должны запоминать и вызывать другие функции.

`__del__` – это деструктор

Конструктор `__init__` вызывается во время создания экземпляра. Противоположный 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 завершающих действий (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.

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 луч-

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 Python это поведение может отличаться).

ше выполнять завершающие действия в явно вызываемом методе (или в инструкции `try/finally`, которая описывается в следующей части книги) – в некоторых случаях в системных таблицах могут сохраняться ссылки на ваши объекты, что будет препятствовать вызову деструктора.

Я привел достаточно много примеров использования перегрузки операторов. 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. Далее в книге вы увидите еще несколько примеров, но полный охват этой темы я оставляю за другими источниками информации.

Пространства имен: окончание истории

Теперь, когда мы уже исследовали объекты классов и экземпляров, повествование о пространствах имен в языке Python можно считать завершенным. Для справки я напомним здесь все правила, используемые при разрешении имен. Первое, что вам нужно запомнить: составные и простые имена интерпретируются по-разному, и некоторые области видимости служат для инициализации пространств имен объектов:

- Простые имена (например, `X`) располагаются в областях видимости.
- Составные имена атрибутов (например, `object.X`) принадлежат объектам пространств имен.
- Некоторые области видимости инициализируют пространства имен объектов (в модулях и классах).

Простые имена: глобальные, пока не выполняется присваивание

Поиск неполных простых имен выполняется в соответствии с правилом лексической видимости **LEGB**, выведенном для функций в главе 16:

Присваивание (`X = value`)

Операция присваивания делает имена локальными: создает или изменяет имя `X` в текущей локальной области видимости, если имя не объявлено глобальным.

Ссылка (`X`)

Пытается отыскать имя `X` в текущей локальной области видимости, затем в области видимости каждой из вменяющих функций, затем в текущей глобальной области видимости и, наконец, во встроенной области видимости.

Имена атрибутов: пространства имен объектов

Полные (составные) имена атрибутов ссылаются на атрибуты конкретных объектов и к ним применяются правила, предназначенные для

модулей и классов. Для объектов классов и экземпляров эти правила дополняются включением процедуры поиска в дереве наследования:

Присваивание (`object.X = value`)

Создает или изменяет атрибут с именем *X* в пространстве имен объекта `object`, 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 дереве наследования. В случае объектов, которые создаются не из классов, таких 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, касающиеся 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)

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), локальной переменной в функции (22), атрибута класса (33), локальной переменной в методе (44) и атрибута экземпляра (55). Все пять переменных имеют одинаковые 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 собраны 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 значения 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 f.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 вне соответствующих инструкций `def`. Локальные переменные видимы только программному коду внутри инструкции `def` и существуют в памяти только во время выполнения функции или метода.

Некоторые из имен, определяемых этим файлом, видимы и за пределами файла, в других модулях, но не забывайте, что мы всегда должны сначала выполнять операцию импортирования, чтобы получить доступ к именам в другом файле – 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 определяется местоположением инструкции присваивания в программном коде (то есть лексически) и не зависит от того, что импортируется и куда импортируется. 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.

Конечно, вы не должны использовать одно и то же имя для обозначения всех переменных в своем сценарии! Но этот пример демонстрирует, что если даже 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 19 мы узнали, что пространства имен модулей фактически реализуются как словари и доступны в виде встроенного атрибута `__dict__`. То же относится к объектам классов и экземпляров: обращение к составному имени атрибута фактически является операцией доступа к элементу словаря, а механизм наследования атрибута работает лишь как поиск в связанных словарях. Фактически объекты экземпляра и класса – это в значительной степени просто словари со ссылками, ведущими вглубь интерпретатора. Интерпретатор 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 интерактивной оболочки проследим, как растут словари пространств имен, когда в игру вступают классы. Сначала определим суперкласс и подкласс с методами, 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 пустым словарем пространства имен, но имеет ссылку на класс, стоящий выше в дереве наследования. Фактически дерево наследования доступно в виде специальных атрибутов, которые вы можете проверить. В экземплярах имеется атрибут `__class__`, который ссылается на класс, а классы имеют атрибут `__bases__`, который является кортежем, содержащим ссылки на суперклассы выше в дереве наследования:

```
>>> X = sub()
>>> X.__dict__
{ }

>>> X.__class__
<class __main__.sub at 0x00A48448>

>>> sub.__bases__
(<class __main__.super at 0x00A3E1C8>,)

>>> super.__bases__
()
```

Так как в классах выполняется присваивание атрибутам аргумента `self`, тем самым они заполняют объекты экземпляров, то есть атрибуты включаются в словари пространств имен экземпляров, но не классов. В пространство имен объекта экземпляра записываются данные, которые могут отличаться для разных 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__
{'__module__': '__main__', '__doc__': None, 'hola': <function hola at 0x00A47048>}

>>> super.__dict__
{'__module__': '__main__', 'hello': <function hello at 0x00A3C5A8>, '__doc__': None}

>>> sub.__dict__.keys(), super.__dict__.keys()
(['__module__', '__doc__', 'hola'], ['__module__', 'hello', '__doc__'])

>>> Y.__dict__
{ }
```

Обратите внимание на имена в словарях классов, содержащие символы подчеркивания, – эти имена определяются интерпретатором автоматически. Большинство из них обычно не используются в програм-

мах, но существуют такие инструменты, которые используют некоторые из этих имен (например, `__doc__` хранит строки документирования, обсуждавшиеся в главе 14).

Кроме того, обратите внимание, что второй экземпляр `Y`, созданный в начале сеанса, по-прежнему имеет пустой словарь пространства имен несмотря на то, что словарь экземпляра `X` заполнялся инструкциями присваивания в методах. Напомню еще раз, что у каждого экземпляра имеется свой, независимый словарь, который изначально пуст и может быть заполнен совершенно другими атрибутами, 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 составным 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 составному имени также вызывает запуск процедуры поиска в дереве наследования, такой способ может обеспечить доступ к атрибутам, которые нельзя получить индексированием 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 14, к объектам 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 версии Python 2.2, функция `dir` также автоматически собирает унаследованные атрибуты:¹

```
>>> X.__dict__
{'data1': 'spam', 'data3': 'ham', 'data2': 'eggs'}
```

¹ Содержимое словарей атрибутов и результаты вызова функции `dir` могут отличаться. Например, т.к. теперь интерпретатор позволяет встроенным типам классифицировать себя как классы, для встроенных типов функция `dir` включает информацию о методах перегрузки операторов. Вообще, имена атрибутов, начинающиеся и завершающиеся двумя символами подчеркивания, являются особенностью интерпретатора. Подклассы типов будут рассматриваться в главе 26.


```
>>> X.__dict__.keys()
['data1', 'data3', 'data2']

>>> dir(X)
['__doc__', '__module__', 'data1', 'data2', 'data3', 'hello', 'hola']
>>> dir(sub)
['__doc__', '__module__', 'hello', 'hola']
>>> dir(super)
['__doc__', '__module__', '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 программах, понимание того, что пространства имен — это всего лишь обычные словари, поможет лишить покрова таинственности само понятие пространств имен.

Ссылки на пространства имен

В предыдущем разделе были представлены атрибуты экземпляра и класса `__class__` и `__bases__`, но не объяснялось, зачем они могут понадобиться. В двух словах, эти атрибуты позволяют осматривать иерархии наследования в вашем программном коде. Например, их можно использовать для отображения дерева классов на экране, как в следующем примере:

```
# classtree.py

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)       # Войти 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 классов произвольной формы — в процессе рекур-

сии 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 дерево классов, создаст в нем два экземпляра и выведет структуры классов, соответствующие им:

```
% python classtree.py
Tree of <__main__.B instance at 0x00ACB438>
... B
..... A
Tree of <__main__.F instance at 0x00AC4DA8>
... F
..... D
..... B
..... A
..... C
..... A
..... E
```

Отступы, отмеченные точками, обозначают высоту в дереве классов. Конечно, мы могли бы улучшить формат вывода и даже отобразить дерево в графическом интерфейсе.

Мы можем импортировать эти функции везде, где нам может потребоваться быстро отобразить дерево классов:

```
>>> class Emp: pass
...
>>> class Person(Emp): pass
...
>>> bob = Person()
>>> import classtree
>>> classtree.instancetree(bob)
Tree of <__main__.Person instance at 0x00AD34E8>
... Person
..... Emp
```

Независимо от того, будете вы создавать и использовать нечто подобное в своей практике или нет, этот пример демонстрирует один из многих способов использования специальных атрибутов, которые создаются внутренними механизмами интерпретатора. Еще один пример вы увидите в разделе «Множественное наследование» в главе 25, где будет реализован класс общего назначения для получения списка атрибутов.

Более реалистичный пример

Большая часть примеров, которые мы видели до сих пор, были искусственными и ограниченными и были предназначены, чтобы помочь вам сосредоточить свое внимание на основах. Однако мы закончим эту главу большим примером, в котором воедино собирается многое из то-

го, что мы уже изучили. Я включил этот пример по большей части как упражнение для самостоятельного изучения – попробуйте внимательно прочитать программный код примера и понять, как разрешаются вызовы методов.

Если говорить двух словах, следующий модуль, *person.py*, определяет три класса:

- *GenericDisplay* – это смешанный класс, который предоставляет универсальный метод `__str__`; для любого класса, который будет наследовать его, данный метод возвращает строку, содержащую имя класса, из которого был создан данный экземпляр, а также пары «имя = значение» для всех атрибутов в дереве наследования. Для построения списка пар «имя = значение» он использует атрибут пространства имен `__dict__`. А для определения имени класса используется встроенный атрибут `__name__` атрибута экземпляра `__class__`. Так как инструкция `print` вызывает метод `__str__`, при выводе всех экземпляров, наследующих поведение этого класса, будет использован собственный формат вывода. Это общий инструмент.
- *Person* хранит общую информацию о человеке и предоставляет два метода, которые используются для изменения информации о состоянии объекта. Кроме того, от своего суперкласса он наследует логику вывода в нестандартном формате. Экземпляр класса *Person* имеет два атрибута и два метода, управляющие этим классом.
- *Employee* – это адаптация класса *Person*, которая наследует извлечение отчества и вывод в нестандартном формате, а также добавляет новый метод, с помощью которого выполняется увеличение зарплаты служащего, и переопределяет операцию изменения возраста (очевидно, служащие стареют быстрее, чем другие люди). Обратите внимание, что конструктор суперкласса вызывается вручную – нам необходимо выполнить версию суперкласса, чтобы заполнить атрибуты `name` и `age`.

По мере изучения программного кода этого модуля вы увидите, что каждый экземпляр имеет свою собственную информацию о состоянии. Обратите внимание, как используется наследование для адаптации поведения и как используется перегрузка операторов для инициализации и вывода экземпляров:

```
# person.py

class GenericDisplay:
    def gatherAttrs(self):
        attrs = '\n'
        for key in self.__dict__:
            attrs += '\t%s=%s\n' % (key, self.__dict__[key])
        return attrs
    def __str__(self):
        return '<%= %s>' % (self.__class__.__name__, self.gatherAttrs())

class Person(GenericDisplay):
```

```

def __init__(self, name, age):
    self.name = name
    self.age = age
def lastName(self):
    return self.name.split()[-1]
def birthDay(self):
    self.age += 1

class Employee(Person):
    def __init__(self, name, age, job=None, pay=0):
        Person.__init__(self, name, age)
        self.job = job
        self.pay = pay
    def birthDay(self):
        self.age += 2
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)

if __name__ == '__main__':
    bob = Person('Bob Smith', 40)
    print bob
    print bob.lastName()
    bob.birthDay()
    print bob

    sue = Employee('Sue Jones', 44, job='dev', pay=100000)
    print sue
    print sue.lastName()
    sue.birthDay()
    sue.giveRaise(.10)
    print sue

```

Чтобы проверить этот программный код, его можно импортировать и создать экземпляры в интерактивной оболочке. Ниже приводится пример класса `Person` в действии. Создание экземпляра вызывает метод `__init__`, вызов именованных методов использует или изменяет информацию о состоянии экземпляра (атрибутов), а попытка вывести экземпляр вызывает унаследованный метод `__str__`, что обеспечивает единообразный формат вывода всех атрибутов:

```

>>> from person import Person
>>> ann = Person('Ann Smith', 45)
>>> ann.lastName()
'Smith'
>>> ann.birthDay()
>>> ann.age
46
>>> print ann
<Person:
  age=46
  name=Ann Smith
>

```

Наконец, ниже приводятся результаты выполнения программного кода самопроверки (в нижней части файла, вслед за проверкой атрибута `__name__`), который создает экземпляры классов `Person` и `Employee` и изменяет каждый из них. Как обычно, программный код самопроверки выполняется, только если файл запускается как самостоятельный сценарий, а не импортируется как библиотечный модуль. Обратите внимание, как класс `Employee` наследует реализацию операции форматированного вывода и извлечения отчества, добавляет дополнительную информацию о состоянии, добавляет метод для получения надбавки и адаптирует версию метода увеличения возраста (сотрудники стараются в два раза быстрее!):

```
% python person.py
<Person:
  age=40
  name=Bob Smith
>
Smith
<Person:
  age=41
  name=Bob Smith
>
<Employee:
  job=dev
  pay=100000
  age=44
  name=Sue Jones
>
Jones
<Employee:
  job=dev
  pay=110000.0
  age=46
  name=Sue Jones
>
```

Проследите, как работает программный код в этом примере, чтобы понять, как эти результаты отражают вызовы методов, – в этом примере сведена воедино большая часть идей, составляющих основу ООП.

Теперь, когда вы познакомились с классами в языке Python, вы наверняка оцените тот факт, что использованные здесь классы – это немногим более, чем пакеты функций, которые внедряют и манипулируют встроенными объектами, присоединенными к атрибутам экземпляра в качестве информации о состоянии. Например, когда метод `lastName` разбивает строку и извлекает последнюю часть, он просто применяет встроенные операции обработки строки и списка к объекту, управляемому классом.

Перегрузка операторов и наследование – автоматический поиск атрибутов в подразумеваемом дереве классов – это главные механизмы

ООП. В конечном итоге именно они позволяют классу `Employee`, находящемуся в самом низу дерева, получать большую часть своего поведения «просто так», что составляет главную идею ООП.

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 пространствах имен в языке Python, расширив это понятие, чтобы охватить его применение к классам. По пути мы рассмотрели еще несколько дополнительных концепций, таких как абстрактные суперклассы, атрибуты данных класса и вызов методов и конструкторов суперкласса вручную. В заключение мы изучили большой пример, соединяющий в себе многое из того, что мы уже знаем об ООП.

Теперь, когда мы знаем все о программировании классов в языке Python, в следующей главе мы обратимся к некоторым распространенным шаблонам проектирования – способам использования и комбинирования классов для оптимизации многократного использования программного кода. Часть сведений в этой главе напрямую не связана с языком Python, но ее необходимо знать для грамотной работы с классами. Однако, прежде чем приступить к чтению, ответьте на обычные контрольные вопросы, чтобы освежить в памяти все, о чем говорилось в этой главе.

Закрепление пройденного

Контрольные вопросы

1. Что такое абстрактный суперкласс?
2. Какие два метода перегрузки операторов можно использовать для поддержки итераций в классах?
3. Что произойдет, когда простая инструкция присваивания появится на верхнем уровне в инструкции `class`?
4. Зачем может потребоваться в классе вручную вызывать метод `__init__` суперкласса?
5. Как можно расширить унаследованный метод вместо полного его замещения?
6. Какие методы вызываются в последнем примере этой главы, когда выводится экземпляр `sue` класса `Employee`?
7. Назовите... столицу Ассирии.

Ответы

1. Абстрактный суперкласс – это класс, который вызывает методы, но не наследует и не определяет их. Он ожидает, что методы будут реа-

лизированы в подклассах. Часто такой прием используется для обобщения классов, когда поведение будущих подклассов трудно предсказать заранее. Платформы ООП также используют этот прием для выполнения операций, определяемых клиентом.

2. Классы могут обеспечить поддержку итераций, определив (или унаследовав) метод `__getitem__` или `__iter__`. Во всех итерационных контекстах интерпретатор Python сначала пытается использовать метод `__iter__` (который 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 метода извлечения элемента по его индексу `__getitem__` (который вызывается многократно и при каждом вызове получает постоянно увеличивающиеся значения индексов).
3. 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 методами.
4. Вручную вызывать метод `__init__` суперкласса может потребоваться, когда класс определяет свой собственный конструктор `__init__` и при этом необходимо, чтобы выполнялись действия, предусмотренные конструктором суперкласса. Интерпретатор Python автоматически вызывает только один конструктор – самый нижний в дереве наследования. Конструктор суперкласса вызывается через имя класса, и ему вручную передается аргумент `self`: `Superclass.__init__(self, ...)`.
5. 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, ...)`.
6. В конечном итоге при выводе экземпляра `sue` будет вызван метод `GenericDisplay.__str__`, который в свою очередь вызовет метод `GenericDisplay.gatherAttrs`. Подробнее, при печати экземпляра `sue` инструкция `print` преобразует информацию в объекте в удобочитаемую строку, передавая объект встроенной функции `str`. Для класса это означает попытку отыскать в дереве наследования метод перегрузки оператора `__str__` и, в случае успеха, его вызов. Объект `sue` является экземпляром класса `Employee`, который сам не имеет метода `__str__`, поэтому следующий цикл поиска производится в классе `Person` и метод `__str__` будет найден в конце концов в классе `GenericDisplay`.
7. Ашшур (или Калат-Шеркат), Калах (или Нимруд), короткое время был столицей Дур-Шаррукин (или Хорсабад) и, наконец, Ниневия.

25

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 моделирования полезных объектов. В этой главе мы коснемся некоторых базовых идей ООП и рассмотрим несколько дополнительных примеров, более реалистичных, чем мы видели до сих пор. Многие термины, упоминающиеся здесь (делегирование, композиция, фабрики и другие), 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

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 соглашение 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 полиморфизме в языке Python – он произрастает из отсутст-

вия объявления типов в языке Python. Поскольку разрешение имен атрибутов производится на этапе выполнения, объекты, реализующие одинаковые интерфейсы, являются взаимозаменяемыми – клиентам не требуется знать тип объекта, реализующего вызываемый метод.

Инкапсуляция в языке Python означает упаковывание – то есть сокрытие подробностей реализации за интерфейсом объекта. Это не означает принудительное сокрытие, как будет показано в главе 26. Инкапсуляция позволяет изменять реализацию интерфейсов объекта, не оказывая влияния на пользователей этого объекта.

Использование перегрузки путем сигнатур вызова (или отказ от нее)

В некоторых объектно-ориентированных языках под полиморфизмом также понимается возможность перегрузки функций, основанной на сигнатурах типов их аргументов. Но, так как в языке Python отсутствуют объявления типов, эта концепция в действительности здесь не применима – полиморфизм в языке Python основан на *интерфейсах* объектов, а не на типах.

Вы можете попробовать выполнить перегрузку методов, изменяя списки их аргументов, 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 имени в области видимости класса, сохранено будет только последнее определение метода (это все равно, что записать 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 16:

```
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 15, следует писать такой код, который полагается 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 появятся в будущем:

```
class C:
    def meth(self, x):
        x.operation()    # Предполагается, что x работает правильно
```

Кроме того, считается, что лучше выбирать разные имена для методов, выполняющих разные операции, и не полагаться на сигнатуры вызова (при этом неважно, какой язык программирования вы используете).

Классы как записи

В главе 8 демонстрировалось, как использовать словари для хранения записей свойств сущностей в программах. Рассмотрим этот прием более подробно. Ниже приводится пример использования записи на основе словаря, использовавшийся ранее:

```
>>> rec = {}
>>> rec['name'] = 'mel'
>>> 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 видели в главе 23, существует еще множество способов сделать то же самое с помощью классов. Ниже приводится, пожалуй, самый простой из них:

```
>>> class rec: pass
...
>>> rec.name = 'mel'
>>> 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 инструкция `class` (обратите внимание на инструкцию `pass` — этого требует синтаксис, так как в данном случае класс не содержит никакой логики, которую необходимо было бы реализовать). 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 класса всякий раз, когда нам потребуется новая запись:

```
>>> class rec: pass
...
>>> pers1 = rec()
>>> 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 класса начинают свое существование пустыми, как и классы. После этого производится 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 обязательно должны быть одинаковые наборы имен атрибутов. В данном примере один из экземпляров имеет уникальный атрибут `age`. Экземпляры класса действительно являются разными пространствами имен: каждый из них имеет свой словарь атрибутов. Обычно экземпляры единообразно наполняются атрибутами 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):
...         self.name = name
...         self.job = job
...     def info(self):
...         return (self.name, self.job)
...
>>> mark = Person('ml', 'trainer')
>>> dave = Person('da', 'developer')
>>>
>>> mark.job, dave.info()
('trainer', ('da', '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 налагает некоторые ограничения целостности, требуя значения для двух атрибутов – `name` и `job`.

Мы могли бы продолжить расширение этой реализации, добавляя методы для вычисления зарплаты, разбора имен и т. д. (как это делается, описывается в примере в конце главы 24). В конечном итоге мы могли бы связать класс в иерархию, чтобы обеспечить наследование набора существующих методов через процедуру автоматического поиска атрибутов классов, и даже записывать экземпляры класса в файл, чтобы

обеспечить их постоянное хранение (подробнее о сохранении объектов рассказывается во врезке «Придется держать в уме: классы и их хранение», ниже в этой книге). Наконец, несмотря на всю гибкость таких типов, как словари, классы позволяют наделять объекты поведением таким способом, который встроенными типами и простыми функциями напрямую не поддерживается.

ООП и наследование: взаимосвязи типа «является»

Мы уже достаточно подробно исследовали механизм наследования, но мне хотелось бы показать пример того, как может использоваться модель отношений реального мира. С точки зрения программиста, наследование вступает в игру с момента появления составного имени атрибута, при разрешении которого запускается поиск имен в экземплярах, в их классах и затем в суперклассах. С точки зрения проектировщика, наследование – это способ указать принадлежность к некоторому набору: класс определяет набор свойств, которые могут быть унаследованы и адаптированы более специализированными наборами (то есть подклассами).

Чтобы проиллюстрировать сказанное, давайте вернемся к машине по изготовлению пиццы, о которой мы говорили в начале этой части книги. Предположим, 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 т. д. 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 поведение, общее для всех работников, такое как повышение заработной платы (*giveRaise*) и вывод на экран (*__repr__*). Существует два типа 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 специализированным классом *PizzaRobot*, наследующим класс *Chef*, который в свою очередь наследует класс *Employee*. В терминах ООП мы называем такие взаимоотношения «является»: робот является поваром, а повар – 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"
    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 модуля, создается робот по приготовлению пиццы с именем bob, который наследует атрибуты трех классов: PizzaRobot, Chef и Employee. Например, при попытке вывести экземпляр bob вызывается метод Employee.__repr__, а прибавка роботу зарплаты производится методом Employee.giveRaise, потому что этот метод обнаруживается 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 любого 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 настоящую работу.

ООП и композиция: взаимосвязи типа «имеет»

Понятие композиции в этой книге было введено в главе 22. С точки зрения программиста, композиция – это прием встраивания других объектов в объект-контейнер и использование их для реализации методов контейнера. Для проектировщика композиция – это один из способов представить взаимоотношения 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, посвященных объектно-ориентированному проектированию, композиция называется *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» используется лишь для обозначения коллекции встраиваемых объектов. Вообще составные классы реализуют все свои интерфейсы, управляя работой встраиваемых 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 начинают действовать – официант принимает заказ, повар делает пиццу и т. д. Следующий пример (файл *pizza-shop.py*) имитирует все объекты и взаимоотношения 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
    scene.order('Homer')                # Имитировать заказ клиента Homer
    print '...'
    scene.order('Shaggy')               # Имитировать заказ клиента Shaggy

```

Класс PizzaShop – это контейнер и контроллер – это конструктор, который создает и встраивает экземпляры классов работников, написанные нами в предыдущем 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 выполнению своих обязанностей. Обратите внимание, что для каждого клиента мы создаем новый экземпляр класса `Customer` и передаем встроенный объект `Server` (официант) методом класса `Customer` (клиент) – клиенты приходят и уходят, а официант остается частью коллектива пиццерии. Кроме того, обратите внимание, 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 существительные* классами, *глаголы* – методами, и вы получите первый черновой набросок проекта.

Еще раз об обработке потоков

Рассмотрим более реалистичный пример использования приема композиции. Вспомните универсальную функцию обработки потоков данных, которая частично была реализована во введении в ООП в главе 22:

```
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 возможных реализаций этого класса содержится в файле *streams.py* и приводится в ниже:

```
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 0, 'converter must be defined'
```

При таком подходе объекты чтения и записи встраиваются в экземпляр класса (*композиция*), а логика преобразования поставляется в виде подкласса, а не в виде отдельной функции (*наследование*). 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
    Uppercase(open('spam.txt'), sys.stdout).process()
```

Здесь класс *Uppercase* наследует логику цикла обработки потока данных (и все остальное, что может присутствовать в суперклассах). В нем необходимо определить лишь то, что будет уникальным для него – логику преобразования данных. Если запустить этот файл, он создаст и запустит экземпляр класса *Uppercase*, который прочитает содержимое файла *spam.txt*, преобразует все символы в верхний регистр и выведет их в поток *stdout*:


```
C:\lp3e> type spam.txt
spam
Spam
SPAM!

C:\lp3e> python converters.py
SPAM
SPAM
SPAM!
```

Для обработки потоков различных видов достаточно передать конструктору класса объекты требуемых типов. Ниже приводится пример реализации вывода в файл вместо стандартного потока вывода:

```
C:\lp3e> python
>>> import converters
>>> prog = converters.Uppercase(open('spam.txt'), open('spamup.txt', 'w'))
>>> prog.process()

C:\lp3e> type spamup.txt
SPAM
SPAM
SPAM!
```

Но, как предлагалось ранее, мы могли бы также реализовать объекты, обернутые 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:\lp3e> python
>>> from converters import Uppercase
>>>
>>> class HTMLize:
...     def write(self, line):
...         print '<PRE>%s</PRE>' % line[:-1]
...
>>> 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 верхнему 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` предоставляет только цикл скачивания файла. Для выполнения более существенных действий его можно было бы расширить, чтобы обеспечить поддержку дополнительных инструментов в его подклассах и постепенно превратить все это в полноценную платформу. Создав такой инструмент один раз, вы

Придется держать в уме: классы и их хранение

В этой части книги я уже несколько раз упоминал о возможности сохранения объектов, потому что этот метод особенно хорошо работает с экземплярами классов. Например, помимо возможности имитировать взаимодействия в реальном мире, классы, разработанные для пиццерии, могли бы также использоваться как основная база данных пиццерии. Экземпляры классов могут сохраняться на диск за одно действие – с помощью модулей `pickle` или `shelve`. Интерфейс модуля `pickle` очень прост 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 последовательные потоки байтов, которые можно сохранять в файлах, передавать по сети и т. д. При извлечении объектов происходит обратное преобразование: из последовательного потока байтов в идентичные объекты в памяти. Модуль `shelve` реализует похожую возможность, но он автоматически сохраняет объекты в базе данных 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 работников означает, что можно достаточно легко создать простую базу данных сотрудников и пиццерий: записывая экземпляры объектов в файл, мы сможем сохранять их между запусками программы. Более подробную информацию о сохранении ищите в руководстве к стандартной библиотеке.

сможете многократно использовать его во всех своих программах. Даже в этом простом примере благодаря тому, что с помощью классов можно упаковать и унаследовать так много, все, что нам пришлось 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 26, а его решение – в приложении В. Он напоминает пример с пиццерией. 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 взаимодополняющие (а иногда и альтернативные) приемы.

Композиция – это проблема проектирования, которая далеко выходит за рамки языка Python и этой книги, поэтому полный охват этой темы я оставляю за другими источниками информации.

ООП и делегирование

В ООП часто используется термин *делегирование*, под которым обычно подразумевается наличие объекта-контроллера, куда встраиваются другие объекты, получающие запросы на выполнение операций. Контроллеры могут решать административные задачи, такие как слежение за попытками доступа и т. д. 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 `__getattr__` для перенаправления обращений к обернутому объекту. Класс-обертка имеет 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 24 говорилось, что метод `__getattr__` получает имя атрибута в виде строки. В этом примере для извлечения из обернутого объекта атрибута, имя которого 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 с выражени-

ем `X.__dict__[N]`, только в первом случае дополнительно выполняется поиск в дереве наследования, как в выражении `X.N`, а во втором – нет (подробнее об атрибуте `__dict__` рассказывается в разделе «Словари пространств имен» в главе 24).

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 дополнительно или адаптированному программному коду и т. д.

В главе 26 мы еще вернемся к обернутым объектам и делегированию операций как к одному из способов расширения встроенных типов. Если шаблон проектирования с делегированием заинтересовал вас, тогда смотрите обсуждение *декораторов функций* в главе 26 – это очень близкая концепция, призванная обеспечить расширение отдельных функций и методов, а не всего интерфейса объекта.

Множественное наследование

В строке заголовка инструкции `class` в круглых скобках может быть перечислено более одного суперкласса. В этом случае используется то, что называется *множественным наследованием*, – класс и его экземпляры наследуют имена 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 поиск сначала продолжается по направлению снизу вверх всеми возможными путями, вплоть до вершины дерева наследования, а затем слева направо, потому что любой суперкласс может иметь собственные суперклассы.

Вообще множественное наследование хорошо использовать для моделирования объектов, принадлежащих более чем одной группе. Например,

человек может быть инженером, писателем, музыкантом и т. д., и наследовать свойства всех этих групп.

Пожалуй, самый распространенный случай, где используется множественное наследование, – это «смешивание» методов общего назначения из нескольких суперклассов. Обычно такие суперклассы называются *классами-смесями* – они предоставляют методы, которые добавляются в прикладные классы наследованием. Например, способ вывода экземпляра класса, используемый в Python по умолчанию, не отличается информативностью:

```
>>> class Spam:
...     def __init__(self):          # Нет метода __repr__
...         self.data1 = "food"
...
>>> X = Spam()
>>> print X                        # По умолчанию: класс, адрес
<__main__.Spam instance at 0x00864818>
```

Как было отмечено в предыдущем разделе, где рассматривалась перегрузка оператора, существует возможность с помощью метода `__repr__` реализовать свою собственную операцию вывода. Но вместо того чтобы воспроизводить 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 классы-смеси. В следующем файле *mytools.py* определяется класс-смесь с именем `Lister`, переопределяющий метод `__repr__` в каждом классе, который наследует его. Он просто сканирует словарь атрибутов экземпляра (экспортируется в виде атрибута `__dict__`) и собирает строку из имен и значений всех атрибутов экземпляра. Так как классы – это объекты, логика форматирования класса `Lister` может использоваться и для экземпляров любых подклассов – это универсальный инструмент.¹

Класс `Lister` использует два специальных приема, чтобы получить из экземпляра имя класса и адрес. Каждый экземпляр имеет встроенный атрибут `__class__`, указывающий на класс, из которого он был создан, и каждый класс имеет атрибут `__name__`, указывающий на имя в заголовке, поэтому выражение `self.__class__.__name__` возвращает имя класса экземпляра. Получить адрес экземпляра в памяти можно с помощью встроенной функции `id`, способной возвращать адрес любого объекта (по определению, уникальный идентификатор объекта):

¹ Альтернативный способ реализации похожего поведения можно найти в примере модуля *person.py* в конце главы 24. В нем также просматриваются словари пространств имен, но там не предполагается пропуск имен, начинающихся и заканчивающихся двумя символами подчеркивания.

```
#####
# Lister можно подмешивать в любые классы, чтобы обеспечить форматированный
# вывод экземпляров через наследование метода __repr__, реализованного здесь;
# self - это экземпляр самого нижнего класса.
#####

class Lister:
    def __repr__(self):
        return ("<Instance of %s, address %s:\n%s>" %
                (self.__class__.__name__, # Имя класса
                 id(self), # Адрес
                 self.attrnames()) ) # Список имя=значение

    def attrnames(self):
        result = ''
        for attr in self.__dict__.keys(): # Словарь имен экземпляра
            if attr[:2] == '__':
                result = result + "\tname %s=<built-in>\n" % attr
            else:
                result = result + "\tname %s=%s\n" % (attr, self.__dict__[attr])
        return result
```

Экземпляры, унаследовавшие этот класс, при выводе будут автоматически отображать свои атрибуты, давая о себе больше информации, чем просто свой адрес:

```
>>> from mytools import Lister
>>> class Spam(Lister):
...     def __init__(self):
...         self.data1 = 'food'
...
>>> x = Spam()
>>> x
<Instance of Spam, address 8821568:
    name data1=food
>
```

Класс Lister будет полезен для наследования любыми вашими классами, у которых уже есть суперкласс. Это тот случай, где удобно использовать множественное наследование: добавляя класс Lister в список суперклассов в строке заголовка класса (вмешивая его), вы получаете в свое распоряжение его метод __repr__, что не мешает наследовать другой существующий суперкласс. Это демонстрирует файл *testmixin.py*:

```
from mytools import Lister # Получить класс с инструментами

class Super:
    def __init__(self): # Метод суперкласса __init__
        self.data1 = "spam"

class Sub(Super, Lister): # Наследовать __repr__ из класса-смеси
    def __init__(self): # Lister имеет доступ к self
        Super.__init__(self)
        self.data2 = "eggs" # Дополнительные атрибуты экземпляра
        self.data3 = 42
```

```

if __name__ == "__main__":
    X = Sub()
    print X                                # __repr__ из класса-смеси

```

Здесь класс `Sub` наследует имена из двух классов, `Super` и `Lister` – этот объект состоит из своих собственных имен и из имен обоих суперклассов. Если создать и вывести экземпляр класса `Sub`, автоматически будет получено адаптированное представление, подмешанное из класса `Lister`:

```

C:\lp3e> python testmixin.py
<Instance of Sub, address 7833392:
    name data3=42
    name data2=eggs
    name data1=spam
>

```

Метод класса `Lister` будет работать в любом классе, унаследовавшем его, потому что аргумент `self` ссылается на экземпляр подкласса, который наследует `Lister`, каким бы этот подкласс ни был. Если позднее вы решите усовершенствовать класс `Lister`, чтобы метод `__repr__` выводил также все атрибуты класса, которые были унаследованы экземпляром, вы без опаски сможете сделать это – так как это наследуемый метод, изменение в реализации `Lister.__repr__` автоматически начнет действовать и во всех подклассах, которые импортируют его.¹

В некотором смысле, классы-смеси – это классы, эквивалентные модулям, потому что они упаковывают методы, которые будут полезны самым разным клиентам. Ниже демонстрируется работа класса `Lister` в режиме единственного наследования с экземплярами различных классов:

```

>>> from mytools import Lister
>>> class x(Lister):
...     pass
...
>>> t = x()

```

¹ Если вам любопытно, как это произойдет, вернитесь назад к разделу «Словари пространств имен» в главе 24. Там мы видели, что у каждого класса имеется свой атрибут `__bases__`, который является кортежем объектов суперклассов. Универсальный класс просмотра иерархии может выполнять обход дерева наследования от экземпляра класса `__class__` к его классу и затем с помощью атрибута `__bases__` ко всем его суперклассам рекурсивно, как это делается в модуле `classtree.py`, показанном ранее. В версии Python 2.2 и более поздних реализовать это можно еще проще, поскольку теперь встроенная функция `dir` автоматически включает имена унаследованных атрибутов. Если не требуется отображать структуру дерева наследования, то можно просто сканировать список, получаемый от функции `dir`, вместо просмотра списка ключей словаря и использовать функцию `getattr` для получения значения атрибута по его имени, заданному в виде строки, вместо обхода словаря. Мы еще вернемся к этой идее в одном из заключительных примеров к этой части.

```
>>> t.a = 1; t.b = 2; t.c = 3
>>> t
<Instance of x, address 7797696:
      name b=2
      name a=1
      name c=3
>
```

ООП тесно связано с повторным использованием программного кода, и классы-смеси 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.

Классы – это объекты: фабрики универсальных объектов

Классы – это объекты, поэтому их легко можно передавать между компонентами программы, сохранять в структурах данных и т. д. Можно также передавать 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 тривиальной. Функция `apply` и новейшая синтаксическая конструкция, с которыми мы познакомились в главе 17, могут вызывать любые классы с любым числом аргументов конструкторов за один присест, генерируя экземпляр любого типа:¹

```
def factory(aClass, *args):      # Кортеж переменного числа аргументов
    return apply(aClass, args)  # Вызывайте aClass, или: aClass(*args)

class Spam:
    def doit(self, message):
        print message

class Person:
    def __init__(self, name, job):
        self.name = name

self.job = job
```

¹ Фактически функция `apply` 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 (независимо от имени аргумента). Кроме того, обратите внимание: в последних версиях Python синтаксис вызова `aClass(*args)` вообще предпочтительнее, чем вызов встроенной функции `apply(aClass, args)`.


```
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 аргументов конструктора класса. Она создает экземпляр с помощью функции `apply` и возвращает его.

Остальная часть примера просто определяет два класса и генерирует экземпляры этих классов, передавая 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 заключается в обеспечении поддержки именованных аргументов конструктора; фабричная функция может собрать их в аргумент `**args` и передать функции `apply` в виде третьего аргумента:

```
def factory(aClass, *args, **kwargs): # +kwargs
    return apply(aClass, args, kwargs) # Вызвать aClass
```

К настоящему времени вы должны знать, что в языке Python все является «объектом», включая и сами классы, которые в других языках, таких как C++, являются лишь объявлениями для компилятора. Однако, как упоминалось в начале шестой части книги, в языке Python только объекты, созданные из классов, являются субъектами ООП.

Зачем нужны фабрики?

Итак, чем же хороша функция `factory` (помимо иллюстрации того, что классы 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 программный код от динамически настраиваемой конструкции объекта.

Вспомним пример функции `processor`, представленный в главе 22, и затем пример применения принципа композиции в этой главе. В обоих случаях принимаются объекты, выполняющие чтение и запись обрабатываемого потока данных.

В оригинальной версии этого примера мы вручную передавали экземпляры 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 конфигурационных файлах или в элементах управления графического интерфейса.

В таком динамическом мире не представляется возможным жестко задавать в сценарии объекты, реализующие интерфейс к потоку дан-

ных, но вполне возможно создавать их во время выполнения, в соответствии с содержимым конфигурационных файлов.

Например, в файле с настройками может определяться имя класса потока, который должен быть импортирован из модуля, и дополнительные аргументы 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 программе. 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
reader = factory(aclass, classarg)                # Получить экземпляр aclass(classarg)
processor(reader, ...)
```

Здесь встроенная функция `getattr` снова используется для извлечения атрибута модуля, имя которого задано в виде строки (это все равно, что записать выражение `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 неизвестно заранее, то есть когда универсальная фабричная функция способна повысить гибкость реализации. За дополнительной информацией по этой теме обращайтесь к книгам, которые посвящены вопросам ООП и шаблонам проектирования.

Методы – это объекты: связанные и несвязанные методы

Методы – это разновидность объектов, напоминающая функции. Доступ к методам класса осуществляется через экземпляр класса или через сам класс и, следовательно, в языке Python имеется две разновидности методов:

Несвязанные методы класса: без аргумента `self`

Попытка обращения к функциональному атрибуту класса через имя класса возвращает объект несвязанного метода. Чтобы вызвать этот метод, необходимо явно передать ему объект экземпляра в виде первого аргумента.

Связанные методы экземпляра: пара `self` + функция

Попытка обращения к функциональному атрибуту класса через имя экземпляра возвращает объект связанного метода. Интерпре-

татор автоматически упаковывает экземпляр с функцией в связанный объект метода, поэтому вам не требуется передавать экземпляр в вызов метода.

Обе разновидности методов – это полноценные объекты. Они могут передаваться между программными компонентами, сохраняться в списках и т. д. При запуске оба требуют наличия экземпляра в первом аргументе (то есть значения для аргумента `self`). Именно по этой причине в предыдущей главе было необходимо явно передавать экземпляр при вызове методов суперкласса из методов подкласса – с технической точки зрения такие вызовы порождают объекты несвязанных методов.

Вызывая объект связанного метода, интерпретатор автоматически подставляет экземпляр, который использовался при создании объекта связанного метода. Это означает, что объекты связанных методов обычно взаимозаменяемы с объектами простых функций и создание их особенно полезно в случае интерфейсов, изначально ориентированных на использование функций (реалистичный пример приводится во врезке «Придется держать в уме: связанные методы и функции обратного вызова»).

Чтобы проиллюстрировать вышесказанное, предположим, что имеется следующее 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 связанный метод и без его вызова. Составное имя `object.name` – это выражение, 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 другому имени и затем использовать это имя для вызова, как простую функцию:

```
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 экземпляр 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
```

Чаще всего вы будете вызывать методы немедленно, сразу же после указания составного имени, поэтому вы не всегда будете замечать, что попутно создается объект метода. Но как только вы начнете писать программный код, который вызывает объекты единообразным способом, то сразу обратите внимание на несвязанные методы, потому что обычно они требуют явной передачи экземпляра в первом аргументе.¹

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.

Еще раз о строках документирования

Строки документирования, которые мы подробно рассматривали в главе 14, — это литералы строк, которые присутствуют на верхнем уровне различных структур и автоматически сохраняются интерпретатором Python в атрибутах `__doc__` соответствующих им объектов. Строки документирования могут присутствовать в модулях, в инструкциях `def`, а также в определениях классов и методов. Теперь, когда мы ближе познакомились с классами и методами, можно изучить 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__"

class spam:
    "I am: spam.__doc__ or docstr.spam.__doc__"
    def method(self, arg):
        "I am: spam.method.__doc__ or self.method.__doc__"
```

¹ Смотрите обсуждение статических методов класса в главе 26, 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.

Придется держать в уме: связанные методы и функции обратного вызова

В объектах связанных методов вместе с функцией автоматически сохраняется экземпляр класса, поэтому они могут использоваться везде, где используются обычные функции. Одно из обычных мест, где можно увидеть эту идею в действии, – это программный код, регистрирующий методы как обработчики событий в интерфейсе Tkinter GUI. 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, который 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 – это объект связанного метода, где сохраняются `self` и `MyWidget.handler`. Так как аргумент `self` ссылается на оригинальный экземпляр, то когда метод `handler` будет вызван для обработки события, он получит доступ к атрибутам экземпляра, где может сохраняться информация о состоянии между событиями. При использовании обычных функций для этих целей, как правило, используются глобальные переменные. Другой способ обеспечения совместимости классов с прикладным интерфейсом, основанным на применении функций, приводится в главе 24, где обсуждается метод перегрузки операторов `__call__`.

```
pass

def func(args):
    "I am: docstr.func.__doc__"
    pass
```

Основное преимущество строк документирования состоит в том, что их содержимое доступно во время выполнения. То есть, если текст был оформлен в виде строки документирования, можно будет обратиться к атрибуту `__doc__` объекта, чтобы получить его описание:

```
>>> import docstr
>>> docstr.__doc__
'I am: docstr.__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__'

>>> docstr.func.__doc__
'I am: docstr.func.__doc__'
```

В главе 14 также обсуждается PyDoc – инструмент, который позволяет формировать отчеты из всех этих строк.

Строки документирования доступны во время выполнения, но синтаксически они менее гибки, чем комментарии # (которые могут находиться в любом месте программы). Обе формы несут пользу, и любая документация к программе – это хорошо (при условии, что она точная).

Классы и модули

Мы завершаем эту главу кратким сравнением предметов обсуждения двух последних частей книги: модулей и классов. Так как оба представляют собой 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, недоступные в модулях, такие как перегрузка операторов, создание множества экземпляров и наследование. Несмотря на то, что и классы, и модули являются пространствами имен, к настоящему времени вы должны четко понимать, что между ними имеются существенные различия.

В заключение

В этой главе мы рассмотрели подборку типичных способов комбинирования классов для получения наибольшей пользы от их повторного ис-

пользования 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 способен облегчить их решение). Мы изучили приемы делегирования (обертывание объектов в классы-обертки), композиции (управление встраиваемыми объектами), наследования (приобретение поведения от других классов) и некоторые другие не совсем обычные концепции, такие как 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 программы, но эти сведения все же заслуживают того, чтобы с ними ознакомились 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?

Ответы

1. Множественное наследование имеет место, когда класс наследует более одного суперкласса, – это удобно для объединения пакетов программного кода, оформленных в виде классов.
2. Делегирование подразумевает обертывание объекта классом-оберткой, который расширяет функциональные возможности обернутого объекта и передает ему выполнение части операций. Класс-обертка сохраняет интерфейс 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, потому что внутри таких методов по-прежнему доступен оригинальный экземпляр.

26

Дополнительные возможности классов

Эта глава завершает шестую часть книги и наше изучение ООП на языке Python представлением нескольких более сложных тем, связанных с использованием классов: мы рассмотрим возможность создания подклассов встроенных типов, псевдочастные (pseudoprivate) атрибуты, классы нового стиля, статические функции, декораторы функций и многое другое.

Как мы уже знаем, модель ООП в языке Python чрезвычайно проста, а некоторые из приемов, представленных в этой главе, настолько сложны и совершенно необязательны к использованию, что вы, возможно, не слишком часто 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 типичных проблем, с которыми сталкиваются программисты при использовании классов, и набором упражнений к этой части. Я советую вам обязательно проработать эти упражнения, чтобы прочнее ухватить идеи, которые мы изучали в этой части. Я также предлагаю вам самостоятельно познакомиться с крупными объектно-ориентированными проектами на языке Python в дополнение к этой книге. Как и все в программировании, преимущества ООП становятся более очевидными с обретением опыта.

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. Например, чтобы добавить в списки дополнитель-

ные методы вставки и удаления, можно создать класс, который оберты- вает (встраивает) объект списка и экспортирует методы вставки и уда- ления, которые особым образом обрабатывают список, подобно тому, как реализуется прием делегирования, рассмотренный в главе 25. На- чиная с версии Python 2.2 для специализации встроенных типов мож- но также использовать наследование. Следующие два раздела демон- стрируют оба приема в действии.

Расширение типов встраиванием

Помните те функции для работы со множествами, которые мы написа- ли в четвертой части книги? Ниже показано, как они выглядят, реани- мированные в виде класса на языке Python. Следующий пример (файл *setwrapper.py*) реализует новый тип объектов за счет перемещения не- нескольких функций в методы и добавления перегрузки нескольких ос- новных операторов. По большей части этот класс просто оберты- вает список, добавляя дополнительные операции. Поскольку это класс, он также поддерживает возможность создания множества экземпляров и адаптацию своего поведения наследованием в подклассах:

```
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:' + 'self.data'     # Вывод
```

Перегрузка операции доступа к элементам по их индексам позволяет экземплярам нашего класса Set выглядеть как настоящие списки. В упражнениях в конце этой главы вам будет предложено организо-

вать взаимодействие с этим классом и расширить его, поэтому подробнее об этом фрагменте мы поговорим в приложении В.

Расширение типов наследованием

Начиная с версии Python 2.2 все встроенные типы можно наследовать. Функции преобразования типов, такие как `list`, `str`, `dict` и `tuple`, превратились в имена встроенных типов. Теперь вызов функции преобразования типа (например, `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 новыми именами типов, где реализовать необходимые изменения. Экземпляры вашего нового подкласса могут использоваться везде, где допускается использовать оригинальный встроенный тип. Например, предположим, вас не устраивает тот факт, что стандартные списки начинают отсчет элементов с 0, а не с 1. Это не проблема – вы всегда можете создать свой подкласс, который изменяет 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 списка

    x.reverse(); print x
```

В этом файле подкласс `MyList` расширяет метод `__getitem__` встроенных списков простым отображением диапазона значений от 1 до N на необходимый список 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 достаточно для достижения поставленной цели:

```
% python typesubclass.py
['a', 'b', 'c']
['a', 'b', 'c']
(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 выполнении индексирования. Является ли изменение способа индексирования универсально хорошей идеей, это уже другой вопрос: пользователи вашего класса `MyList` могут быть повергнуты в недоумение таким отступлением от общепринятого поведения последовательностей в языке Python. Однако возможность адаптировать встроенные типы подобным образом может оказаться мощным инструментом.

Например, такой шаблон дает начало новому способу реализации множеств – в виде подкласса встроенного списка, а не в виде самостоятельного класса, который управляет встроенным в него объектом списка. Следующий пример реализации класса в файле *setsubclass.py* адаптирует списки, добавляя методы и операторы, используемые для работы с множествами. Все остальное поведение наследуется от 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])
    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 самопроверки, находящегося в конце файла. Поскольку проблема наследования встроенных типов достаточно сложна, я опущу дальнейшие подробности, но я 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]
```

Есть и более эффективные способы реализации множеств – с помощью словарей, позволяющих заменить последовательное сканирование, используемое в данной реализации, на операцию обращения по ключу (хеширование) и тем самым повысить скорость работы. (За дополнительной информацией обращайтесь к книге «Programming Python».) Если вас заинтересовали множества, тогда вам также стоит взглянуть на тип объектов `set`, который рассматривался в главе 5, – этот встроенный тип реализует операции 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.

Псевдочастные атрибуты класса

В четвертой части книги мы узнали, что все имена, для которых выполняется присваивание на верхнем уровне модуля, становятся глобальными для этого модуля. То же самое 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 терминах языка C++, – они доступны отовсюду и динамически отыскиваются во время выполнения.¹

¹ Это излишне пугает программистов, работающих с языком C++. В языке Python возможно 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 24, где говорилось, что методы `__getattr__` и `__setattr__` могут использоваться для имитации поведения частных атрибутов, но на практике эта возможность обычно не используется.

На сегодняшний день все это пока так и есть. Однако Python поддерживает такое понятие, как «искажение» («mangling») имен (то есть расширение), с целью придать им черты локальных имен класса. Искраженные имена иногда ошибочно называют частными атрибутами, но в действительности это всего лишь способ *ограничить* доступ к именам в классе – искажение имен не предотвращает доступ из программного кода, находящегося за пределами класса. Эта особенность в основном предназначена, чтобы избежать конфликтов имен в экземплярах, а не для ограничения доступа к именам – поэтому искаженные имена лучше называть «псевдочастными», чем «частными».

Псевдочастные имена – это совершенно необязательная возможность и вы, скорее всего, не найдете в ней особого смысла, пока не начнете писать крупные иерархии классов в проектах, над которыми трудятся несколько программистов. Но так как использование этой возможности может встретиться в программном коде других программистов, вам следует знать о ней, даже если вы сами не используете ее.

Об искажении имен в общих чертах

Здесь описывается, как действует искажение имен: имена внутри инструкции `class`, которые начинаются с двух символов подчеркивания, но не заканчиваются двумя символами подчеркивания, автоматически расширяются за счет включения имени вмещающего класса. Например, такое имя как `__X` в классе с именем `Spam` автоматически изменится на `_Spam__X`: к оригинальному имени будет добавлен префикс, состоящий из символа подчеркивания и имени вмещающего класса, и в результате будет получено достаточно уникальное имя, которое не будет вступать в конфликт с именами в других классах иерархии.

Искажение имен происходит только внутри инструкций `class` и только для имен, которые начинаются двумя символами подчеркивания. Однако это происходит со *всеми* именами, которые начинаются двумя символами подчеркивания, 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 атрибуты более чем из одного класса, такое искажение позволяет избежать конфликтов, 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 сохранения 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 присваивание 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 предполагал, что экземпляры этого класса будут владеть атрибутом `X`. В методах класса выполняется присваивание этому атрибуту и позднее извлекается его значение:

```
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 присваивания атрибуту `self.X` будут воздействовать на один и тот же экземпляр, у которого может быть только один атрибут `X` — `I.X` — независимо от того, сколько классов используют это имя.

Чтобы гарантировать принадлежность атрибута тому классу, который его использует, достаточно в начале имени атрибута поставить два символа подчеркивания 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()                                # У меня 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, можно увидеть 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 атрибута внутри класса, вы сможете обратиться к нему из любой точки программы, где имеется ссылка на экземпляр, используя для этого расширенное имя (например, `I._C1__X = 77`). С другой стороны, эта особенность делает менее вероятным, что вы *случайно* вступите в конфликт с существующими именами в классе.

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 необходимо обеспечить принадлежность атрибута единственному классу. Для простых программ этот прием будет излишеством.



Кроме того, посмотрите, как имитируется сокрытие атрибутов в главе 24, в разделе с описанием метода `__getattr__`. Несмотря на то, что в языке Python вполне возможно организовать управление доступом к атрибутам классов, тем не менее, этот прием редко используется на практике, даже в крупных программах.

Классы нового стиля

В версии Python 2.2 появилась новая разновидность классов, известная как классы «нового стиля». Классы, рассматривавшиеся до сих пор в этой части книги, называют «классическими классами», когда сравнивают их с новой разновидностью.

Классы нового стиля лишь немного отличаются от классических классов, и эти отличия совершенно незаметны для подавляющего большинства пользователей Python. Кроме того, классическая модель классов, используемая в языке Python на протяжении последних 15 лет, по-прежнему работает именно так, как было описано выше.

Классы нового стиля практически сохраняют обратную совместимость с классическими классами в синтаксисе и в поведении – они привносят лишь несколько новых особенностей. Однако, т.к. они изменили один особый случай наследования, их следует представлять как отдельный инструмент, чтобы избежать нежелательных воздействий на любой существующий программный код, работоспособность которого зависит от прежнего поведения классов.

Классы нового стиля создаются с помощью привычной синтаксической конструкции, которую мы уже изучили. Главное отличие состоит в том, что новые классы объявляются как подклассы встроенных типов (например, `list`). Если ни один из встроенных типов не подходит для наследования, тогда новые классы в качестве суперкласса должны указывать новое встроенное имя `object`:

```
class newstyle(object):  
    ...обычная реализация класса...
```

Если говорить более широко, любой класс, наследующий `object` или любой другой встроенный тип, автоматически будет рассматриваться как класс нового стиля. (Под *наследованием* я подразумеваю все подклассы `object`, подклассы подклассов `object`, и т. д. – если где-то в дереве наследования в качестве суперкласса присутствует встроенный тип, класс-наследник будет рассматриваться как класс нового стиля.) Классы, не наследующие встроенные типы, считаются классическими.



Согласно утверждениям Гвидо ван Россума (Guido van Rossum), создателя языка Python, начиная с версии Python 3.0 все классы автоматически будут считаться классами нового стиля, поэтому требование наследования встроенных типов отпадет автоматически. Даже самостоятельные классы будут считаться классами нового стиля, к тому же классы нового стиля сохраняют обратную совместимость с классическими классами, поэтому для большинства программистов эти изменения будут малозаметны.

В прошлом имелось некоторое беспокойство, что в Python 3.0 классы верхнего уровня необходимо будет наследовать от `object`. Однако недавно Гвидо (Guido) заявил, что этого не потребуется. Для большинства программистов все классы в версии 3.0 будут работать так, как описывается в этой книге, но с некоторыми новыми особенностями, свойственными классам нового стиля. Тем не менее, я не берусь предсказывать будущее, поэтому за дополнительной информацией обращайтесь к примечаниям к выпуску 3.0.

Ромбоидальное наследование

Пожалуй, самым ощутимым изменением в классах нового стиля является немного отличная интерпретация наследования – так называемая *ромбоидальная* схема в деревьях множественного наследования, когда более одного суперкласса наследует один и тот же суперкласс более высокого уровня. Ромбоидальная схема – это сложная концепция про-

ектирования, которую мы даже не обсуждали при изучении обычных классов.

Проще говоря, в классической модели процедура поиска в дереве наследования сначала движется строго вверх по дереву, а потом слева направо – сначала интерпретатор поднимается вверх всеми возможными путями по левой стороне 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. Вследствие такого изменения суперклассы, расположенные ниже, получают возможность переопределять атрибуты суперклассов, стоящих выше, независимо от вида деревьев множественного наследования.

Пример ромбоидального наследования

В качестве иллюстрации рассмотрим следующую упрощенную реализацию ромбоидального наследования для классических классов:

```
>>> class A: attr = 1          # Классическая модель
>>> class B(A): pass
>>> class C(A): attr = 2
>>> class D(B,C): pass        # Сначала поиск дойдет до A, потом до C
>>> x = D()
>>> x.attr
1
```

В этом случае атрибут `attr` будет найден в суперклассе `A`, потому что в классической модели поиск в дереве наследования сначала производится в высоту, и только потом происходит смещение вправо – интерпретатор выполнит поиск в классах в следующем порядке: `D`, `B`, `A` и затем `C` (впрочем, поиск прекратится, как только атрибут `attr` будет найден в суперклассе `A`, расположенном выше суперкласса `B`). В классах нового стиля, наследующих встроенный тип, такой как `object`, прежде чем просмотреть суперкласс `A`, интерпретатор сначала выполнит поиск в суперклассе `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): attr = 1   # Новый стиль
>>> class B(A): pass
>>> class C(A): attr = 2
>>> class D(B,C): pass         # Сначала поиск дойдет до C, потом до A
>>> x = D()
>>> x.attr
2
```

Это изменение процедуры поиска основано на предположении, что если вы добавляете класс `C` в дерево ниже, это значит, что вы хотите улучшить его атрибуты 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 неверно, когда в ромбоидальной схеме принимают участие классические классы – вы можете даже не подозревать, что класс `C` может участвовать в подобной схеме наследования, когда пишете его.

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 запоминания или вам требуется более полное управление процедурой поиска, вы всегда можете произвести выбор желаемого атрибута из любого места в дереве, выполнив присваивание или как-то иначе обозначив его там, где может возникнуть смешение классов:

```
>>> class A: attr = 1                # Классическая модель
>>> class B(A): pass
>>> class C(A): attr = 2
>>> class D(B,C): attr = C.attr      # Выбрать C, справа
>>> x = D()
>>> x.attr                           # Работает как класс нового стиля
2
```

Здесь дерево классических классов имитирует порядок поиска, принятый в модели классов нового стиля: присваивание атрибуту `attr` в классе `D` явно выбирает 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): attr = 1        # Новый стиль
>>> class B(A): pass
>>> class C(A): attr = 2
>>> class D(B,C): attr = B.attr      # Выбрать A.attr, выше
>>> x = D()
>>> x.attr                           # Работает как классический класс
1
```

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 методам, 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 используется классический порядок поиска
A.meth

>>> class D(B,C): meth = C.meth # Выбрать метод класса C: новый стиль
>>> x = D()
>>> x.meth()
C.meth

>>> class D(B,C): meth = B.meth # Выбрать метод класса B:
                                # классическая модель
>>> x = D()
>>> x.meth()
A.meth

```

Здесь мы явно выбираем методы, выполняя присваивание именам, находящимся ниже в дереве. Мы могли бы просто вызвать метод желаемого класса явно; на практике этот подход, возможно, является более принятым, 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 эффективно 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 зависеть от версии Python в будущем (независимо от необходимости наследовать встроенные типы, чтобы использовать новую модель).¹

Итак, поиск в ромбоидальной схеме наследования выполняется по-разному в классической и в новой моделях, и это изменение нарушает обратную совместимость. Однако имейте в виду, что это изменение за-

¹ Даже если не учитывать расхождения между классической и новой моделями, данная методика иногда может пригодиться в случаях множественного наследования. Если вам необходимо получить часть атрибутов от суперкласса слева, а часть — от суперкласса справа, вам может потребоваться указать интерпретатору Python, какие именно атрибуты следует выбирать, выполняя явное присваивание в подклассах. Мы еще вернемся к этому вопросу в разделе с описанием типичных проблем в конце этой главы. Кроме того, обратите внимание, что ромбоидальные схемы наследования 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 необходимо вызывать оба наследуемых конструктора в подклассе?), но это лежит за пределами рассмотрения данной книги.

трагивает только ромбоидальные схемы наследования – во всех других схемах принцип действия модели наследования нового стиля не изменился. Кроме того, вполне возможно, что вся эта проблема будет носить скорее теоретический характер, чем практический, – с выходом версии Python 2.2 эти изменения не оказали достаточно существенного влияния, и потому маловероятно, что они затронут значительную часть программного кода на языке Python.

Другие расширения в классах нового стиля

Помимо изменения порядка поиска в ромбоидальной схеме (что само по себе слишком неясно, чтобы иметь большое значение для большинства читателей этой книги) классы нового стиля привносят несколько еще более сложных возможностей. Ниже приводится краткий обзор каждой из них.

Статические методы и методы класса

Начиная с версии Python 2.2 появилась возможность определять методы класса, 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. Чтобы сделать возможными эти режимы работы методов, внутри класса должны вызываться специальные встроенные функции `staticmethod` и `classmethod`. Несмотря на то, что эта особенность была добавлена вместе с классами нового стиля, ее можно использовать и в классических классах тоже. Вследствие этого отложим обсуждение данной темы до следующего раздела.

Слоты экземпляров

Присваивая список имен атрибутов в виде строк специальному атрибуту `__slots__` класса, в классах нового стиля можно ограничить множество разрешенных атрибутов для экземпляров класса. Обычно этот атрибут устанавливается присваиванием 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, прежде чем получить доступ к атрибутам экземпляра им должны быть присвоены значения, даже если они перечислены в списке `__slots__`. Например:

```
>>> 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 слотах
AttributeError: 'limiter' object has no attribute 'ape'
(AttributeError: объект 'limiter' не имеет атрибута 'ape')
```

Эта особенность должна помочь ликвидировать ошибки, обусловленные простыми «опечатками» (обнаруживается попытка присваивания атрибутам, отсутствующим в списке `__slots__`) и обеспечить некоторую оптимизацию (атрибуты слота могут сохраняться не в словаре, а в кортеже, обеспечивая тем самым более высокую скорость поиска).

Однако слоты – это своего рода нарушение динамической природы языка Python, которая диктует, что операция присваивания может создавать любые имена. Кроме того, у них имеются дополнительные ограничения и следствия, которые слишком сложны, чтобы обсуждать их здесь. Например, некоторые экземпляры со слотами могут не иметь атрибут словаря `__dict__`, что может сделать некоторые метапрограммы, которые мы создавали в этой книге, более сложными, например, инструментам, выполняющим поиск атрибутов, скорее всего придется искать имена в двух источниках вместо одного. За дополнительной информацией обращайтесь к документации к выпуску Python 2.2 и стандартному набору руководств по языку Python.

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 24. Свойства обладают тем же эффектом, что и эти два метода, только в этом случае выполняется вызов метода 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 обсуждать ее здесь.

Проще говоря, свойства – это тип объектов, который присваивается именам атрибутов класса. Они создаются вызовом встроенной функции `property` с тремя методами (обработчиками операций получения, присваивания и удаления) и строкой документирования – если в каком-либо аргументе передается значение `None`, следовательно, эта операция не поддерживается. Определение свойств обычно производится на верхнем уровне в инструкции `class` (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                # Запустит метод __getattr__
40
>>> x.name              # Запустит метод __getattr__
AttributeError

```

Ниже тот же пример, но уже с использованием свойств:

```

>>> class newprops(object):
...     def getage(self):
...         return 40
...     age = property(getage, None, None, None) # get, set, del, docs
...
>>> x = newprops()
>>> x.age                # Запустит 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 привлекательными – программный код выглядит компактнее и в операцию присваивания 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                # Запустит метод getage
40
>>> x.age = 42           # Запустит метод setage
set age: 42
>>> x._age              # Нормальная операция извлечения; нет вызова getage
42
>>> 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 производить лишние вызовы метода и, возможно, выполнять присваивание значения атрибуту с использованием словаря, чтобы избежать заикливания:

```
>>> 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                                     # Запустит метод __getattr__
40
>>> x.age = 41                               # Запустит 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 многих случаях невозможно заранее определить набор поддерживаемых атрибутов, которые могут даже не существовать вообще в каком-либо виде на момент написания класса (например, при делегировании ссылок на произвольные методы в обернутых/встроенных объектах). В таких случаях использование более универсальных методов обслуживания атрибутов `__getattr__` и `__setattr__`, которым передается имена атрибутов, может оказаться предпочтительнее. Кроме того, простейшие ситуации могут обслуживаться этими обработчиками, поэтому свойства следует рассматривать как дополнительное и необязательное к использованию расширение.

Новый метод перегрузки `__getattribute__`

Метод `__getattribute__` имеется только в классах нового стиля и позволяет классам перехватывать *все* попытки обращения к атрибутам, а не только к неопределенным (как метод `__getattr__`). Кроме того, этот метод более сложен в обращении, чем `__getattr__` и `__setattr__` (из-за более высокой вероятности заикливания). Полное описание этого метода я оставляю за стандартной документацией по языку Python.

Помимо всех этих особенностей классы нового стиля интегрируются с понятием расширения типов, упоминавшимся выше в этой главе, – возможность расширения типов и классы нового стиля были введены вместе с разделением тип/класс в версии Python 2.2 и выше.

Поскольку особенности классов нового стиля – это достаточно сложные темы, мы не будем углубляться в подробности в этой книге. За дополнительной информацией обращайтесь к документации к выпуску Python 2.2 и справочным руководствам по языку.

Статические методы и методы класса

До выхода версии Python 2.2 методы класса никогда нельзя было вызывать без передачи экземпляра в качестве аргумента. В Python 2.2 и более поздних версиях такое поведение также используется по умолчанию, но существует возможность изменить такое положение дел с помощью новой дополнительной особенности, получившей название *статические методы* – простые функции без аргумента `self`, вложенные в определение класса и предназначенные для работы с атрибутами класса, а не экземпляра. Такие методы обычно используются для обработки информации, которая имеет отношение ко всем экземплярам (например, число созданных экземпляров), а не для реализации поведения экземпляров.

В предыдущей главе мы говорили о *несвязанных* методах: когда мы извлекаем функцию по имени класса (а не экземпляра), мы получаем объект несвязанного метода. Хотя объекты несвязанных методов определяются с помощью инструкции `def`, они не являются простыми функциями – они не могут вызываться без передачи им экземпляра класса.

Например, предположим, что необходимо использовать атрибуты класса для подсчета числа экземпляров, созданных из класса (как показано в следующем файле *spam.py*). Не забывайте, что атрибуты класса совместно используются всеми экземплярами, поэтому мы можем хранить счетчик непосредственно в объекте класса:

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances():
        print "Number of instances created: ", Spam.numInstances
```

Но такая реализация не будет работать – метод `printNumInstances` по-прежнему ожидает получить экземпляр при вызове, потому что функция ассоциирована с классом (даже при том, что в заголовке инструкции `def` отсутствуют какие-либо аргументы):

```
>>> from spam import *
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
```



```
>>> Spam.printNumInstances()
Traceback (innermost last):
File "<stdin>", line 1, in ?
TypeError: unbound method must be called with class instance 1st argument
(TypeError: несвязанный метод должен вызываться с экземпляром класса
в 1-м аргументе)
```

Проблема состоит в том, что несвязанные методы экземпляра – это не то же самое, что простые функции. Эта проблема главным образом является проблемой знания особенностей языка. Самая простая мысль, которая приходит в голову, – сделать метод обычной функцией, а не методом класса. При таком способе функции не требуется передавать экземпляр класса:

```
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()
Number of instances created: 3
>>> spam.Spam.numInstances
3
```

Поскольку имя класса доступно простой функции в виде глобальной переменной, 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.

Мы могли бы сделать то же самое, вызывая функцию через экземпляр, как обычно, хотя это не очень удобно, особенно если создание экземпляра приводит к изменениям в данных класса:

```
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
>>> b.printNumInstances()
Number of instances created: 3
```

```
>>> Spam().printNumInstances()
Number of instances created: 4
```

До появления статических методов в Python 2.2 некоторые теоретики языка утверждали, что в языке Python отсутствуют методы класса – только методы экземпляра. Я полагаю, в действительности они имели в виду, что классы в языке Python работают совсем не так, как в других языках программирования. Что в действительности имеется в языке Python, так это связанные и несвязанные методы с четкой семантикой – при обращении к методу через имя класса вы получаете несвязанный метод, который представляет собой особую разновидность функций. В языке Python имеются атрибуты класса, но функции в классе ожидают получить экземпляр в виде аргумента.

Кроме того, в языке Python уже имеются модули, которые играют роль инструмента разделения пространства имен, поэтому обычно не возникает необходимости упаковывать функции в классы, если они не реализуют функциональность объектов. Простые функции внутри модуля обычно способны решать большую часть задач, которые возлагаются на методы класса. Например, в первом фрагменте в этом разделе функция `printNumInstances` уже связана с классом, потому что располагается в том же самом модуле. Единственный недостаток этой функции состоит в том, что она имеет более широкую область видимости – весь модуль, а не класс.

Использование статических методов и методов класса

На сегодняшний день существует еще одна возможность писать простые функции, связанные с классом. Начиная с версии Python 2.2 имеется возможность создавать классы со статическими методами и с методами класса, ни один из которых не требует передачи экземпляра класса в виде аргумента. Чтобы определить такие методы, в классах необходимо вызывать встроенные функции `staticmethod` и `classmethod`, как упоминалось в обсуждении классов нового стиля. Например:

```
class Multi:
    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 статичным методом
    cmeth = classmethod(cmeth)  # Сделать cmeth методом класса.
```

Обратите внимание, как две последние операции присваивания в этом фрагменте просто *переприсваивают* имена методов `smeth` и `cmeth`. Атрибуты создаются и изменяются с помощью операции присваивания в инструкции `class`, поэтому эти заключительные операции присваивания переопределяют инструкции `def`, выполненные ранее.

С технической точки зрения, язык Python теперь поддерживает три разновидности методов классов: методы экземпляра, статические методы и методы 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 экземпляр, интерпретатор автоматически передает экземпляр в первом аргументе – когда метод вызывается через имя класса, экземпляр необходимо передавать вручную:

```
>>> obj = Multi()           # Создать экземпляр
>>> obj.imeth(1)            # Обычный вызов, через экземпляр
__main__.Multi instance...> 1
>>> Multi.imeth(obj, 2)     # Обычный вызов, через класс
__main__.Multi instance...> 2
```

Статические методы, напротив, вызываются без аргумента с экземпляром класса – их имена ограничены областью видимости класса, в котором они определяются и могут отыскиваться механизмом наследования. Главным образом они действуют как обычные функции, которые просто находятся внутри класса:

```
>>> Multi.smeth(3)         # Вызов статического метода, через имя класса
3
>>> obj.smeth(4)           # Вызов статического метода, через экземпляр
4
```

Методы класса похожи на них, но интерпретатор автоматически передает методам класса сам класс (а не экземпляр) в первом аргументе:

```
>>> Multi.cmeth(5)         # Вызов метода класса, через имя класса
__main__.Multi 5
>>> obj.cmeth(6)           # Вызов метода класса, через экземпляр
__main__.Multi 6
```

Статические методы и методы класса являются новыми дополнительными особенностями языка с узко специализированным назначением, для описания которого у нас недостаточно места в этой книге. Статические методы обычно используются для работы с атрибутами класса и управления информацией, которая касается всех экземпляров, созданных из класса. Например, чтобы вести подсчет количества созданных экземпляров класса (как в примере выше), можно было бы использовать статический метод, управляющий счетчиком, присоединенным к классу в виде атрибута. Такой счетчик не имеет ничего общего ни с одним из экземпляров, поэтому было бы неудобно иметь методы, которые обслуживают его, вызываемые через экземпляр (тем более, что создание экземпляра для доступа к счетчику может изменить этот счетчик). Кроме того, близость статических методов к классу обеспечивает более естественное решение, чем ориентированные на класс функции за пределами класса.

Ниже приводится реализация статического метода, эквивалентного оригинальному примеру этого раздела:

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances():
        print "Number of instances:", Spam.numInstances
    printNumInstances = staticmethod(printNumInstances)

>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()
Number of instances: 3
>>> a.printNumInstances()
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) и программный код перемещен туда, где он используется (внутри инструкции `class`). Судите сами, является ли это чистым усовершенствованием или нет.

В последних версиях Python определение статических методов выполняется еще проще; следующий раздел описывает – как.

Декораторы функций

Прием с вызовом функции `staticmethod`, описанный в предыдущем разделе, выглядит малопонятным для некоторых пользователей, поэтому была добавлена возможность, упрощающая эту операцию. *Декораторы функций* обеспечивают способ определять специальные режимы работы для функций, обертывая их дополнительным слоем логики, реализованной в виде других функций.

Декораторы функций представляют собой более универсальные инструменты: их удобно использовать для добавления самой разной логики к функциям, помимо статических методов. Например, их можно использовать для расширения функций программным кодом, выполняющим регистрацию вызовов этих функций, проверяющим типы передаваемых аргументов в процессе отладки и т. д. В некоторой степени декораторы функций напоминают шаблон проектирования *делегирования*, исследованный нами в главе 25, но их главная цель состоит в том, чтобы расширять определенные функции или методы, а не весь интерфейс объекта.

Язык Python предоставляет несколько встроенных декораторов функций для выполнения таких действий, как создание статических методов, но программисты также имеют возможность создавать свои собственные декораторы. Несмотря на то, что они строго не привязаны

к классам, тем не менее, пользовательские декораторы функций часто оформляются как классы, в которых сохраняется оригинальная функция наряду с другими данными, такими как информация о состоянии.

Синтаксически декоратор функции – это разновидность объявления функции времени выполнения. Декоратор функции записывается в строке непосредственно перед строкой с инструкцией `def`, которая определяет функцию или метод, и состоит из символа `@`, за которым следует то, что называется *метафункцией*, – 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
    def meth():
        ...
```

С технической точки зрения, это объявление имеет тот же эффект, что и фрагмент ниже (передача функции декоратору и присваивание результата первоначальному имени функции):

```
class C:
    def meth():
        ...
    meth = staticmethod(meth) # Повторное присваивание имени
```

В результате вызов метода по имени функции фактически будет приводить к вызову результата, полученному от декоратора `staticmethod`. Декоратор может возвращать объекты любого типа, поэтому данный прием позволяет 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 оригинальная функция, переданная декоратору, которая будет вызываться косвенно после того, как будет выполнен дополнительный слой логики.

Синтаксис декораторов поддерживает добавление нескольких слоев оберты логики к декорируемой функции или методу. В этом случае строка с декоратором выглядит следующим образом:

```
@A @B @C
def f():
    ...
```

что эквивалентно следующему фрагменту:

```
def f():
    ...
f = A(B(C(f)))
```

Здесь оригинальная функция проходит через три различных декоратора, а результат присваивается оригинальному имени. Напомню еще

раз, что в результате всего этого, когда происходит обращение к оригинальному имени функции, вызываются три слоя логики, дополняющие оригинальную функцию.

Пример декоратора

Ниже приводится пример декоратора, определяемого пользователем. Вспомните, как в главе 24 говорилось, что метод перегрузки оператора `__call__` реализует интерфейс вызова функций в экземплярах классов. В следующем примере этот метод используется в определении класса, который сохраняет декорируемую функцию в экземпляре и перехватывает вызовы по оригинальному 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
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 вызов к оригинальному имени `spam`, в действительности вызывается метод `__call__` в классе. Этот метод подсчитывает и регистрирует вызовы, а затем вызывает оригинальную обернутую функцию. Обратите внимание, как используется синтаксис аргумента `*name` для упаковки аргументов, передаваемых функции, – благодаря этому данный декоратор может использоваться для обертывания любой функции с любым числом 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.

Декораторы функций являют собой универсальный механизм, и, тем не менее, эта дополнительная особенность представляет интерес в первую очередь для разработчиков инструментальных средств, а не для прикладных программистов, поэтому я снова отсылаю вас за более подробной информацией к стандартному набору руководств по языку Python.

Типичные проблемы при работе с классами

Большая часть типичных проблем, связанных с классами, сводится к проблемам, связанным с пространствами имен (особенно если учесть, что классы – это всего лишь пространства имен с некоторыми дополнительными особенностями). Некоторые темы, которые мы затронем в этом разделе, скорее являются 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 *изменяемых* 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 состояние), но, изменяя атрибуты, об этом необходимо помнить. Все экземпляры класса совместно используют одно и то же пространство имен класса, поэтому 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 помощью операций присваивания. Рассмотрим следующий класс. В теле класса выполняется присваивание имени `a`, в результате чего создается атрибут `X.a`, который во время выполнения располагаетс^я в объекте класса и будет унаследован 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 происходит, когда атрибут класса изменяется динамически, из-за его пределов: 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 динамически установленное значение независимо от того, что написано в исходном программном коде класса:

```
>>> X.a = 2      # Может измениться не только в классе X
>>> I.a          # Я 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. Фактически 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 работают 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 классам, написанным другими программистами, – вы не всегда можете быть уверены, 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'
```

Множественное наследование: порядок имеет значение

Это достаточно очевидно, но тем не менее, стоит подчеркнуть: в случае использования множественного наследования порядок, в котором пе-

речислены суперклассы в строке заголовка инструкции `class`, может иметь критическое значение. В ходе поиска интерпретатор всегда просматривает суперклассы слева направо в соответствии с порядком их следования в заголовке инструкции.

Например, в примере множественного наследования, который был продемонстрирован в главе 25, предположим, что класс `Super` тоже реализует метод `__repr__`. От какого класса мы унаследовали бы метод – от класса `Lister` или `Super`? Это зависело бы от того, какой класс стоит первым в заголовке объявления класса `Sub`, так как поиск унаследованных атрибутов производится слева направо. Очевидно, мы поставили бы класс `Lister` первым в списке, потому что его основная цель состоит в предоставлении метода `__repr__`:

```
class Lister:
    def __repr__(self): ...

class Super:
    def __repr__(self): ...

class Sub(Lister, Super): # Будет унаследован метод __repr__ класса
                        # Lister, так как он стоит в списке первым
```

А теперь предположим, что классы `Super` и `Lister` имеют свои собственные версии еще одного одноименного атрибута. Если необходимо, чтобы одно имя наследовалось от класса `Super`, а другое от класса `Lister`, изменение порядка их расположения в заголовке инструкции определения подкласса уже не поможет – мы должны вручную переопределить результат наследования, явно выполнив присваивание имени атрибута в классе `Sub`:

```
class Lister:
    def __repr__(self): ...
    def other(self): ...

class Super:
    def __repr__(self): ...
    def other(self): ...

class Sub(Lister, Super): # Унаследует __repr__ класса Lister,
                        # 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 Super/Lister
```

Здесь присваивание атрибуту с именем `other` в классе `Sub` создает атрибут `Sub.other` – ссылку на объект `Super.other`. Поскольку эта ссылка находится ниже в дереве классов, это не позволит механизму наследования выбрать версию атрибута `Lister.other`, который был обнаружен первым при обычных обстоятельствах. Точно так же, если бы класс `Super` стоял первым в списке, то, чтобы атрибут `other` наследовался обыч-

ным образом, нам могло бы потребоваться явно выбрать метод `__repr__` класса `Lister`:

```
class Sub(Super, Lister):      # Получить Super.other по наследованию
    __repr__ = Lister.__repr__ # Явно выбрать Lister.__repr__
```

Множественное наследование – это довольно сложная тема. Даже если вы поняли предыдущий параграф, все равно этот прием лучше использовать осторожно и только в случае крайней необходимости. В противном случае могут возникать 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 предположений об именах, связанных с другими классами в дереве. Псевдочастные атрибуты, которые были изучены нами ранее, могут помочь в локализации имен, на владение которыми опирается класс, и ограничить вероятность появления конфликтов имен в суперклассах, которые вы добавляете в список наследуемых классов. Например, в данном случае класс `Lister` служит только для того, чтобы экспортировать метод `__repr__`, поэтому 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 видимости вложенных функций, но я сохранил это описание исключительно ради истории, для тех из вас, кому приходилось работать с более старыми версиями 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 инструкции `class`. Кроме того, методы, по сути, являются вложенными функциями, поэтому здесь имеют место те же самые проблемы. Похоже, что путаница особенно 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 локальной области видимости функции `generate`. Но в версиях Python, появившихся до версии 2.2, внутри метода `method` имя класса `Spam` недоступно – `method` имеет доступ только к своей локальной области видимости, к области видимости модуля, вмещающей его окружающую функцию `generate`, и к встроенным именам:

```

def generate():
    class Spam:
        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
Traceback (innermost last):
  File "nester.py", line 8, in ?
    generate().method()
  File "nester.py", line 5, in method
    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 локальные области вмещающих функций автоматически видимы для вложенных функций (включая и вложенные методы, как в данном примере). Но он не работал в версиях Python, вышедших до версии 2.2 (некоторые возможные решения приводятся ниже).

Обратите внимание, что даже в версии 2.2 методам недоступна локальная область видимости вмещающего *класса* – им доступны только области видимости вмещающих функций. Именно по этой причине методы должны использовать аргумент `self` с экземпляром или именем 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 несколько способов заставить предыдущий пример работать. Самый простой заключается в том, чтобы переместить имя `Spam` в область видимости вмещающего 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 использования объявления `global` определение класса `Spam` находилось на верхнем уровне модуля. После этого вложенный метод `method`

и функция `generate` будут отыскивать класс `Spam` в глобальной области видимости:

```
def generate():
    return Spam()

class Spam:
    count = 1
    def method(self):
        print Spam.count
generate().method()
```

Определение на верхнем уровне в модуле
Работает: глобальное имя (вещающий модуль)

В действительности такой подход рекомендуется использовать во всех версиях Python – программный код выглядит проще, если в нем отсутствуют вложенные классы и функции.

Если вам требуется нечто более сложное и замысловатое, то можно просто избавиться от ссылки на имя `Spam` в методе `method`, используя специальный атрибут `__class__`, который возвращает класс объекта экземпляра:

```
def generate():
    class Spam:
        count = 1
        def method(self):
            print self.__class__.count
    return Spam()
generate().method()
```

Работает: используется атрибут
для получения класса

«Многослойное обертывание»

При грамотном использовании способность объектно-ориентированного программного кода к многократному использованию поможет существенно снизить затраты времени на его разработку. Однако иногда неправильное использование потенциала абстракции ООП может серьезно осложнить понимание программного кода. Если классы наложены друг на друга слишком глубоко, программный код становится малопонятным – возможно, вам придется изучить 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 такой 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, чтобы понять принцип действия какого-либо участка программного кода, требовалось 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. Обертывание программного кода несколькими слоями классов на грани непостижимости – всегда плохая идея. Абстракция – это основа полиморфизма и инкапсуляции, и при грамотном использовании она может быть весьма эффективным инструментом. Однако вы упростите отладку и сопровождение, если сделаете интерфейсы своих классов интуитивно понятными. Избегайте чрезмерной абстракции и сохраняйте иерархии своих классов короткими и плоскими, если не существует веских причин сделать иначе.

В заключение

В этой главе было представлено несколько расширенных возможностей классов, включая наследование встроенных типов, псевдочастные атрибуты, классы нового стиля, статические методы и декораторы функций. Большинство из них являются необязательными расширениями модели ООП в языке Python, но они могут стать более полезными, когда вы начнете создавать крупные объектно-ориентированные программы.

Это конец части, посвященной классам, поэтому ниже вы найдете обычные в этом случае упражнения – обязательно проработайте их, чтобы получить некоторую практику создания настоящих классов. В следующей главе мы начнем изучение последней базовой темы – исключений. Исключения – это механизм взаимодействий с ошибками и другими ситуациями, возникающими в программном коде. Это относительно несложная тема, но я оставил ее напоследок, 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 дополнительного слоя логики к существующим функциям, который запускается при каждом вызове функции. Они могут использоваться для регистрации вызовов этих функций, проверки типов передаваемых аргументов и т. д. 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 умолчанию будут классами нового стиля.
4. При использовании ромбоидальной схемы наследования в классах нового стиля поиск в дереве наследования выполняется иначе – сначала производится поиск в ширину, а не в высоту. Кроме того, классы нового стиля поддерживают ряд новых дополнительных особенностей, включая свойства и список атрибутов экземпляра `__slots__`.
5. Обычные методы (экземпляра) принимают аргумент `self` (подразумеваемый экземпляр), а статические методы – нет. Статические методы – это простые функции, вложенные в объект класса. Чтобы превратить обычный метод в статический, необходимо передать его специальной встроенной функции, или декоратору, с использованием правил декорирования.
6. Три секунды. (Или, если быть более точным: «И сказал Господь: Допреже всего Пресвятую чеку извлечь долженствует. Опосля же того, сочти до трех, не более и не менее. Три есть цифирь, до коей счесть потребно, и сочтенья твои суть три. До четырех счесть не могли, паче же до двух, опричь токмо коли два предшествует трём. Опяти и речи быть не может. Аще же достигнешь ты цифири три, что есть и пребудет третьею цифирью, брось Пресвятою антиохийскою гранатою твоею во врага твоего, и оный враг, будучи ничтожен пред лицом моим, падёт.»¹).

Упражнения к шестой части

В этих упражнениях вам будет предложено написать несколько классов и поэкспериментировать с существующим программным кодом. Единственная проблема существующего кода состоит в том, что он должен существовать. Чтобы поупражняться с набором классов в упражнении 5, вам нужно либо загрузить файл с исходными текстами из Интернета (читайте Предисловие) или ввести его вручную (он достаточно короткий). Поскольку программы становятся все сложнее, обязательно ознакомьтесь с решениями в конце книги. Решения вы найдете в приложении В в разделе «Часть VI. Классы и ООП».

¹ Цитата из фильма «Monty Python and the Holy Grail». Перевод взят из Википедии. В русском переводе фильм вышел под названием «Монти Пайтон и Священный Грааль». – *Примеч. перев.*

1. *Наследование.* Напишите класс с именем `Adder`, экспортирующий метод `add(self, x, y)`, который выводит сообщение «Not Implemented» («Не реализовано»). Затем определите два подкласса класса `Adder`, которые реализуют метод `add`:

`ListAdder`

С методом `add`, который возвращает результат конкатенации двух списков из аргументов.

`DictAdder`

С методом `add`, который возвращает новый словарь, содержащий элементы из обоих словарей, передаваемых как аргументы (пойдет любое определение сложения).

Поэкспериментируйте с экземплярами всех трех классов в интерактивной оболочке, вызывая их методы `add`.

Теперь расширьте суперкласс `Adder`, добавив сохранение объекта в экземпляре 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, выражение `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?

На практике гораздо проще написать метод `add`, который принимает один действительный аргумент (например, `add(self, y)`) и складывает его с текущими данными экземпляра (например, `self.data + y`). Будет ли в такой реализации больше смысла, чем в реализации, которая принимает два аргумента? Можно ли сказать, что это делает ваши классы более «объектно-ориентированными»?

2. *Перегрузка операторов.* Напишите класс с именем `Mylist`, который «обертывает» списки языка Python: он должен перегружать основные операторы действий над списками, включая `+`, доступ к элементам по индексу, итерации, извлечение среза и такие методы списка, как `append` и `sort`. Полный перечень методов, поддерживаемых списками, вы найдете в справочном руководстве 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 своим классом в интерактивной оболочке. В ходе экспериментов выясните следующее:
 - a. Почему здесь так важно копировать начальное значение?
 - b. Можно ли использовать пустой срез (например, `start[:]`) для копирования начального значения, если им является `Mylist`?
 - c. Существует ли универсальный способ передачи управления унаследованным методам списка?
 - d. Можно ли складывать `Mylist` и обычный список? А список и `Mylist`?

- e. Объект какого типа должны возвращать операции сложения и извлечения среза? А операции извлечения элементов по индексу?
 - f. Если у вас достаточно новая версия Python (2.2 или выше), вы сможете реализовать такого рода класс-обертку, встраивая настоящий список в отдельный класс или наследуя класс `list`. Какой из двух способов проще и почему?
3. *Подклассы.* Напишите подкласс с именем `MylistSub` от класса `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 выводиться сообщение, увеличиваться счетчик вызовов операции сложения и вызываться метод суперкласса. Кроме того, добавьте новый метод, который 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 каждый из этих случаев? (Подсказка: зависит от того, в каком объекте производится присваивание значения счетчика: атрибут класса используется всеми 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 интерактивной оболочке. Что произойдет, если попытаться использовать экземпляр класса в выражении? Попробуйте выполнить над своим классом операции сложения, доступа к элементам по индексу и получения среза.
5. *Объекты множеств.* Поэкспериментируйте с набором классов, описанных в разделе «Расширение типов встраиванием». Выполните команды, которые выполняют следующие операции:
- a. Создайте два множества целых чисел и найдите их пересечение и объединение с помощью операторов `&` и `|`.
 - b. Создайте множество из строки и поэкспериментируйте с извлечением элементов множества по индексу. Какой метод в классе при этом вызывается?
 - c. Попробуйте выполнить итерации через множество, созданное из строки, с помощью цикла `for`. Какой метод вызывается на этот раз?
 - d. Попробуйте найти пересечение и объединение множества, созданного из строки, и простой строки. Возможно ли это?
 - e. Теперь расширьте класс множества наследованием, чтобы подкласс мог обрабатывать произвольное число операндов, исполь-

зую для этого форму аргумента `*args`. (Подсказка: вернитесь к рассмотрению этих алгоритмов в главе 16.) Найдите пересечение и объединение нескольких операндов с помощью вашего подкласса множества. Как можно реализовать вычисление пересечения трех и более множеств, если оператор `&` работает всего с двумя операндами?

- f. Как бы вы реализовали другие операции над списками в классе множества? (Подсказка: метод `__add__` перехватывает операцию конкатенации, 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 24 и в разделе «Множественное наследование» в главе 25 я упоминал, что классы имеют атрибут `__bases__`, который возвращает кортеж объектов суперклассов (тех, что перечислены в круглых скобках в заголовке инструкции `class`). Используя атрибут `__bases__`, расширьте класс `Lister` (глава 25) так, чтобы он выводил 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, Lister), address 7841200:
```

Как бы вы реализовали вывод значений унаследованных атрибутов? (Подсказка: у классов имеется атрибут `__dict__`.) Попробуйте расширить класс `Lister` так, чтобы он выводил список всех доступных суперклассов и их атрибуты (подсказка: порядок подъема по дереву наследования описывается в примере *classtree.py* в главе 24 и в сноске об использовании функций `dir` и `getattr` в Python 2.2 в разделе «Множественное наследование» в главе 25).

7. *Композиция.* Сымитируйте сценарий оформления заказа в ресторане быстрого питания, определив четыре класса:

Lunch

Вмещающий и управляющий класс.

Customer

Действующее лицо, покупающее блюдо.

Employee

Действующее лицо, принимающее заказ.

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`, кроме того, экспортировать метод 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 блюда (новый объект `Food`) у официанта (`Employee`) вызовом метода `takeOrder` класса `Employee`.
- b. Объекты типа `Food` должны сохранять строку с названием блюда (например, «буррито»), которое передается через `Lunch.order` в `Customer.placeOrder`, затем в `Employee.takeOrder` и, наконец, в конструктор класса `Food`. Кроме 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, чтобы клиент (`Customer`) мог вызвать метод официанта (`Employee`).

Поэкспериментируйте с получившимися классами в интерактивной оболочке, 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 атрибутом `__name__` из главы 21. В этой имитации активность проявляет клиент (`Customer`); как бы вы изменили свои классы, чтобы инициатором взаимодействий между клиентом и официантом был официант (`Employee`)?

8. *Классификация животных в зоологии.* Изучите дерево классов, представленное на рис. 26.1. Напишите шесть инструкций `class`, которые моделировали бы эту модель классификации средствами наследования в языке Python. 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 текст сообщения,

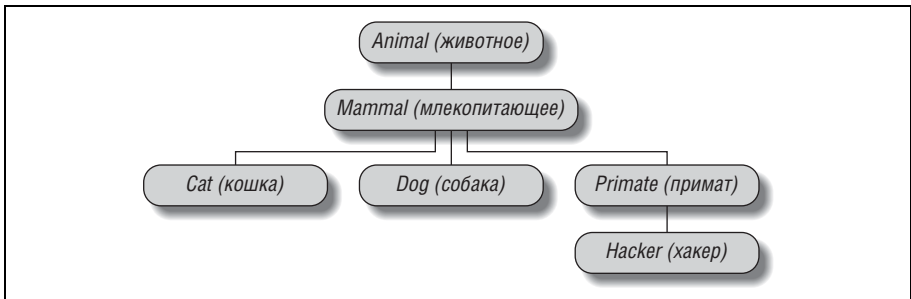


Рис. 26.1. Классификация животных в зоологии, составленная из классов, связанных в дерево наследования. Класс *Animal* имеет общий метод «reply», но каждый из классов имеет свой собственный метод «speak», который вызывается методом «reply»

характерного для каждой категории в подклассах, расположенных 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, представленную на рис. 26.2. Напишите набор классов на языке Python, которые 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 должен определять метод *line*, выводящий уникальное сообщение. Встраиваемые 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
```

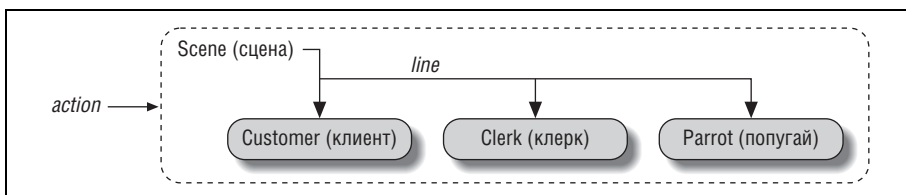


Рис. 26.2. Составная сцена, представленная управляющим классом (*Scene*), который встраивает и управляет экземплярами трех других классов (*Customer*, *Clerk*, *Parrot*). Встроенные экземпляры классов могут также участвовать в иерархии наследования – композиция и наследование часто являются одинаково полезными способами организации классов с целью обеспечения возможности повторного использования программного кода

Придется держать в уме: ООП глазами специалистов

Когда я рассказываю о классах в языке Python, я все время обнаруживаю, что в середине лекции о классах люди, имевшие опыт ООП в прошлом, заметно активизируются, а те, кто такого опыта не имеет, начинают 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 22 – настоятельно рекомендую вам перечитать эту главу, как только вам начинает казаться, что ООП – это всего лишь некоторое украшение в программировании.

В реальной аудитории, чтобы привлечь (и удержать) внимание начинающих программистов, я обычно останавливаюсь и спрашиваю у присутствующих опытных специалистов, почему они используют ООП. Ответы, которые они дают, могут пролить свет на цели, которые преследует ООП для тех, кто плохо знаком с этой темой.

Ниже приводятся лишь самые общие причины, побуждающие использовать 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 писать программы, адаптируя существующий программный код, а не писать каждый новый проект с самого начала.

Инкапсуляция

Скрытие деталей реализации за интерфейсом объекта предохраняет пользователей класса от необходимости изменять свой программный код.

Организация

Классы предоставляют новые локальные области видимости, которые минимизируют вероятность конфликтов имен. Кроме того, они обеспечивают место для естественного размещения программного кода реализации и управления состоянием объекта.

Поддержка

Классы обеспечивают естественное разделение программного кода, что позволяет уменьшить его избыточность. Благодаря организации и возможности повторного использования программного кода в случае необходимости бывает достаточно изменить всего одну копию программного кода.

Непротиворечивость

Классы и возможность наследования позволяют реализовать общие интерфейсы и, следовательно, обеспечить единообразие вашего программного кода – такой код легко поддается отладке, выглядит более осмысленно и прост в сопровождении.

Полиморфизм

Это скорее свойство ООП, чем причина его использования, но благодаря поддержке общности программного кода полиморфизм делает код более гибким, расширяет область его применения и, следовательно, увеличивает его шансы на повторное использование.

Другие

И, конечно, причина номер один состоит в том, что упоминание о владении приемами ООП увеличивает шанс быть принятым на работу! (Согласен, я привел эту причину в шутку, но, если вы собираетесь работать 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 использовать его какое-то время. Выберите себе проект, изучите большие примеры, поработайте над упражнениями – это заставит вас попотеть над объектно-ориентированным программным кодом, но оно стоит того.

VII

Исключения и инструменты

27

ОСНОВЫ ИСКЛЮЧЕНИЙ

В последней части книги рассказывается об *исключениях*, которые, по сути, являются событиями, способными изменить ход выполнения программы. Исключения в языке Python возбуждаются автоматически, когда программный код допускает ошибку, а также могут возбуждаться и перехватываться самим программным кодом. Обработываются 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 – является дополнительным расширением до выхода версии Python 2.6:

`try/except`

Перехватывает исключения, возбужденные интерпретатором или вашим программным кодом, и выполняет восстановительные операции.

`try/finally`

Выполняет заключительные операции независимо от того, возникло ли исключение или нет.

`raise`

Дает возможность возбудить исключение программно.

`assert`

Дает возможность возбудить исключение программно, при выполнении определенного условия.

`with/as`

Реализует менеджеры контекста в версии Python 2.6 и выше (в версии 2.5 является дополнительным расширением).

Эта тема была оставлена напоследок потому, что для работы с исключениями 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 непосредственно в сам язык, как и другие высокоуровневые средства.

Одно техническое примечание перед началом: после выхода первого издания этой книги в исключениях произошли два изменения – предложение `finally` может теперь присутствовать в инструкции `try` вместе с предложениями `except` и `else`, а исключения, определяемые программой, теперь должны быть экземплярами классов, а не простыми строками. В этом издании я опишу и старый, и новый способы работы с исключениями, потому что в существующем программном коде вам еще часто будут встречаться изначальные приемы. Попутно я расскажу, какими путями шло развитие в этой области. Я также опишу новую инструкцию `with` несмотря на то, что ее официальное появление ожидается только в следующем выпуске Python.

Зачем нужны исключения?

В двух словах, исключения позволяют перепрыгнуть через фрагмент программы 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 программу, написанную на языке Python: она должна принимать заказ, приготовить тесто, выбрать добавки, выпечь основу и т. д.

Теперь предположим, что что-то пошло совсем не так во время «выпекания основы». Возможно, сломалась печь или, возможно, наш робот ошибся в расчетах расстояния до печи и воспламенился. Совершенно очевидно, что нам необходимо предусмотреть быстрый переход к программному коду, который быстро обрабатывает такие ситуации. Кроме того, поскольку в таких необычных условиях у нас нет никакой надежды на успешное окончание процесса приготовления 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. Исключение – это своего рода «супер-`goto`».¹ Обработчик исключений (инструкция `try`) ставит метку и выполняет некоторый программный код. Если затем где-нибудь в программе возникает исключе-

¹ Если вы использовали язык C, вам будет интересно узнать, что исключения в языке Python немного похожи на стандартную для языка C пару функций `setjmp/longjmp`: инструкция `try` действует как функция `setjmp`, а инструкция `raise` как `longjmp`. Только в языке Python исключения основаны на объектах и являются стандартной частью модели исполнения.

ние, интерпретатор немедленно возвращается к метке, отменяя все активные вызовы функций, которые были произведены после установки метки. Код в обработчике исключения может соответствующим образом отреагировать на ситуацию (вызвать пожарных, например). Кроме того, переход к обработчику 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 обнаруживает ошибку во время выполнения программы. Программа может перехватывать такие ошибки и обрабатывать их или просто игнорировать. Если ошибка игнорируется, интерпретатор выполняет действия, предусмотренные по умолчанию, — останавливает выполнение программы и выводит сообщение об ошибке. Если такое поведение по умолчанию является нежелательным, можно добавить инструкцию `try`, которая позволит перехватывать 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 флаги результата или явно проверять их. Например, функция поиска может возбуждать исключение в случае неудачи, вместо того чтобы возвращать целочисленный признак в виде результата (и надеяться, что этот признак всегда будет интерпретироваться правильно).

Обработка особых ситуаций

Некоторые условия могут наступать так редко, что было бы слишком расточительно предусматривать проверку наступления таких условий с целью их обработки. Нередко такие проверки необычных ситуаций можно заменить 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 можно использовать как основу для экзотического управления потоком выполнения 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

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 рассматривались в этой книге, исключения в языке Python представляют собой чрезвычайно легкий инструмент. Поскольку они так просты, перейдем сразу к первому примеру. Предположим, что мы пишем следующую функцию:

```
>>> def fetcher(obj, index):  
...     return obj[index]  
...
```

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. Обнаруживая выход за пределы последовательности, интерпретатор сообщает об этом, *возбуждая* встроенное исключение `IndexError`:

```
>>> fetcher(x, 4)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in ?  
  File "<stdin>", line 2, in fetcher  
IndexError: string index out of range  
(IndexError: выход индекса за пределы строки)
```

Поскольку наш программный код не перехватывает это исключение явно, оно возвращает выполнение на верхний уровень программы и вызы-

¹ Настоящая обратная трассировка – это довольно сложная тема, и данная возможность не является частью самого языка Python (даже с появлением функций-генераторов в версии 2.2), поэтому я не буду больше говорить об этом. Грубо говоря, обратная трассировка отменяет все вычисления перед переходом – исключения этого не делают (то есть в переменных, которым было выполнено присваивание между моментом выполнения инструкции `try` и до момента возбуждения исключения, прежние значения не восстанавливаются). Если вам любопытна эта тема, обращайтесь к книгам по искусственному интеллекту или языкам программирования Prolog или Icon.

вает *обработчик исключений по умолчанию*, который просто выводит стандартное сообщение об ошибке. К настоящему моменту вы наверняка видели в своих программах подобные сообщения об ошибках. Они включают тип исключения, а также диагностическую информацию – список строк и функций, которые были активны в момент появления исключения. При работе в интерактивной оболочке файлом является «stdin» (стандартный поток ввода) или «pyshell» (в IDLE), поэтому в данном случае номера строк не несут сколько-нибудь полезной информации.

В настоящей программе, запущенной не в интерактивной оболочке, обработчик кроме этого по умолчанию *завершает* работу программы. 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. Но 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, достаточно просто перехватить исключение, обернув вызов функции инструкцией `try`:

```
>>> try:
...     fetcher(x, 4)
... except IndexError:
...     print 'got exception'
...
got exception
>>>
```

Теперь, когда исключение будет возникать при выполнении инструкций в блоке `try`, интерпретатор будет автоматически переходить к вашему *обработчику* (блок под предложением `except`, в котором указано имя исключения). При работе 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 действия 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, программа продолжила выполнение ниже всей инструкции `try` –

именно поэтому в данном примере было выведено сообщение «continuing». Стандартное сообщение об ошибке не появилось на экране, и программа продолжила работу как ни в чем не бывало.

Исключения могут возбуждаться интерпретатором или самой программой, и могут перехватываться или не перехватываться. Чтобы возбудить исключение вручную, достаточно просто выполнить инструкцию `raise` (или `assert`, с условным выражением, возвращающим ложь). Исключения, определяемые программой, перехватываются точно так же, как и встроенные исключения:

```
>>> bad = 'bad'
>>> try:
...     raise bad
... except bad:
...     print 'got bad'
...
got bad
```

Если исключение, определяемое программой, не перехватывается, оно будет передано обработчику исключений по умолчанию, что приведет к завершению программы с выводом стандартного сообщения об ошибке. В данном случае стандартное сообщение включает текст строки, использовавшейся для идентификации исключения:

```
>>> raise bad
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in ?
    raise bad
bad
```

В других случаях сообщение об ошибке может включать текст, предоставляемый классами, использованными для идентификации исключений. Как будет показано в следующей главе, исключения, определяемые в программе, могут определяться в виде строк или классов, но в отличие от строк, исключения на базе классов позволяют сценариям создавать категории исключений, наследовать поведение и добавлять к ним информацию о состоянии. Исключения на базе классов более предпочтительны, чем исключения на базе строк, а кроме того, они станут обязательными в Python 3.0:

```
>>> class Bad(Exception): pass
...
>>> def doomed(): raise Bad()
...
>>> try:
...     doomed()
... except Bad:
...     print 'got Bad'
...
got Bad
>>>
```

Наконец, инструкции `try` могут включать блоки `finally`. Комбинация `try/finally` определяет завершающие действия, которые всегда выполняются «на выходе», независимо от того, возникло исключение в блоке `try` или нет:

```
>>> try:
...     fetcher(x, 3)
... finally:
...     print 'after fetch'
...
'm'
after fetch
```

Здесь, если блок `try` выполнится без ошибок, будет выполнен блок `finally` и программа продолжит свою работу дальше. В этом случае данная инструкция 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/finally` 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 ?
  File "<stdin>", line 3, in after
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
(IndexError: выход индекса за пределы строки)
```

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`, после чего исключение достигло предыдущего обработчика (в данном случае – обработчик по умолчанию). Если изменить вызов внутри функции `action`, чтобы он не вызывал исключение, блок `finally` все равно будет выполнен, но программа 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 восстановления после исключений, а комбинацию `try/finally` – в случаях, когда необходимо гарантировать выполнение заключительных действий независимо от того, возникло исключение в блоке `try` или нет. Например, комбинацию `try/except` можно было бы использовать для перехвата ошибок, возникающих в импортированной 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 предложения `except` и `finally` в одной и той же инструкции `try` – блок `finally` будет выполняться всегда, независимо от того, было ли перехвачено исключение предложением `except`.

Такое поведение составляет основу исключений – исключения действительно являются очень простым инструментом. Далее в этой части книги мы подробнее поговорим о самих инструкциях, исследуем разные виды предложений, которые могут появляться в инструкции `try`, и обсудим объекты исключений, основанные на строках и классах.

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 инструкций `try` (для обработки в своем программном коде). Инструкция `try` может 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 возбуждать вручную, с помощью инструкций `raise` и `assert`. А теперь, получив общее представление, рассмотрим подробнее общие формы употребления этих инструкций.

Инструкция try/except/else

Для начала я представляю try/except/else и try/finally как разные инструкции, потому что они имеют разное предназначение и не могут комбинироваться в версиях Python ниже, чем 2.5. Как уже говорилось, начиная с этой версии, except и finally могут смешиваться 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 располагается блок инструкций (как правило) с отступами, затем следует одно или более предложений except, которые определяют перехватываемые исключения, и затем 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>, <data>:
    <statements>                # Запускается в случае исключения name2
                                # и получает дополнительные данные
except (name3, name4):
    <statements>                # Запускается, если возникло любое из этих исключений
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 обработчик для случая *отсутствия* исключений. Элемент <data> имеет отношение к особенности инструкций raise, которая будет обсуждаться далее в этой главе.

Ниже описывается принцип действия инструкции try. Когда запускается инструкция 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, зависит 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 другого исключения).

- Если в блоке `try` возникло исключение, и не было найдено ни одного соответствия среди предложений `except`, исключение будет передано инструкции `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, интерпретатор выполнит инструкции в блоке `else` (если имеется) и затем выполнение продолжится с первой инструкции, находящейся за всей инструкцией `try`.

Другими словами, предложения `except` перехватывают любые исключения, которые 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 перехватывают 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 29.

Предложения инструкции `try`

В инструкции `try` могут присутствовать разные предложения, располагающиеся вслед за блоком `try`. В табл. 27.1 приводятся все возможные формы, из которых хотя бы одна должна присутствовать. Мы уже встречали некоторые из них: как вы уже знаете, предложение `except` перехватывает исключения, предложение `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 Python 2.4 предложение `finally` должно быть единственным (без предложений `else` или `except`); в действительности это отдельная инструкция. Однако, начиная с версии Python 2.5, предложение `finally` может присутствовать в той же инструкции, что и предложения `except` и `else`.

Таблица 27.1. Различные формы предложений в инструкции `try`

Форма предложения	Интерпретация
<code>except:</code>	Перехватывает все (остальные) типы исключений.
<code>except name:</code>	Перехватывает только указанное исключение.
<code>except name, value:</code>	Перехватывает указанное исключение и получает соответствующие ему дополнительные данные (или экземпляр).

Форма предложения	Интерпретация
<code>except (name1, name2):</code>	Перехватывает любое из перечисленных исключений.
<code>except (name1, name2), value:</code>	Перехватывает любое из перечисленных исключений и получает соответствующие им дополнительные данные.
<code>else:</code>	Выполняется, если не было исключений.
<code>finally:</code>	Этот блок выполняется всегда.

Исследованием дополнительного значения `value` мы займемся, когда будем рассматривать инструкцию `raise`. Новыми здесь для нас являются первая и четвертая строки в табл. 27.1:

- 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 исключений (`except (e1, e2, e3):`), перехватывают *любое* из перечисленных исключений.

Интерпретатор Python просматривает предложения `except` сверху вниз в поисках соответствия, поэтому версию предложения с круглыми скобками можно рассматривать как аналог нескольким отдельным выражениям `except`, по одному 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 порядок 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 инструкции в первом найденном совпадении. Если совпадений не будет найдено, исключение продолжит распространение выше этой инструкции `try`. Обратите внимание, что блок `else` выполняется только при *отсутствии* исключения в функции `action` —

этот блок не выполняется при наличии исключения, которому не было найдено соответствующее предложение `except`.

Если вам действительно необходимо организовать перехват всех исключений, используйте пустое предложение:

```
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 символ, потому что оно перехватывает любые исключения, что позволяет вам создавать и универсальные, и специфичные обработчики, по своему усмотрению. В некоторых случаях эта форма может быть более удобна, чем перечисление всех возможных исключений в инструкции `try`. Например, в следующем примере выполняется 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 программного кода, и по случайности прерывать распространение исключений, предназначенных для других обработчиков. Например, даже выход из программы в языке Python возбуждает исключение, и поэтому было бы желательно, чтобы это исключение было пропущено. Но пока я скажу лишь, что предложение `except` требует внимательного отношения.



В версии Python 3.0 предполагается изменить форму, представленную в третьей строке табл. 27.1: `except name, value:` планируется преобразовать в форму `except name as value:`. Этим изменением предполагается ликвидировать возможность перепутать эту форму со случаем, когда в предложении `except` указывается кортеж имен исключений – в версии Python 3.0 форма в четвертой строке табл. 27.1 больше не будет требовать наличия круглых скобок. Это изменение также приведет к изменению правил области видимости: с введением новой синтаксической конструкции `as переменная value` будет удаляться в конце блока `except`.

Кроме того, в версии 3.0 разновидность инструкции `raise E, V` должна будет записываться как `raise E(V)`, чтобы явно обозначить

создание класса, который будет играть роль возбуждаемого исключения. Предыдущая форма будет сохранена для обратной совместимости со строковыми исключениями в Python 2.x. (Далее в этой главе подробнее рассказывается об инструкции `raise`, а в следующей главе рассказывается об исключениях на базе классов.)

В настоящее время невозможно использовать форму `as` предложения `except` в Python 2.x, чтобы подготовить свой программный код к грядущим изменениям, однако в составе Python 3.0 будет распространяться инструмент преобразования «2to3», который автоматически будет выполнять преобразования предложения `except` и инструкции `raise` в написанном для версии Python 2.x программном коде.

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 исключение 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 выхода 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 исключения ниже (тонко, но верно!). При явном использовании выражения `else` логика выполнения становится более очевидной и гарантируется, что обработчики исключений будут вызываться только для обработки истинных ошибок в блоке, обернутом инструкцией `try`, а не при выполнении действий, предусмотримых в блоке `else`.

Пример: поведение по умолчанию

Поскольку объяснить порядок выполнения программы проще на языке Python, чем на естественном языке, рассмотрим несколько примеров, иллюстрирующих основы исключений. Я уже упоминал, что исключения, не перехваченные инструкциями `try`, распространяются до самого верхнего уровня процесса и запускают логику обработки исключений по умолчанию (то есть интерпретатор аварийно завершает работающую программу и выводит стандартное 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 завершает ее и выводит сообщение:¹

```
% 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: integer division or modulo by zero
(ZeroDivisionError: целочисленное деление или деление по модулю на ноль)
```

Сообщение состоит из содержимого стека и имени (с дополнительными данными) 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 интерак-

¹ Текст сообщений и отладочная информация время от времени немного изменяются. Поэтому не надо беспокоиться, если ваши сообщения не соответствуют в точности тем, что приводятся здесь. Например, когда я запускал этот пример в среде IDLE, входящей в состав Python 2.5, в тексте сообщения выводились полные пути к файлам.

тивной оболочке интерпретатора, имена файлов и номера строк содержат полезную для нас информацию. Например, здесь видно, что ошибка произошла во 2 строке в файле *bad.py*, в инструкции *return*.

Так как интерпретатор Python определяет и сообщает обо всех ошибках, появившихся во время выполнения программы, возбуждая исключения, эти исключения тесно связаны с идеями обработки ошибок и отладки вообще. Если вы работали с примерами из этой книги, вы без сомнений встречались с несколькими исключениями — даже опечатки нередко приводят к возбуждению исключения *SyntaxError* или других при импортировании и выполнении файла (то есть, когда запускается компилятор). По умолчанию интерпретатор выводит полезное информативное сообщение, как показано выше, которое позволяет легко отыскать источник проблем.

Чаще всего для разрешения проблем достаточно стандартных сообщений об ошибках. Для более надежной отладки своих программ вы можете перехватывать исключения с помощью инструкций *try* или использовать средства отладки, которые будут представлены в главе 29, такие как модуль *pdb* из стандартной библиотеки.

Пример: перехват встроенных исключений

Обработка исключений, которая выполняется интерпретатором по умолчанию, зачастую удовлетворяет всем нашим потребностям, особенно для программного кода верхнего уровня, где ошибки должны приводить к немедленному завершению программы. Для большинства программ нет никакой необходимости предусматривать какие-то особые варианты обработки ошибок.

Однако иногда бывает необходимо перехватить ошибку и выполнить восстановительные 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 важная возможность для таких программ, как серверы сети, которые должны продолжать работать постоянно. Например, следующий фрагмент перехватывает и обрабатывает исключение *TypeError*, которое возбуждается интерпретатором при попытке выполнить операцию конкатенации для списка и строки (оператор *+* 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 предложению `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.

Обратите внимание: как только ошибка будет перехвачена, выполнение продолжается с того места, где ошибка была перехвачена (то есть после инструкции `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 функций – нет никакой возможности вернуться к программному коду, вызвавшему ошибку.

Инструкция `try/finally`

Другая разновидность инструкции `try` специализируется на выполнении завершающих действий. Если в инструкцию `try` включено предложение `finally`, интерпретатор всегда будет выполнять этот блок инструкций при «выходе» из инструкции `try` независимо от того, произошло ли исключение во время выполнения инструкций в блоке `try`. Общая форма этой инструкции имеет следующий вид:

```
try:
    <инструкция> # Выполнить эти действия первыми
finally:
    <инструкция> # Всегда выполнять этот блок кода при выходе из инструкции
```

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 возникло, интерпретатор переходит к выполнению блока `finally` и затем продолжает 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`, `finally` не завершает распространение исключения – оно остается актуальным после выполнения блока `finally`.

Форма `try/finally` бывает удобна, когда необходимо гарантировать выполнение некоторых действий независимо от реакции программы на исключение. С практической точки зрения эта форма инструкции по-

звояет определять 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 `finally` не может использоваться в той же инструкции `try`, где уже используется предложение `except` или `else`, поэтому форму `try/finally` лучше считать отдельной формой инструкции при работе со старыми версиями. Однако в Python 2.5 предложение `finally` может присутствовать в инструкции `try` вместе с предложениями `except` и `else`, поэтому в настоящее время существует единая инструкция `try`, которая может употребляться с несколькими необязательными предложениями (вскоре мы поговорим об этом подробнее). Однако какую бы версию Python вы не использовали, назначение предложения `finally` остается прежним – определить завершающие действия, которые должны выполняться всегда, независимо от возникновения исключений.



Как будет показано далее в этой главе, в версии Python 2.6, инструкция `with` и контекстные менеджеры обеспечивают объектно-ориентированный подход к выполнению аналогичных завершающих действий, но кроме того, эта инструкция поддерживает возможность выполнения действий по инициализации.

Пример: реализация завершающих действий с помощью инструкции try/finally

Выше мы видели несколько простых примеров применения инструкции `try/finally`. Ниже приводится более близкий к реальности пример, иллюстрирующий 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
    ...                   # Продолжить с этого места,
                           # только если не было исключения
```

В этом фрагменте мы обернули вызов функции в инструкцию `try`, с предложением `finally`, чтобы гарантировать, что файл будет закрыт при любых обстоятельствах, 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 закрытие соединения с сервером и т. д.

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 интерпретатор 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 передается обратно инструкции `try` и начинает выполняться блок `finally`, в котором производится закрытие файла. После этого исключение продолжает свое распространение либо пока не встретит другую инструкцию `try`, либо пока не будет достигнут обработчик по умолчанию, который выведет стандартное сообщение об ошибке и остановит работу программы; инструкция, находящая 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 настоящее время 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 разновидностях, и в действительности имелось две отдельные инструкции. Мы могли либо использовать предложение `finally`, чтобы гарантировать выполнение завершающего программного кода, либо писать блоки `except`, чтобы перехватывать определенные исключения и выполнять действия по восстановлению после 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 восстановление после исключений выглядит никак не связанным с выполнением действий по завершению.

Однако в Python 2.5 (эта версия Python используется в данном издании книги) две инструкции были объединены. Сейчас у нас имеется

¹ Если, конечно, сам интерпретатор не завершит свою работу аварийно. Разработчики Python упорно трудятся над тем, чтобы избежать подобного развития событий, проверяя 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.

возможность смешивать предложения *finally*, *except* и *else* в одной и той же инструкции. То есть теперь мы можем написать инструкцию, имеющую следующий вид:

```
try:
    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, соответствующего возникшему исключению. Если было возбуждено исключение *Exception1*, будет выполнен блок *обработчик1*, исключение *Exception2* приведет к запуску *обработчика2* и т. д. 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 возникших 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 исключения – если к моменту выполнения блока *finally* имеется активное исключение, оно продолжает свое распространение после выполнения блока *finally* и управление передается куда-то в другое место программы (другой инструкции *try* или обработчику по умолчанию). Если к моменту, когда блок *finally* будет выполнен, нет активного исключения, выполнение программы продолжится сразу 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 выполнение завершающих действий, которые должны выполняться всегда при выходе из инструкции *try*, независимо от того, было ли возбуждено исключение и было ли оно обработано.

Объединение `finally` и `except` вложением

До появления версии Python 2.5 существовала возможность объединять предложения `finally` и `except` в инструкции `try` за счет вложения инструкции `try/except` в блок `try` инструкции `try/finally` (более полно этот прием будет рассматриваться в главе 29). В действительности фрагмент ниже имеет тот же эффект, что и новая форма инструкции, представленная в предыдущем разделе:

```
try:
    try:
        основное действие
    except Exception1:
        обработчик1
    except Exception2:
        обработчик2
    ...
except:
    нет ошибок
finally:
    завершающие действия
```

Здесь также блок `finally` всегда выполняется при выходе из инструкции `try` независимо 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). Однако этот эквивалент выглядит менее понятным, чем новая, объединенная форма инструкции. Смешанная форма инструкции проще в написании и выглядит понятнее, поэтому такая форма записи считается в настоящее время предпочтительной.

Пример использования объединенной инструкции `try`

Ниже приводится пример использования объединенной формы инструкции `try`. Здесь представлены четыре типичных варианта с инструкциями `print`, описывающими значение каждого из них:

```
print '-' * 30, '\nEXCEPTION RAISED AND CAUGHT'
try:
    x = 'spam'[99]
except IndexError:
    print 'except run'
finally:
    print 'finally run'
print 'after run'

print '-' * 30, '\nNO EXCEPTION RAISED'
try:
    x = 'spam'[3]
except IndexError:
    print 'except run'
```

```

finally:
    print 'finally run'
print 'after run'

print '-' * 30, '\nNO EXCEPTION RAISED, ELSE RUN'
try:
    x = 'spam'[3]
except IndexError:
    print 'except run'
else:
    print 'else run'
finally:
    print 'finally run'
print 'after run'

print '-' * 30, '\nEXCEPTION RAISED BUT NOT CAUGHT'
try:
    x = 1 / 0
except IndexError:
    print 'except run'
finally:
    print 'finally run'
print 'after run'

```

После запуска этот пример выводит на экран следующий ниже текст. Исследуйте программный код, чтобы понять, как работает каждый из вариантов:

```

-----
EXCEPTION RAISED AND CAUGHT
except run
finally run
after run
-----
NO EXCEPTION RAISED
finally run
after run
-----
NO EXCEPTION RAISED, ELSE RUN
else run
finally run
after run
-----
EXCEPTION RAISED BUT NOT CAUGHT
finally run

Traceback (most recent call last):
  File "C:/Python25/mergedexc.py", line 32, in <module>
    x = 1 / 0
ZeroDivisionError: integer 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 интер-

претатор всегда определяет появление ошибок во время выполнения программного кода. В следующем 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` состоит из слова `raise`, за которым могут следовать имя возбуждаемого исключения и дополнительный элемент данных, передаваемый вместе с исключением:

```
raise <name>           # Возбуждает исключение
raise <name>, <data>    # Передает дополнительные данные для обработчика
raise                  # Повторно возбуждает самое последнее исключение
```

Вторая форма позволяет передавать дополнительные данные вместе с исключением, чтобы сообщить обработчику о некоторых подробностях. Данные, указанные после имени исключения в инструкции `raise`, передаются обратно в инструкцию `try` в виде переменной. Если инструкция `try` включает предложение `except name, X:`, то дополнительный элемент данных, указанный в инструкции `raise`, будет присвоен переменной `X`. Третья форма инструкции `raise` просто повторно возбуждает текущее исключение – это удобно, когда возникает необходимость передать перехваченное исключение другому обработчику.

А что используется в качестве имени исключения? Это может быть имя встроенного исключения из встроенной области видимости (например, `IndexError`) или имя любого строкового объекта в программе. Это может быть также ссылка на класс или экземпляр класса – данная возможность позволяет еще больше обобщить формат инструкции `raise`. Я не буду касаться подробностей такого обобщения, пока мы не приступим к изучению классов исключений в следующей главе

Независимо от того, какие исключения будут использованы, они всегда идентифицируются обычными объектами и только одно исключение может быть активным в каждый конкретный момент времени. Как только исключение перехватывается предложением `except`, находящимся в любом месте программы, исключение деактивируется (то есть оно не будет передано другой инструкции `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 собственные исключения в программе должны быть представлены объектами экземпляров классов, как, например, `MyBad` в следующем примере:

```
class MyBad: pass

def stuff():
    raise MyBad() # Возбудить исключение вручную
try:
    stuff()       # Возбуждает исключение
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 исключения программы точно так же, как и встроены исключения.

Пример: передача дополнительных данных инструкции raise

Как утверждалось выше, инструкция `raise` может передавать дополнительные данные в месте с исключением для использования в обработчике. Дополнительные данные позволяют передавать обработчику контекстную информацию об исключении. Например, если вы пишете программу анализа содержимого файла, то в случае нахождения ошибки можно было бы возбуждать исключение синтаксической ошибки и вместе с этим исключением передавать обработчику объект, включающий в себя строку и информацию о файле (такой пример будет рассматриваться в главе 28).

Это бывает полезно, потому что после того, как исключение будет возбуждено, оно может быть перехвачено в другом модуле, то есть инструкция `raise`, возбудившая исключение, и инструкция `try`, перехватившая его, могут находиться в совершенно разных файлах. Поэтому вообще может оказаться невозможным использовать глобальные переменные для сохранения дополнительной информации об исключении, потому что инструкция `try` может даже не знать, в каком модуле находятся эти переменные. Возможность передачи данных непосредственно с самим исключением обеспечивает более надежный доступ к ним в инструкции `try`. Строго говоря, дополнительные данные имеются у каждого исключения: как в случае с возвращаемыми значениями функций, если данные не были переданы явно, то в качестве таких данных используется специальный объект `None`. Следующий фрагмент, *raisedata.py*, иллюстрирует действие этой концепции на примере простых исключений на базе строк:

```
myException = 'Error' # Строковый объект

def raiser1():
    raise myException, "hello" # Возбудить исключение, передать данные

def raiser2():
```

```

        raise myException          # Возбудить исключение, предполагается None

def tryer(func):
    try:
        func()                    # Вызвать func
    except myException, extraInfo: # перехватить исключение с данными
        print 'got this:', extraInfo

% python
>>> from raisedata import *
>>> tryer(raiser1)                # Данные были переданы явно
got this: hello
>>> tryer(raiser2)                # Дополнительные данные - None по умолчанию
got this: None

```

Здесь функция `tryer` всегда запрашивает получение объекта с дополнительными данными — он поступает из функции `raiser1` в виде явно заданной строки, а из функции `raiser2` — в виде объекта `None`, используемого по умолчанию.

В следующей главе мы увидим, что тот же самый прием может использоваться для организации доступа к экземплярам при использовании исключений на базе классов — в этом случае переменной в предложении `except` присваивается ссылка на экземпляр класса исключения, который обеспечивает доступ к присоединенной информации и к методам класса.

Пример: повторное возбуждение исключений с помощью инструкции `raise`

Инструкция `raise`, в которой отсутствует имя исключения или дополнительные данные, просто повторно возбуждает текущее исключение. В таком виде она обычно используется, когда необходимо перехватить и обработать исключение, но при этом не требуется деактивировать исключение:

```

>>> try:
...     raise IndexError, 'spam'
... except IndexError:
...     print 'propagating'
...     raise
...
propagating
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
IndexError: spam

```

При таком использовании инструкция `raise` повторно возбуждает исключение, которое затем передается обработчику более высокого уровня или обработчику по умолчанию, который останавливает выполнение программы и выводит стандартное сообщение об ошибке.

Инструкция assert

Язык Python включает инструкцию `assert` в качестве особого случая возбуждения исключений. Это сокращенная форма типичного шаблона использования инструкции `raise`, которая представляет собой *условную* инструкцию `raise`. Инструкция в форме:

```
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 дополнительных данных исключения. Как и все исключения, исключение `AssertionError` приводит к завершению программы, если не будет перехвачено инструкцией `try`.

Существует дополнительная возможность удалить все инструкции `assert` из скомпилированного байт-кода программы за счет использования флага командной строки `-O` при запуске интерпретатора и тем самым оптимизировать программу. Исключение `AssertionError` является встроенным исключением, а имя `__debug__` — встроенным флагом, который автоматически получает значение `1` (истина), когда не используется флаг `-O`.

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 ошибках, 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 ?
  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 про-

граммистом, а не для перехвата настоящих ошибок. Так как интерпретатор Python в состоянии сам выявлять ошибки во время выполнения программы, обычно нет необходимости 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 могли бы положиться в этом на него.¹ Еще один пример типичного использования инструкции `assert` приводится в примере абстрактного суперкласса в главе 24 – там инструкция `assert` использовалась для того, чтобы вызов неопределенных методов приводил к исключению с определенным текстом сообщения.

Контекстные менеджеры `with/as`

В версии Python 2.6 (на момент, когда писалась эта книга, данная версия еще не вышла) появится 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 использовании методов.

Проще говоря, инструкция `with/as` может использоваться как альтернатива известной идиомы `try/finally`; подобно этой инструкции она предназначена для выполнения завершающих действий независимо от того, возникло ли исключение на этапе выполнения основного действия. Однако, в отличие от инструкции `try/finally`, инструкция `with` поддерживает более богатый возможностями протокол, позволяющий определять как предварительные, так и заключительные действия для заданного блока программного кода.

Язык Python дополняет некоторые встроенные средства контекстными менеджерами, например, файлы, которые закрываются автоматически, или блокировки потоков выполнения, которые автоматически запираются и отпираются. Однако программист также может создавать с классами и свои контекстные менеджеры.

¹ По крайней мере, в большинстве случаев. Как предлагалось в четвертой части книги, проверка на наличие ошибки может использоваться в функции, выполняющей расчеты длительное время или необратимые действия. Но даже в этом случае старайтесь не использовать чрезмерно специализированные или чрезмерно ограничительные проверки, т. к. в противном случае это ограничит область применения вашего программного кода.

Основы использования

Данная функциональная возможность официально станет частью языка Python, только начиная с версии 2.6. В Python 2.5 по умолчанию она еще не доступна – ее следует активировать с помощью специальной инструкции импорта, с которой мы уже встречались в этой книге при изучении модулей (так как появятся два новых зарезервированных слова `with` и `as`, это новшество, как обычно, вводится постепенно):

```
from __future__ import with_statement
```

Когда эта инструкция `import` выполняется в Python 2.5, появляется возможность использовать новую инструкцию `with` и два новых зарезервированных слова. 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 подробнее). Этот объект может возвращать значение, которое будет присвоено *переменной*, если присутствует необязательное предложение `as`.

Обратите внимание, что результат *выражения* не присваивается *переменной* – результатом *выражения* является объект, который поддерживает контекстный протокол, а *переменной* может быть присвоено некоторое другое значение. Объект, возвращаемый *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 исключение при его выполнении.

Некоторые встроенные объекты языка Python были дополнены поддержкой протокола управления контекстом и потому могут использоваться в инструкции `with`. Например, объекты файлов снабжены менеджером контекста, который автоматически закрывает файл после выполнения блока `with` независимо от того, было ли возбуждено исключение при его выполнении:

```
with open(r'C:\python\scripts') as myfile:  
    for line in myfile:  
        print line  
        line = line.replace('spam', 'SPAM')  
    ...остальной программный код...
```

Здесь вызываемая функция `open` возвращает объект файла, который присваивается 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, используемый инструкцией `with`. После того как инструкция `with` начнет выполнение, 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.

Мы не будем рассматривать в этой книге многопоточную модель выполнения в языке Python (за дополнительной информацией по этой теме вам следует обращаться к книгам, посвященным прикладному программированию, таким как «Programming Python»), но блокировка и средства синхронизации 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 автоматически будет приобретена до того, как начнет выполняться блок, и освобождена по завершении работы блока.

Модуль `decimal` (подробнее о числах с фиксированной точностью представления рассказывается в главе 5) также использует менеджеры контекста для упрощения сохранения и восстановления текущего контекста вычислений, определяющего параметры точности и округления, используемые в вычислениях.

Протокол управления контекстом

Интерфейс, который должны реализовать объекты для использования совместно с инструкцией `with`, достаточно сложен, хотя большинству программистов достаточно лишь знать, как используются существующие контексты менеджеров. Однако разработчикам программных инструментов может потребоваться знание правил создания новых менеджеров, поэтому коротко рассмотрим основные принципы. Ниже описывается, как в действительности работает инструкция `with`:

1. Производится вычисление выражения, возвращающего объект, известный как *менеджер контекста*, который должен иметь методы `__enter__` и `__exit__`.
2. Вызывается метод `__enter__` менеджера контекста. Возвращаемое значение метода присваивается переменной – при наличии предложения `as`, в противном случае оно просто уничтожается.
3. Затем выполняется блок программного кода, вложенный в инструкцию `with`.
4. Если при выполнении блока возбуждается исключение, вызывается метод `__exit__` (тип, значение, диагностическая информация), которо-

му передается подробная информация об исключении. Обратите внимания, что это те же самые значения, которые возвращает функция `sys.exc_info`, описываемая в руководстве по языку Python и далее в этой книге. Если этот метод возвращает ложное значение, исключение возбуждается повторно, в противном случае исключение деактивируется. Обычно исключение следует возбуждать повторно, чтобы оно могло выйти за пределы инструкции `with`.

5. Если в блоке `with` исключение не возникает, метод `__exit__` все равно вызывается, но в аргументах *тип, значение и диагностическая информация* ему передается значение `None`.

Рассмотрим небольшой пример, демонстрирующий работу протокола. Следующий фрагмент определяет объект менеджера контекста, который сообщает о входе и выходе из блока программного кода любой инструкции `with`, с которой он используется:

```
from __future__ import with_statement    # Требуется в Python 2.5

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'
        else:
            print 'raise an exception!', exc_type
            return False # propagate

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 разрешить дальнейшее распространение исключения – отсутствие инструкции `return` обеспечивает тот же самый эффект, потому что в этом случае по умолчанию 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 метод 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`:

```
% python withas.py
starting with block
running test 1
reached
exited normally

starting with block
running test 2
raise an exception! <type 'exceptions.TypeError'>

Traceback (most recent call last):
  File "C:/Python25/withas.py", line 22, in <module>
    raise TypeError
TypeError
```

Менеджеры контекста являются новейшими инструментами, которые официально еще не стали частью языка Python, поэтому мы не будем рассматривать здесь дополнительные подробности (за полной информацией обращайтесь к стандартным руководствам по языку; например, новый стандартный модуль `contextlib` содержит дополнительные средства, которые могут использоваться для создания менеджеров контекстов). В более простых случаях инструкция `try/finally` обеспечивает достаточную поддержку для выполнения завершающих действий.

Придется держать в уме: проверка ошибок

Один из способов увидеть, насколько полезными могут быть исключения, 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 результаты 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();
}
```

Фактически в настоящих программах на языке C значительная доля всего программного кода выполняет проверку наличия ошибок. Но в языке Python такая настойчивость и методичность не требуется. Достаточно просто обернуть произвольные участки программы 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. Кроме того, благодаря тому, что интерпретатор Python автоматически обнаруживает 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 проверки на наличие ошибок.

В заключение

В этой главе мы приступили к изучению вопросов обработки исключений и к исследованию инструкций, связанных с исключениями: инструкция `try` используется для перехвата исключений, `raise` используется для их возбуждения, `assert` используется для возбуждения исключений по условию и `with` используется для обертывания программного кода менеджерами контекстов, определяющими действия на входе и выходе.

Пока исключения выглядят достаточно простым инструментом, впрочем, таковым они и являются, — единственная сложность заключается в их идентификации. Следующая глава продолжит наши исследования описанием реализации наших собственных объектов исключений, где будет показано, что классы позволяют создавать исключения более

полезные, чем простые строки. Однако, 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 напоминает?
5. Для чего служит инструкция `with/as` и какие другие инструкции она напоминает?

Ответы

1. Инструкция `try` служит для перехвата исключений и проведения восстановительных действий после них. Она определяет блок выполняемого программного 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 действий, 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 посредством внутренних механизмов возбуждает встроенные исключения, а ваши сценарии с помощью инструкции `raise` могут возбуждать как встроенные, так и свои собственные исключения.
4. Инструкция `assert` возбуждает исключение `AssertionError`, когда условное выражение возвращает ложное значение. Она напоминает инструкцию `raise`, обернутую инструкцией `if`.
5. Инструкция `with/as` предназначена для автоматического запуска программного кода, выполняющего предварительные и завершающие действия перед входом и после выхода из обернутого блока программного кода. Она в общих чертах напоминает инструкцию `try/finally`, так как тоже выполняет действия на выходе независимо от того, возникло исключение или нет, но в отличие от последней позволяет определять действия на входе и на выходе, используя для этого протокол, основанный на использовании объектов.

28

Объекты исключений

До сих пор я преднамеренно умалчивал о том, *чем* в действительности являются исключения. В языке Python понятие исключения было обобщено – как уже упоминалось в предыдущей главе, они могут идентифицироваться строковыми объектами или объектами экземпляра класса. В настоящее время предпочтительнее использовать объекты экземпляров класса, но в скором будущем они станут обязательными. У обоих подходов имеются свои достоинства, но классы обеспечивают лучшее решение, когда дело доходит до поддержки иерархий исключений.

Проще говоря, исключения на основе классов позволяют создавать исключения, организованные в категории, включающие информацию о состоянии и поддерживающие наследование. Если говорить более подробно, по сравнению со строками, классы исключений:

- Лучше поддерживают возможные изменения в будущем – добавление новых исключений в будущем вообще не будет требовать изменений в инструкциях `try`.
- Предоставляют естественное место для хранения информации, доступной для обработчиков в инструкции `try`. Они могут включать как информацию о состоянии, так и методы, доступные через экземпляры класса.
- Позволяют исключениям принимать участие в иерархиях наследования с целью обладания общим поведением – наследовать методы отображения, например, чтобы обеспечить единый стиль сообщений об ошибках.

Вследствие этих отличий исключения в виде классов лучше поддерживают возможность развития программ и крупных систем, чем исключения на основе строк. Строковые исключения на первый взгляд выглядят более простыми в использовании, пока программы имеют

небольшой размер, но пользоваться ими становится значительно сложнее по мере роста размеров программ. В действительности, по причинам, указанным выше, все встроенные исключения идентифицируются классами и организованы в виде дерева наследования. Вы можете избрать такой же подход при создании своих собственных исключений.

В интересах обратной совместимости я представлю здесь и строковые исключения, и исключения на основе классов. В настоящее время могут использоваться оба вида исключений, однако строковые исключения генерируют предупреждения о нежелательности их использования в текущей версии Python (2.5) и не будут поддерживаться в версии Python 3.0. Мы будем их рассматривать, потому что они наверняка будут встречаться вам в существующем программном коде, но новые исключения, которые вам придется определять, уже в настоящее время должны оформляться в виде классов отчасти потому, что реализация на основе классов обладает неоспоримыми преимуществами, а отчасти потому, что вам едва ли захочется вносить изменения в свой программный код после выхода Python 3.0.

Исключения на основе строк

Во всех примерах, что мы видели до сих пор, исключения, определяемые программой, были оформлены в виде строк. Это простейший способ создать исключение. Например:

```
>>> myexc = "My exception string"
>>> try:
...     raise myexc
... except myexc:
...     print 'caught'
...
Caught
```

Любое строковое значение может использоваться для идентификации исключения. С технической точки зрения исключение идентифицируется строковым *объектом*, а не *значением* – вы должны использовать одно и то же имя переменной (то есть ссылку) при возбуждении и при перехвате исключения (подробнее об этой идее я расскажу в разделе, описывающем типичные проблемы, в конце седьмой части). В этом примере исключение `myexc` – это обычная переменная, она может импортироваться другими модулями, и т. д. Текст строки практически не играет никакой роли за исключением того, что он выводится как текст сообщения об ошибке:

```
>>> raise myexc
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
My exception string
```



Если учесть, что текст строковых исключений может выводиться на экран, вы наверняка предпочтете использовать более осмысленный текст, чем в примерах, показанных в этой книге.

Строковые исключения уходят в прошлое!

Как уже упоминалось ранее, строковые исключения по-прежнему можно использовать, но в версии Python 2.5 они генерируют предупреждения и, как предполагается, вообще не будут поддерживаться в Python 3.0, если не раньше. Ниже приводится истинный результат работы предыдущего фрагмента в среде разработки IDLE, входящей в состав Python 2.5:

```
>>> myexc = 'My exception string'
>>> try:
    raise myexc
except myexc:
    print 'caught'

Warning (from warnings module):
  File "__main__", line 2
DeprecationWarning: raising a string exception is deprecated
(DeprecationWarning: возбуждение строковых исключений не приветствуется)
caught
```

Вы можете запретить вывод таких предупреждений, но они выводятся для того, чтобы дать вам знать, что строковые исключения в будущем будут рассматриваться как ошибка и потому станут недопустимыми. В этой книге строковые исключения описываются лишь для того, чтобы дать вам возможность понимать программный код, написанный в прошлом; в настоящее время все встроенные исключения являются экземплярами классов, и все исключения, определяемые в программе, также должны создаваться на основе классов. В следующем разделе объясняется – почему.

Исключения на основе классов

Строки обеспечивают самый простой способ определения исключений. Однако, как описывалось ранее, классы предоставляют дополнительные преимущества, которые заслуживают того, чтобы познакомиться с ними. Наиболее важное 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 поэтому в скором будущем будут представлять единственную возможность определения новых исключений.

Помимо отличий в программном коде главное различие между строковыми исключениями и исключениями на базе классов заключается

в способе идентификации возбужденных исключений в предложениях `except` инструкции `try`:

- Строковые исключения идентифицируются по *идентичности объекта*: идентификация возбужденного исключения в предложении `except` выполняется с помощью оператора `is` (а не `==`).
- Исключения на основе классов идентифицируются *отношением наследования*: 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 становятся различными видами исключений внутри категории. Используя имя общего суперкласса, предложение `except` сможет перехватывать целую категорию исключений – каждый конкретный подкласс будет соответствовать этому предложению.

В дополнение к этой идее исключения на основе классов обеспечивают лучшую поддержку *информации о состоянии* (присоединенной к экземплярам), и позволяют исключениям принимать участие в *иерархиях наследования* (с целью обрести общие черты поведения). Они представляют собой более мощную альтернативу строковым исключениям при незначительном увеличении объемов программного кода.

Пример исключения-класса

Давайте рассмотрим на примере программного кода, как работают исключения-классы. В следующем файле `classex.py` определяется суперкласс с именем `General` и два подкласса с именами `Specific1` и `Specific2`. Этот пример иллюстрирует понятие категорий исключений, где `General` – это имя категории, а два подкласса – это определенные типы исключений внутри категории. Обработчики, которые перехватывают исключение `General`, также будут перехватывать и все его подклассы, в том числе `Specific1` и `Specific2`:

```
class General: 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:\python> python classexc.py
caught: __main__.General
caught: __main__.Specific1
caught: __main__.Specific2
```

Мы еще вернемся к используемой здесь функции `sys.exc_info` в следующей главе – с ее помощью можно получить информацию о самом последнем исключении. А пока коротко замечу, что первый элемент результата, возвращаемого функцией, – это имя класса возбужденного исключения, а второй – фактический экземпляр класса исключения. Кроме этого метода нет никакого другого способа определить точно, что произошло, в пустом предложении `except`, которое перехватывает все исключения.

Обратите внимание, что здесь для использования в инструкциях `raise` создаются *экземпляры* классов – как будет показано ниже в этом разделе, когда будет приводиться формализованное описание инструкции `raise`, при возбуждении исключений на основе классов всегда используются экземпляры. В этом фрагменте также присутствуют функции, которые возбуждают исключения всех трех классов, а кроме того на верхнем уровне имеется инструкция `try`, в которой производится вызов функций и осуществляется перехват исключения `General` (та же самая инструкция `try` перехватывает и два более специфичных исключения, потому что они являются подклассами класса `General`).

Еще одно замечание: текущая документация по языку Python указывает, что предпочтительнее (но не обязательно) создавать свои собственные классы исключений, наследуя встроенный класс исключения с именем `Exception`. Для этого нам придется переписать первую строку файла `classexc.py`, как показано ниже:

```
class General(Exception): pass
class Specific1(Generic): pass
class Specific2(Generic): pass
```

Хотя это не является обязательным требованием, и в настоящее время отдельные классы прекрасно справляются с ролью исключений, тем не менее, этот предпочтительный способ со временем станет обязательным. Если вы хотите, чтобы ваш программный код был готов к будущим изменениям в языке, создавайте свои корневые суперклассы, наследуя класс `Exception`, как показано здесь. Кроме всего прочего, при

таком подходе ваши классы приобретают по наследству некоторые полезные интерфейсы. Например, класс `Exception` имеет конструктор `__init__`, который автоматически присоединяет к экземплярам свои аргументы.

В чем преимущества классов исключений?

Поскольку в примере предыдущего раздела имеется всего три возможных исключения, 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 строковых исключений в круглых скобках. Как это можно сделать, показано в файле *stringexc.py*:

```
General = 'general'
Specific1 = 'specific1'
Specific2 = 'specific2'

def raiser0(): raise General
def raiser1(): raise Specific1
def raiser2(): raise Specific2

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except (General, Specific1, Specific2): # Перехватывает любые из этих
        import sys
        print 'caught:', sys.exc_info()[0]
```

C:\python> python stringexc.py
caught: general
caught: specific1
caught: specific2

Однако в случае больших или глубоких иерархий 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, добавляя 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 ситуации, которые могут приводить к таким ошибкам, как деление на ноль и переполнение. Вы описываете эти ошибки как исключения, которые могут возбуждаться этой библиотекой, и определяете их как простые строки:

```
# mathlib.py

divzero = 'Division by zero error in library'
oflow   = 'Numeric overflow error in library'
...
```

```
def func():  
    ...  
    raise divzero
```

Теперь те, кто будет использовать вашу библиотеку, будут стремиться обертывать вызовы ваших функций или классов инструкцией `try`, чтобы перехватывать два ваших исключения (если они не будут перехватывать их, эти исключения будут приводить к аварийному завершению программ):

```
# client.py  
  
import mathlib  
...  
try:  
    mathlib.func(...)  
except (mathlib.divzero, mathlib.oflow):  
    ...вывод сообщения и восстановление после ошибки...
```

Все работает просто замечательно и многие начинают использовать вашу библиотеку. Однако шесть месяцев спустя вы обнаруживаете еще одну ситуацию, которая может приводить к другой ошибке – потере значимых разрядов, после чего добавляете новое строковое исключение:

```
# mathlib.py  
divzero = 'Division by zero error in library'  
oflow   = 'Numeric overflow error in library'  
uflow   = 'Numeric underflow error in library'
```

К сожалению, выпуском новой версии своей библиотеки вы создаете проблему для тех, кто ею пользуется. Если они явно указывали имена ваших исключений, теперь им придется вернуться к своим программам и внести соответствующие 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):  
    ...вывод сообщения и восстановление после ошибки...
```

Вероятно, это не конец света. Если ваша библиотека предназначена исключительно 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 свой программный код всякий раз,

когда вы изменяете свой набор исключений, такая политика обновления определенно не будет расцениваться как самая вежливая.

Ваши пользователи могут попытаться избежать этой ловушки, определяя пустые предложения `except`, которые перехватывают все исключения:

```
# client.py

try:
    mathlib.func(...)
except:                # Перехватывать все исключения
    ...вывод сообщения и восстановление после ошибки...
```

Но при таком решении могут перехватываться посторонние исключения, даже такие, которые вызваны опечатками в именах переменных, ошибками работы с памятью, и исключения, генерируемые программой при завершении, а для вас было бы нежелательно, чтобы перехваченные исключения ошибочно классифицировались как ошибки в библиотеке. Как правило, при обработке исключений чем больше определенности, тем лучше (к этой идее мы еще вернемся в разделе с описанием типичных проблем, в следующей главе).¹

Так как же быть? Исключения на основе классов полностью ликвидируют эту проблему. Вместо того чтобы определять библиотечные исключения как простой набор строк, их можно оформить в виде дерева классов с одним общим суперклассом, охватывающим целую категорию исключений:

```
# mathlib.py

class NumErr(Exception): pass
class Divzero(NumErr): pass
class Oflow(NumErr): pass
...
def func():
    ...
    raise DivZero()
```

¹ Как было предложено одним моим сообразительным студентом, в модуле библиотеки можно было бы определить кортеж, содержащий все исключения, которые могут быть возбуждены библиотекой. Тогда клиент мог бы импортировать этот кортеж и использовать его имя в предложении `except`, чтобы перехватывать все библиотечные исключения (вспомните, что при использовании кортежа в предложении `except` будут перехватываться *все* перечисленные в нем исключения). Когда позднее в библиотеку добавляется новое исключение, можно просто расширять 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 и используют механизм наследования.

При таком подходе пользователям вашей библиотеки достаточно будет указать общий суперкласс (то есть категорию), чтобы перехватывать все исключения, возбуждаемые библиотекой, причем, как существующие, так и те, что появятся 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
```

В результате программный код пользователей, перехватывающий исключения 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 беспокоиться об изменениях в вашем наборе исключений. Другими словами, исключения на основе классов лучше отвечают требованиям сопровождения, чем строки. Кроме того, исключения на основе классов могут поддерживать хранение информации о состоянии и наследование, чего не скажешь о строках — эти концепции мы будем исследовать на примере ниже в этой главе.

Классы встроенных исключений

Примеры в предыдущем разделе в действительности возникли не на пустом месте. Исключения, определяемые программой, могут быть представлены как в виде строк, так и в виде классов, однако все встроенные исключения, которые могут возбуждаться самим интерпретатором Python, представляют собой объекты классов, а не строки. Кроме того, они организованы в неглубокую иерархию с общими суперклассами категорий и подклассами определенных типов исключений, практически так же, как в примере выше.

Все знакомые исключения, с которыми нам уже приходилось встречаться (например, `SyntaxError`), в действительности являются обычными классами, доступными в виде встроенных имен (в модуле `__builtin__`) и в виде атрибутов модуля `exceptions`, входящего в состав стандартной библиотеки. Кроме того, в языке Python встроенные исключения организованы в иерархию с целью поддержки различных режимов перехвата исключений. Например:

Exception

Корневой суперкласс всех исключений.

StandardError

Суперкласс всех встроенных исключений.

ArithmeticError

Суперкласс всех арифметических ошибок.

OverflowError

Подкласс, идентифицирующий конкретную арифметическую ошибку.

И так далее. Подробнее познакомиться с этой структурой можно либо в справочном руководстве по библиотеке Python, либо в тексте справки в модуле `exceptions` (описание функции `help` приводится в главах 4 и 14):

```
>>> import exceptions
>>> help(exceptions)
...объемный текст справки опущен...
```

Дерево встроенных классов позволяет определять, насколько конкретными или универсальными будут ваши обработчики исключений. Например, встроенное исключение `ArithmeticError` — это суперкласс для таких более конкретных исключений, как `OverflowError` и `ZeroDivisionError`. Указав имя `ArithmeticError` в инструкции `try`, вы будете перехватывать все арифметические ошибки, а указав имя `OverflowError`, вы будете перехватывать только ошибки определенного типа и никакие другие.

Точно так же можно использовать `StandardError`, имя суперкласса всех встроенных исключений, для организации выбора между встроенными исключениями и исключениями, определяемыми программой, например:

```
try:
    try:
        action()
    except StandardError:
        ...обработка встроенных исключений...
except:
    ... обработка исключений, определяемых программой...
else:
    ...обработка случая отсутствия исключения...
```

Используя имя корневого класса `Exception`, можно очень близко имитировать поведение пустого предложения `except` (которое перехватывает любые исключения). Однако это будет не полная имитация, так как в этом случае не будут перехватываться строковые исключения и исключения на основе отдельных классов, которые пока не обязательно должны быть подклассами корневого класса `Exception`.

Неважно, будете вы использовать категории в дереве встроенных классов или нет, этот подход служит отличным примером; при использовании подобных методов к созданию своих собственных исключений вы сможете реализовать гибкие наборы исключений, которые легко можно изменять.

Во всех других отношениях встроенные исключения практически неотличимы от первоначальной модели строковых исключений. Обычно вам даже не придется беспокоиться по поводу имеющихся различий, если, конечно, вы не исходите из предположения, что встроенные исключения являются строками, и не пытаетесь использовать их в операции конкатенации без необходимого преобразования (например, выражение `KeyError + "spam"` будет приводить к ошибке, а выражение `str(KeyError) + "spam"` вполне допустимо).

Определение текста исключения

Когда в начале главы мы знакомились с исключениями на основе строк, мы видели, что текст строки выводится в стандартный поток вывода сообщений об ошибках, когда исключение не обрабатывается программой (то есть, когда исключение достигает обработчика по умолчанию). Но какое сообщение будет выводиться в случае исключений-классов? По умолчанию выводится имя класса и не очень удобочитаемая информация об объекте экземпляра:

```
>>> class MyBad: pass
>>> raise MyBad()

Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    raise MyBad()
MyBad: <__main__.MyBad instance at 0x00BB5468>
```

Чтобы улучшить сообщение, необходимо переопределить в классе исключения метод `__repr__` или `__str__`, чтобы возвращалась желаемая строка, которая будет отображаться, когда ваше исключение будет достигать обработчика по умолчанию:

```
>>> class MyBad:
...     def __repr__(self):
...         return "Sorry--my mistake!"
...
>>> raise MyBad()

Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    raise MyBad()
MyBad: Sorry--my mistake!
```

Как мы уже знаем, метод перегрузки `__repr__`, используемый здесь, вызывается при выводе и при преобразовании экземпляра класса в строку; метод `__str__` реализует дружественное строковое представление,

которому инструкции `print` отдают предпочтение. (Подробнее о методах преобразования в строку рассказывается в разделе «Перегрузка операторов» в главе 24.)

Обратите внимание, что если наследовать встроенные классы исключений, как рекомендовалось выше, текст сообщения об ошибке немного изменится – аргументы конструктора автоматически сохраняются в экземпляре и отображаются в тексте сообщения:

```
>>> class MyBad(Exception): pass
>>> raise MyBad()

Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    raise MyBad()
MyBad

>>> class MyBad(Exception): pass
>>> raise MyBad('the', 'bright', 'side', 'of', 'life')

Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    raise MyBad('the', 'bright', 'side', 'of', 'life')
MyBad: ('the', 'bright', 'side', 'of', 'life')
```

Если ваши конечные пользователи могут видеть сообщения об ошибках, порождаемые исключениями, у вас наверняка появится желание определить собственные методы форматирования сообщений на основе перегрузки операторов, как показано здесь. Возможность автоматического присоединения информации о состоянии к экземплярам – очень удобная особенность, о чем подробнее рассказывается в следующем разделе.

Передача данных и поведения в экземплярах

Помимо поддержки гибких иерархий классы исключений также являются удобным местом для хранения дополнительной информации в виде атрибутов экземпляров. Когда возбуждается исключение на основе класса, вместе с исключением интерпретатор автоматически передает объект экземпляра класса в виде элемента дополнительных данных. Так же, как и в случае строковых исключений, вы можете получить доступ к экземпляру, указав дополнительную переменную в инструкции `try`. Это обеспечивает естественный способ передачи дополнительных данных и функциональных возможностей обработчику исключения.

Пример: передача дополнительных данных в исключениях на основе классов и строк

Давайте рассмотрим возможность передачи дополнительных данных на примере и попутно сравним подходы, основанные на использовании классов и строк. Программа, выполняющая анализ файлов, может со-

общать об ошибке форматирования, возбуждая экземпляр исключения, который заполняется дополнительной информацией об ошибке:

```
>>> class FormatError:
...     def __init__(self, line, file):
...         self.line = line
...         self.file = file
...
>>> def parser():
...     # когда обнаруживается ошибка
...     raise FormatError(42, file='spam.txt')
...
>>> try:
...     parser()
... except FormatError, 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 возбуждения исключения.¹ Однако с практической точки зрения этот способ не имеет заметных преимуществ перед возможностью передачи составных объектов (например, кортежей, списков или словарей) в виде дополнительных данных строковых исключений, и сам по себе не выглядит достаточным побудительным мотивом к использованию исключений на основе классов. Ниже приводится эквивалентный фрагмент, в котором используются строковые исключения:

```
>>> formatError = 'formatError'

>>> def parser():
...     # когда обнаруживается ошибка
...     raise formatError, {'line':42, 'file':'spam.txt'}
...
>>> try:
...     parser()
... except formatError, X:
...     print 'Error at', X['file'], X['line']
...
Error at spam.txt 42
```

На этот раз переменной `X` в предложении `except` присваивается словарь с дополнительной информацией, который передается инструкции

¹ Как будет показано в следующей главе, доступ к объекту экземпляра класса исключения обеспечивается также вторым элементом кортежа, возвращаемого функцией `sys.exc_info` — инструментом, который возвращает информацию о самом последнем исключении. Если в предложении `except` не указано имя переменной, можно использовать эту функцию, когда возникает необходимость обратиться к исключению за получением присоединенных к нему данных или для вызова его методов.

`raise`. Результат получается тот же самый, но при этом нет необходимости писать определение класса. Однако подход, основанный на использовании классов, может оказаться более удобным, когда исключение должно обладать еще и поведением. Класс исключения определяет еще и *методы*, которые можно вызывать из обработчика:

```
class FormatError:
    def __init__(self, line, file):
        self.line = line
        self.file = file
    def logerror(self):
        log = open('formaterror.txt', 'a')
        print >> log, 'Error at', self.file, self.line

def parser():
    raise FormatError(40, 'spam.txt')

try:
    parser()
except FormatError, exc:
    exc.logerror()
```

При использовании классов методы (такие как `logerror`) могут наследоваться подклассами, а атрибуты экземпляра (такие как `line` и `file`) предоставляют возможность сохранения информации о состоянии, обеспечивая дополнительный контекст для последующих вызовов методов. Мы могли бы имитировать этот эффект, передавая простые функции вместе со строковыми исключениями, но это приведет к чрезмерному усложнению программного кода:

```
formatError = "formatError"

def logerror(line, file):
    log = open('formaterror.txt', 'a')
    print >> log, 'Error at', file, line

def parser():
    raise formatError, (41, 'spam.txt', logerror)

try:
    parser()
except formatError, data:
    data[2](data[0], data[1]) # Или просто: logerror()
```

Естественно, такие функции не могут быть унаследованы как методы класса и не способны хранить информацию о состоянии в атрибутах экземпляра класса (`lambda`-выражения и глобальные переменные — это самое лучшее, что можно использовать с функциями). Конечно, можно было бы передать экземпляр класса в виде дополнительных данных строковых исключений, чтобы добиться того же самого эффекта, но если мы зашли так далеко, чтобы имитировать исключения на основе классов, лучше уж принять их — нам и так пришлось писать определение класса.

Как уже упоминалось ранее, исключения на основе классов станут единственно возможными в будущей версии Python. Но даже если бы это было не так, существуют веские основания использовать их. Вообще говоря, строковые исключения представляют собой простой инструмент, предназначенный для решения простых задач. Исключения на основе классов полезны для определения категорий, и они выглядят предпочтительнее в более сложных приложениях, когда можно извлечь выгоду из возможности сохранять информацию о состоянии и наследовать атрибуты. Не в каждом приложении требуется мощь ООП, но преимущества исключений на основе классов становятся более очевидными с ростом и расширением программ.

Общие формы инструкции raise

С учетом исключений на основе классов инструкция `raise` может принимать следующие пять форм. Первые две возбуждают строковые исключения, следующие две – исключения на основе классов, и последняя повторно возбуждает текущее исключение (что очень удобно, когда необходимо обеспечить дальнейшее распространение произвольных исключений):

```
raise string          # Соответствует предложению except с тем же объектом
raise string, data    # Передает дополнительные данные (по умолчанию = None)

raise instance        # То же, что и: raise instance.__class__, instance
raise class, instance # Соответствует предложению except с тем же
                     # именем класса или его суперкласса

raise                 # Повторно возбуждает текущее исключение
```

В наши дни наиболее часто используется третья форма. В случае исключений на основе классов Python всегда требует указывать экземпляр класса. Возбуждая экземпляр, интерпретатор в действительности возбуждает исключение, соответствующее классу экземпляра, а экземпляр передается вместе с исключением в виде элемента дополнительных данных (как мы уже видели, это отличное место для хранения информации, которая может использоваться обработчиком). Для обратной совместимости с версиями Python, в которых исключения реализованы на основе строк, можно использовать следующие формы инструкции `raise`:

```
raise class           # То же, что и: raise class()
raise class, arg       # То же, что и: raise class(arg)
raise class, (arg, arg, ...) # То же, что и: raise class(arg, arg, ...)
```

Все они соответствуют форме `raise class(arg, ...)` и, следовательно, соответствуют форме `raise instance` выше. В частности, если вместо экземпляра в инструкции `raise` указывается класс, и дополнительный элемент данных не является экземпляром указанного класса, интерпретатор автоматически вызовет класс с дополнительными элементами данных в качестве аргументов конструктора, чтобы создать и возбудить экземпляр класса исключения.

Например, можно возбудить экземпляр встроенного исключения `KeyError`, просто записав инструкцию `raise KeyError`: даже при том, что `KeyError` в настоящее время является классом, – интерпретатор вызовет `KeyError`, чтобы создать необходимый экземпляр. Фактически исключение `KeyError` и любые другие исключения на основе классов можно возбудить различными способами:

```
raise KeyError()           # Обычная форма: возбуждается экземпляр
raise KeyError, KeyError() # Класс, экземпляр: используется экземпляр
raise KeyError             # Класс: будет создан экземпляр
raise KeyError, "bad spam" # Класс, аргумент: будет создан экземпляр
```

Для всех этих пяти форм инструкция `try`, имеющая следующую форму:

```
try:
    ...
except KeyError, X:
    ...
```

присвоит переменной `X` объект экземпляра класса `KeyError`.

Если это объяснение показалось вам малопонятным, просто запомните, что исключения могут идентифицироваться строкой или экземпляром класса. В случае строк вместе с исключением можно передавать дополнительные данные. В случае классов, если объект экземпляра не передается инструкции `raise`, интерпретатор создаст его автоматически.

В версии Python 2.5 можно вообще отказаться от использования строковых форм инструкции `raise`, потому что при возбуждении строковых исключений генерируются предупреждения, а, кроме того, они будут недоступны в следующей версии Python. Но, к сожалению, обратная совместимость все еще принимается во внимание в книгах, которые обучают языку программирования, используемому более чем одним миллионом человек!

В заключение

В этой главе мы занялись созданием собственных исключений. Здесь мы узнали, что исключения могут быть реализованы как строковые объекты или как экземпляры классов. Однако в настоящее время предпочтение следует отдавать экземплярам классов, а в будущей версии Python классы станут единственно возможным способом реализации исключений. Классы исключений предпочтительнее потому, что они поддерживают концепцию создания иерархий исключений (что положительно сказывается на удобстве сопровождения), позволяют присоединять к исключениям дополнительные данные и поведение в виде атрибутов и методов экземпляров, а также обеспечивают наследование атрибутов и методов от суперклассов.

Мы видели, что перехватывая суперкласс в инструкции `try`, мы перехватываем этот класс, а также все его подклассы, расположенные ни-

же в дереве наследования; суперклассы начинают играть роль названий категорий, а подклассы становятся определенными типами исключений в этих категориях. Мы также видели, что инструкция `raise` поддерживает несколько форматов, хотя большинство современных программ просто создают и возбуждают экземпляры классов.

В этой главе мы исследовали обе альтернативы – исключения на основе строк и на основе классов. Тем не менее, объекты исключений проще запомнить, если ограничиться только рекомендуемой моделью, основанной на использовании классов, – описывайте свои исключения как классы, наследуйте класс `Exception` как вершину своих деревьев исключений, – и вы сможете забыть старую альтернативу на основе строк.

Следующая глава завершает эту часть книги и всю книгу в целом исследованием некоторых типичных случаев использования исключений и рассмотрением инструментов, наиболее часто используемых программистами на языке Python. Однако, прежде чем двинуться дальше, ответьте на контрольные вопросы к этой главе.

Закрепление пройденного

Контрольные вопросы

1. Как определяется соответствие строковых исключений и обработчиков?
2. Как определяется соответствие исключений на основе классов и обработчиков?
3. Как можно присоединить контекстную информацию к исключениям на основе классов и как ее можно использовать в обработчиках?
4. Как можно определить текст сообщения об ошибке в исключениях на основе классов?
5. Почему в настоящее время нежелательно использовать исключения на основе строк?

Ответы

1. Соответствие строковых исключений выявляется по идентичности объекту (технически, с помощью оператора `is`), а не по значению объекта (оператор `==`). Поэтому будет недостаточно использовать то же самое значение – необходимо иметь ссылку на тот же самый объект (обычно переменную). Короткие строки в языке Python кэшируются с целью многократного использования, поэтому использование одного и того же значения может иногда работать, но вы не должны полагаться на это (подробнее об этой проблеме будет рассказываться в конце следующей главы).
2. Соответствие исключений на основе классов определяется отношением к суперклассу: при использовании имени суперкласса в обра-

ботчике исключения будут перехватываться экземпляры этого класса, а также экземпляры всех его подклассов, расположенных ниже в дереве наследования. Благодаря этому суперклассы можно интерпретировать как категории исключений, а подклассы – как более специфичные типы исключений в этих категориях.

3. Присоединение дополнительной информации к исключениям на основе классов производится путем заполнения атрибутов объекта экземпляра исключения, часто внутри конструкторов классов. В обработчиках исключений указывается переменная, которой присваивается 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 можно определить с помощью метода перегрузки `__repr__` или `__str__`. Если вы наследуете свои классы от встроенного класса `Exception`, в тексте сообщения автоматически будут отображаться все аргументы, переданные конструктору.
5. Потому что, как заявил Гвидо (Guido), в будущей версии Python планируется вообще убрать их. На самом деле, для этого есть весьма серьезные основания: строковые исключения не поддерживают деление на категории, не позволяют присоединять информацию о состоянии или наследовать поведение, как исключения на основе классов. С практической точки зрения строковые исключения проще в использовании на первых порах, пока программы достаточно маленькие, но их становится сложно использовать, как только программы становятся больше.

29

Использование исключений

Данная глава завершает эту часть книги рассмотрением некоторых тем, связанных с проектированием исключений и примеров их использования. Далее следует раздел с описанием типичных проблем и упражнения. 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 разработчика приложений на языке Python.

Вложенные обработчики исключений

До сих пор в наших примерах для перехвата исключений использовалась единственная инструкция `try`, но что произойдет, если одну инструкцию `try` вложить внутрь другой? И, раз уж на то пошло, что произойдет, если в инструкции `try` вызывается функция, которая выполняет другую инструкцию `try`? С технической точки зрения инструкции могут вкладываться друг в друга как синтаксически, так и по пути следования потока управления через программный код.

Оба эти варианта проще будет понять, если вы узнаете, что интерпретатор складывает инструкции `try` *стопкой* во время выполнения. Когда возникает исключение, интерпретатор возвращается к самой последней инструкции `try`, содержащей соответствующее предложение `except`. Поскольку каждая инструкция `try` оставляет метку, интерпретатор может возвращаться к более ранним инструкциям `try`, двигаясь по стопке меток. Такое вложение активных обработчиков и есть то, что подразумевается, когда мы говорим о распространении исключений вверх, к обработчикам «более высокого уровня». Эти обработчики являются обычными инструкциями `try`, в которые поток управления ходом выполнения программы вошел раньше.

Рисунок 29.1 иллюстрирует, что происходит, когда возникает вложение инструкций `try/except` во время выполнения. Объем программного кода, который выполняется в инструкции `try`, может оказаться весьма существенным (например, он может содержать вызовы функций) и нередко вызывает другой программный код, который готов перехватить те же самые исключения. Когда исключение наконец возбуждается, интерпретатор переходит к самой последней инструкции `try`, в которой указано имя исключения, запускает блок `except` и продолжает выполнение программы ниже этой инструкции `try`.

Как только такое исключение будет перехвачено, его жизнь заканчивается – управление не передается *всем* соответствующим инструкциям `try`, содержащим имя исключения, – только первая из них получает возможность обработать исключение. Например, на рис. 29.1 инструкция `raise` в функции `func2` возвращает управление обработчику в функции `func1`, после чего программа продолжает выполнение внутри `func1`.

Напротив, когда исключение возникает во вложенных инструкциях `try/finally`, выполняется каждый блок `finally` по очереди – интерпретатор продолжает передавать исключение вверх по цепочке вложенных инструкций `try`, пока не будет достигнут обработчик по умолчанию верхнего уровня (который выводит стандартные сообщения об ошибках). Как показано на рис. 29.2, предложения `finally` не останавливают распространение исключений – они лишь определяют программный код, который должен выполняться на выходе из инструкции `try` в процессе движения исключения. Если к моменту возникновения исключения имелось несколько активных инструкций `try/finally`, они *все* будут выполнены, если где-то на пути исключения не встретится инструкция `try/except`, которая перехватит его.

Другими словами, куда будет выполнен переход при возникновении исключения, полностью зависит от того, *где оно возникло*, – это определяется ходом выполнения программы, а не только синтаксисом.

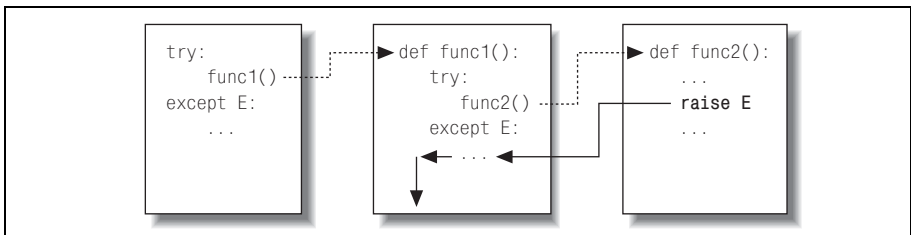


Рис. 29.1. Вложенные инструкции `try/except`: когда возбуждается исключение (программой или интерпретатором), происходит возврат к самой последней инструкции `try` с соответствующим предложением `except` и программа продолжает выполнение после этой инструкции `try`. Предложения `except` перехватывают и останавливают дальнейшее распространение исключений – это место, где выполняются восстановительные операции после исключения

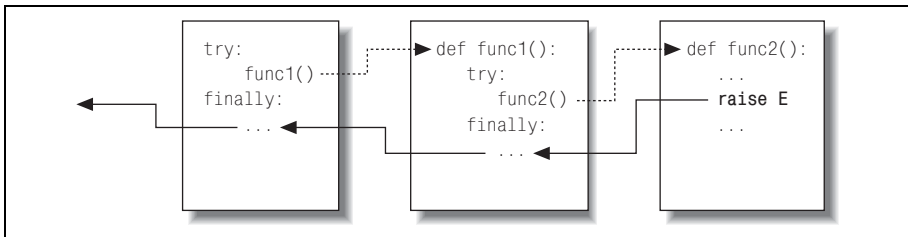


Рис. 29.2. Вложенные инструкции `try/finally`: когда возбуждается исключение, управление возвращается самой последней инструкции `try` и выполняется ее блок `finally`, после этого исключение продолжит свое распространение по блокам `finally` во всех активных инструкциях `try`, пока в конечном счете не будет достигнут обработчик по умолчанию, где производится вывод сообщения об ошибке. Предложения `finally` перехватывают (но не останавливают) исключения – они определяют действия, которые должны выполняться «на выходе»

Распространение исключения, по сути, происходит в порядке, обратном вхождению в инструкции `try`. Это движение останавливается, когда управление переходит к соответствующему блоку `except`, и продолжается когда управление проходит через предложения `finally`.

Пример: вложение в потоке управления

Обратимся к примеру, чтобы рассмотреть этот тип вложения более конкретно. В следующем файле модуля `nestexc.py` определяются две функции. Функция `action2` возбуждает исключение (нельзя складывать числа и последовательности), функция `action1` обертывает вызов функции `action2` в инструкцию `try`, которая 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, вызов функции `action1` также обернут инструкцией `try`. Когда функция `action2` возбуждает исключение `TypeError`, существует две активные ин-

струкции `try` – одна в функции `action1` и одна в программном коде на верхнем уровне модуля. 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 исключения, зависит от того, в каком месте программы находится поток управления. Поэтому, чтобы знать, куда будет выполнен переход, необходимо знать место, где находится управление. В данном случае выбор места, где будет обработано исключение, больше зависит от того, где находится поток управления, чем от синтаксиса. Однако мы можем организовать синтаксическое вложение обработчиков – эквивалентный случай рассматривается в следующем разделе.

Пример: синтаксическое вложение

В главе 27, когда рассматривалась новая объединенная инструкция `try/except/finally`, я уже говорил, что вполне возможно вкладывать инструкции `try` синтаксически, 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 обработчиков, что и предыдущий пример (и ведущую себя точно так же). Фактически инструкции, вложенные синтаксически, работают точно так же, как показано на рис. 29.1 и 29.2; единственное отличие заключается в том, что вложенные обработчики физически объединены в блоке инструкции `try`, а не находятся в разных функциях. Например, исключение пройдет через все блоки `finally` независимо от того, вложены 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 ?
IndexError
```

Графическая иллюстрация порядка выполнения этого фрагмента показана на рис. 29.2 – результат получается тот же самый, но сама логика выполнения в данном случае образована вложенными инструкциями. Более интересный пример синтаксического вложения в действии 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
    try:
        func()
    except IndexError:
        print 'caught IndexError'
    finally:
        print 'finally run'
```

Этот фрагмент перехватывает исключение, если оно будет возбуждено, и выполняет завершающие действия в блоке `finally` независимо от того, возникло исключение или нет. Чтобы понять это, может потребоваться некоторое время на изучение фрагмента, но результат очень напоминает объединение предложений `except` и `finally` в одной современной инструкции `try` (не забывайте, что такая инструкция была недоступимой до появления версии Python 2.5):

```
% python except-finally.py

<function raise1 at 0x00BA2770>
caught IndexError
finally run

<function noraise at 0x00BB47F0>
finally run

<function raise2 at 0x00BB4830>
finally run

Traceback (most recent call last):
  File "C:/Python25/except-finally.py", line 9, in <module>
    func()
  File "C:/Python25/except-finally.py", line 3, in raise2
    def raise2(): raise SyntaxError
SyntaxError: None
```

Как мы видели в главе 27, начиная с версии Python 2.5, появилась возможность использовать предложения `except` и `finally` в одной инструкции `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 альтернативных конструкций обработки исключений.

Идиомы исключений

Мы рассмотрели внутренний механизм исключений. Теперь рассмотрим некоторые другие типичные способы их использования.

Исключения не всегда являются ошибками

В языке Python все ошибки являются исключениями, но не все исключения являются ошибками. Например, в главе 9 мы видели, что по достижении конца файла метод чтения объекта файла возвращает пустую строку. Напротив, встроенная функция `raw_input` (с которой мы впервые встретились в главе 3 и которую 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 отличие от методов объекта файла данная функция не возвращает пустую строку; пустая строка, полученная от функции `raw_input`, означает всего лишь пустую строку. Несмотря на свое название, исключение `EOFError` в данном контексте — это всего лишь сигнал, а не ошибка. По этой причине, чтобы избежать преждевременного завершения работы сценария, функцию `raw_input` оборачивают инструкцией `try`, которую вкладывают в цикл, как показано ниже:

```
while 1:
    try:
        line = raw_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 сигналами, а не ошибками. В языке Python имеется также ряд встроенных исключений, которые являются скорее *предупреждениями*, чем ошибками. Некоторые из них применяются, чтобы сообщить о нежелательности использования некоторых особенностей языка (которые вскоре будут удалены). За дополнительной информацией по предупреждениям обращайтесь к описанию встроенных исключений в руководстве по стандартной библиотеке и к модулю `warnings`.

Передача сигналов из функций по условию

Исключения, определяемые программой, также могут служить сигналами об условиях, которые не являются ошибками. Например, процедура поиска может предусматривать возбуждение исключения в случае нахождения соответствия вместо того, чтобы возвращать флаг состояния, который должен интерпретироваться вызывающей программой. 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 может с успехом использоваться для любой функции, которая не может вернуть специальный признак, свидетельствующий об успехе или неудаче. Например, если любое возвращаемое значение является допустимым, невозможно выбрать какое-то одно значение, которое сигнализировало бы о необычных состояниях. Исключения обеспечивают способ подать сигнал, не возвращая значение:

```
class Failure(Exception): pass

def searcher():
    if ...успех...:
        return ...найденный_элемент...
    else:
        raise Failure()

try:
    item = searcher()
except Failure:
    ...сообщение о неудаче...
else:
    ...обработка найденного элемента...
```

Поскольку язык Python является динамически типизированным и в своей основе поддерживает полиморфизм, исключения, а не возвращение специального признака, являются более предпочтительным способом сообщать о таких состояниях.

Отладка с помощью внешних инструкций try

Обработчики исключений можно также использовать как замену обработчика по умолчанию. Обернув всю программу (или вызов ее) во внешнюю инструкцию `try`, можно перехватывать любые исключения, которые только будут возникать во время работы программы, отменяя тем самым способ завершения программы, заданный по умолчанию.

В следующем фрагменте пустое предложение `except` перехватывает любые необработанные исключения, возникшие в ходе выполнения программы. Чтобы получить доступ непосредственно к самому исклю-

чению, вызовите встроенную функцию `sys.exc_info` из модуля `sys` — она возвращает кортеж, в котором первые два элемента содержат имя исключения и дополнительные данные (если имеются). Для исключений на основе классов эти два элемента представляют имя класса исключения и экземпляр класса возбужденного исключения соответственно (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 позволяет сохранить программу активной даже после ошибки — он позволяет производить дополнительные проверки без необходимости перезапускать программу. Это прием может также использоваться для тестирования 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 тестировать 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 >> log, 'FAILED', testName(), sys.exc_info()[:2]
        else:
            print >> log, 'PASSED', testName()

testdriver()
```

Здесь функция `testdriver` выполняет в цикле серию тестов (модуль `testapi` — некая абстракция в этом примере). Поскольку в обычной ситуации необработанное исключение приводило бы к завершению самого тестового приложения, можно обернуть вызовы очередного теста инструкцией `try`, чтобы обеспечить продолжение процесса тестирования после неудачного завершения любого из тестов. Здесь, как обычно, пустое предложение `except` перехватывает любые необработанные исключения, возникшие в ходе выполнения теста, и регистрирует в файле информацию об исключении, полученную с помощью функции `sys.exc_info`.

Такой подход типичен для систем, которые тестируют функции, модули и классы, запуская их в рамках того же самого процесса, что и само

тестовое приложение. Однако на практике тестирование может оказаться процедурой гораздо более сложной, чем показано здесь. Например, 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 найдете в стандартном руководстве по библиотеке (такие инструменты вообще не возбуждают исключений в случае появления ошибок во внешней программе – фактически тест выполняется параллельно с программой, выполняющей тестирование).

В конце этой главы мы познакомимся с некоторыми законченными платформами, предназначенными для проведения тестов, такими как `Doctest` и `PyUnit`, которые обеспечивают возможность сравнения ожидаемого вывода с фактическими результатами.

Подробнее о функции `sys.exc_info`

Функция `sys.exc_info`, результаты которой использовались в последних двух разделах, является предпочтительным способом доступа к последнему возбужденному исключению. Если в момент ее вызова никакое исключение не обрабатывается, функция возвращает кортеж с тремя объектами `None`. В противном случае возвращаются (*тип*, *значение*, *трассировочная информация*), где:

- *Тип* – это тип обрабатываемого исключения (объект класса для исключений на основе классов).
- *Значение* – это параметр исключения (ассоциированное значение или второй аргумент инструкции `raise`, который всегда является экземпляром класса, если типом исключения является объект класса).
- *Трассировочная информация* – это объект, который представляет стек вызовов в точке, где возникло исключение (в документации к модулю `traceback` описываются инструменты, которые могут использоваться вместе с этим объектом для создания сообщений об ошибках вручную).

Для извлечения типа и значения самого последнего исключения по-прежнему могут использоваться более старые инструментальные средства, такие как `sys.exc_type` и `sys.exc_value`, но они могут использоваться только применительно к единственному исключению, глобальному для всего процесса. В то время как более предпочтительная функция `sys.exc_info` запоминает информацию об исключениях в каждом потоке выполнения. Конечно, это имеет значение только при использовании нескольких потоков выполнения в программах на языке Python (тема, которая далеко выходит за рамки этой книги). За дополнительной информацией обращайтесь к справочному руководству по библиотеке языка Python и к другим специализированным книгам.

Советы по применению исключений

Вообще говоря, исключения в языке Python очень просты в обращении. Настоящее искусство их использования заключается в принятии решения, насколько универсальными должны быть предложения `except` и какой объем программного кода должен быть обернут инструкциями `try`. Рассмотрим сначала вторую проблему.

Что должно быть обернуто

В принципе, можно было бы обернуть каждую инструкцию в сценарии в свою собственную инструкцию `try`, но это будет выглядеть достаточно глупо (тогда инструкции `try` тоже следовало бы обернуть в инструкции `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, которые обычно терпят неудачу. Например, операции, взаимодействующие с системой (открытие файлов, взаимодействия с сокетами и т. д.), являются первыми кандидатами для заключения их в инструкции `try`.
- При этом из первого правила есть исключение – в простых сценариях бывает *желательно*, чтобы подобные неудачи приводили к завершению работы программы. Это особенно верно, когда неудачи ожидаемы. Неудачи в языке Python приводят к выводу полезных сообщений (только не в случае краха программы), и они часто представляют собой лучший результат, на который только можно надеяться.
- Завершающие операции должны заключаться в инструкции `try/finally`, чтобы гарантировать их выполнение. Эта форма инструкции позволяет выполнять программный код независимо от того, возникло исключение или нет.
- Иногда более удобно завернуть вызов крупной функции в единственную инструкцию `try`, чем засорять эту функцию несколькими инструкциями `try`. При таком подходе все исключения, возникшие в функции, будут перехвачены инструкцией `try`, окружающей вызов, за счет чего можно уменьшить объем программного кода внутри самой функции.

Влияние на количество обработчиков исключений нередко оказывает тип программы. Например, серверные программы должны работать постоянно, и поэтому в них инструкции `try` наверняка будут необходимы, чтобы перехватывать исключения и выполнять восстановительные операции после них. В программах тестирования, таких как мы видели в этой главе, также необходимо выполнять обработку исключений. Однако в более простых сценариях часто можно вообще игнорировать 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 позволяет явно указывать, какие исключения должны перехватываться, и иногда бывает необходимо проявлять осторожность, чтобы не перехватывать слишком много. Например, вы уже знаете, что пустое предложение `except` перехватывает *все* исключения, которые только могут возникнуть в блоке `try`.

Сделать это несложно и иногда даже желательно, но это может привести к тому, что будет перехвачена ошибка, обработка которой предусмотрена в инструкции `try` на более высоком уровне вложенной структуры. В предлагаемом примере обработчик исключения перехватывает и деактивирует все исключения, которые достигнут его, независимо от того, ожидает ли какие-либо исключения обработчик 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 исключения, которые вообще не имеют никакого отношения к программе. Даже такие ситуации, как ошибки работы с памятью, настоящие ошибки в программном коде, остановки итераций и выход из программы, возбуждают исключения. Обычно такие исключения не должны перехватываться.

Например, сценарии обычно завершают работу, когда поток управления достигает конца главного файла. Однако в языке Python имеется специальная функция `sys.exit(statuscode)`, с помощью которой можно завершить работу программы. Чтобы завершить программу, эта функция в действительности возбуждает исключение `SystemExit`, благодаря чему имеется возможность предусмотреть возможность выполнения завершающих операций в инструкции `try/finally`, а в специализированных программах – перехватить это событие.¹ По этой причине

¹ Похожая функция `os._exit` также завершает работу программы, но делает это непосредственно – она пропускает этап выполнения завершающих действий и не может быть перехвачена с помощью инструкций `try/except` или `try/finally`. Обычно эта функция используется в дочерних процессах, описание которых выходит далеко за рамки этой книги. За дополнительной информацией обращайтесь к справочному руководству по библиотеке языка Python и к другим специализированным книгам.

инструкция `try` с пустым предложением `except` может непреднамеренно перехватить такое важное исключение, как показано в следующем файле (*exiter.py*):

```
import sys

def bye():
    sys.exit(40) # Серьезная ошибка: завершить работу программы 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.

Вероятно, хуже всего то, что пустое предложение `except` может перехватить настоящие ошибки в программном коде, которым желательно было бы позволить пройти дальше. Фактически пустые предложения `except` могут отключать механизм интерпретатора, предназначенный для вывода сообщений об ошибках, скрывая возможные ошибки в программном коде. Например, рассмотрим такой 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 единственный тип ошибки — это ошибка отсутствующего ключа. Но поскольку в имени словаря `myditctionary` была допущена опечатка (должно быть `mydictionary`), интерпретатор возбуждает исключение `NameError`, встретив ссылку на неопределенное имя, которое благополучно будет перехвачено и проигнорировано обработчиком. Обработчик неправильно запишет в переменную значение по умолчанию, замаскировав ошибку в программе. Если этот программный код будет находиться достаточно далеко от места, где используется выбранное значение, его отладка превратится в весьма захватывающую задачу!

Возьмите за правило специализировать свои обработчики, насколько это возможно — пустые предложения `except` удобны в использовании, но они потенциально опасны. Так, в последнем примере было бы лучше использовать предложение `except KeyError:`, чтобы более явно обозначить свои намерения и избежать возможности перехвата посторон-

них событий. В более простых сценариях подобные проблемы могут иметь не такое существенное значение, чтобы перевесить удобство использования, но в общем универсальные обработчики обычно доставляют массу неприятностей.

Не перехватывайте слишком мало: используйте категории

С другой стороны, было бы нежелательно делать обработчики слишком узкоспециализированными. Когда в инструкции `try` перечисляются конкретные исключения, перехватываться будут только те исключения, которые были перечислены. 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.

Например, следующий обработчик интерпретирует исключения `myerror1` и `myerror2` как нормальную ситуацию, а все остальные – как ошибку. Если в будущем будет добавлено исключение `myerror3`, оно будет обрабатываться как ошибка, если не добавить его в список исключений:

```
try:
    ...
except (myerror1, myerror2): # Работает неправильно при добавлении myerror3
    ...                     # Нет ошибки
else:
    ...                     # Рассматривается как ошибка
```

К счастью, при осторожном использовании исключений на основе классов, обсуждавшихся в главе 28, можно полностью избавиться от этой ловушки. Как мы уже видели, если перехватывать общий суперкласс, в будущем можно будет добавлять и возбуждать более конкретные подклассы исключений без необходимости изменять список исключений в предложении `except`; суперкласс становится легко расширяемой категорией исключений:

```
try:
    ...
except SuccessCategoryName: # Работает правильно при добавлении myerror3
    ...                     # Нет ошибки
else:
    ...                     # Рассматривается как ошибка
```

Если вы используете иерархии исключений на основе классов, порой придется пройти длинный путь, чтобы найти оптимальное решение. Мораль этой истории состоит в том, что вам следует с особым тщанием подходить к выбору степени детализации, чтобы обработчики исключений не были как слишком универсальными, так и слишком узкоспециализированными. Политика исключений должна быть составной частью общего дизайна, особенно в крупных программах,

Типичные проблемы при работе с исключениями

Вам едва ли часто придется сталкиваться с проблемами при работе с исключениями и, тем не менее, здесь я дам две общих рекомендации по их использованию (одна из которых касается концепций, с которыми мы уже встречались).

Соответствие строковым исключениям определяется идентичностью объектов, а не равенством значений

Теперь, когда исключения, как предполагается, идентифицируются классами, а не строками, эта проблема уходит в разряд устаревших проблем. Однако вам все еще могут встретиться исключения на основе строк в существующем программном коде, поэтому вам стоит знать об этой проблеме. Когда возбуждается исключение (вашей программой или самим интерпретатором), интерпретатор пытается отыскать ближайшую инструкцию `try` с соответствующим предложением `except` — то есть предложение `except`, в котором указан тот же самый строковый объект (в случае исключений на основе строк) или тот же самый класс, или его суперкласс (для исключений на основе классов). В случае использования строковых исключений важно знать, что соответствие определяется с помощью операции идентичности, а не равенства.

Например, предположим, что мы определили два строковых объекта, которые будут использоваться в качестве исключений:

```
>>> ex1 = 'The Spanish Inquisition'
>>> ex2 = 'The Spanish Inquisition'

>>> ex1 == ex2, ex1 is ex2
(True, False)
```

Оператор `==` возвращает значение `True`, потому что объекты имеют одинаковые значения, но оператор `is` возвращает `False`, потому что это разные строковые объекты (предполагается, что строки имеют достаточную длину, чтобы устранить влияние внутреннего механизма кэширования строк, описанного в главе 6). Таким образом, предложение `except`, указывающее на тот же самый строковый объект, всегда будет считаться соответствующим:

```
>>> try:
...     raise ex1
... except ex1:
...     print 'got it'
...
got it
```

но предложение, где указан объект с тем же значением, но не идентичный строковому объекту исключения, не будет считаться соответствующим:

```
>>> try:
...     raise ex1
... except ex2:
...     print 'Got it'
...
Traceback (most recent call last):
  File "<pyshell#43>", line 2, in <module>
    raise ex1
The Spanish Inquisition
```

Здесь исключение не было перехвачено, потому что объекты не идентичны. В результате исключение поднимается на верхний уровень процесса и автоматически выводится содержимое стека и текст исключения. Однако, как я уже говорил раньше, в некоторых случаях может получаться так, что различные строковые объекты с одинаковыми значениями будут считаться идентичными из-за влияния механизма кэширования небольших строк в Python (как описывается в главе 6):

```
>>> try:
...     raise 'spam'
... except 'spam':
...     print 'got it'
...
got it
```

Этот пример работает правильно, потому что две строки отображаются на один и тот же объект в кэше. Напротив, в следующем примере используются строки достаточной длины, чтобы не попасть в кэш:

```
>>> try:
...     raise 'spam' * 8
... except 'spam' * 8:
...     print 'got it'
...
Traceback (most recent call last):
  File "<pyshell#58>", line 2, in <module>
    raise 'spam' * 8
spamspamspamspamspamspamspamspam
```

Если эти два примера кажутся вам непонятными, просто запомните: в случае применения строк в инструкциях `raise` и `try` необходимо использовать один и тот же объект, а не одно и то же значение.

Для исключений на основе классов (рекомендуемая ныне методика) все работает точно так же, но язык Python обобщает понятие соответствия исключению, добавляя отношение к суперклассу, поэтому такой проблемы в случае с классами не наблюдается — еще одна причина использовать исключения на основе классов!

Перехват ненужных исключений

Пожалуй, самая типичная проблема, имеющая отношение к исключениям, связана с общими рекомендациями по проектированию, рассматривавшимися в предыдущем разделе. Старайтесь избегать использования пустых предложений `except` (можно перехватить такие события, как завершение программы), а также слишком узкоспециализированных предложений `except` (используйте суперклассы категорий, чтобы избежать проблем сопровождения в будущем, когда в программе появятся новые исключения).

Заключение по основам языка

Поздравляю! Этим разделом заканчивается ваше изучение основ языка программирования Python. Если вы забрались так далеко, что читаете эти строки, можете смело считать себя Официальным Программистом на языке Python (и можете не стесняться упоминать о знании этого языка в своих резюме). Вы уже видели почти все, что можно увидеть в самом языке, и получили знания более глубокие, чем имели многие практикующие программисты на языке Python в начале своего пути. Вы изучили встроенные типы, инструкции и исключения, а также инструменты, которые используются для создания крупных элементов программ (функции, модули и классы). Кроме того, вы исследовали ряд важных проблем, связанных с проектированием, ООП, архитектуру программы и многое другое.

Набор инструментальных средств языка Python

Начиная с этого момента, ваша будущая карьера программиста на языке Python в значительной степени будет состоять из овладения *инструментальными средствами*, доступными для прикладного программирования на языке Python. Это может занять немало времени. Стандартная библиотека, например, содержит сотни модулей, а разработчиками сообщества предлагается еще больше. Чтобы познакомиться со всеми этими инструментами, может потребоваться лет десять, а то и больше, особенно если учесть, что постоянно появляются новые (можете мне поверить!).

Вообще говоря, Python обеспечивает следующую иерархию инструментальных 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 возможности Python своими собственными функциями, модулями и классами.

Компилируемые расширения

Хотя мы и не касались данной темы в этой книге, тем не менее, возможности Python можно расширять с помощью модулей, написанных на других языках программирования, таких как C или C++.

Благодаря такой многоуровневой организации инструментальных средств вы можете выбирать, насколько глубоко погружаться в эту иерархию при создании своих программ, – для простых сценариев достаточно будет встроенных средств, для крупных программ могут потребоваться дополнительные расширения на языке Python, а компилируемые расширения – для решения необычных задач. В этой книге мы охватили первые две категории, и этого вполне достаточно, чтобы начать писать на языке Python серьезные программы.

В табл. 29.1 приводятся некоторые из встроенных и других функциональных возможностей, доступных в языке Python, исследованием которых вы будете заниматься остаток вашей карьеры программиста на языке Python. До настоящего момента наши примеры были очень маленькими и самостоятельными. Главная их цель состояла в том, чтобы помочь вам освоить основы. Но теперь, когда вы узнали все о базовом языке, настало время учиться использовать встроенные интерфейсы Python, чтобы быть в состоянии выполнять настоящую работу. Вы обнаружите, что такой простой язык, как Python, делает решение наиболее распространенных задач намного более легким делом, чем можно было бы ожидать.

Таблица 29.1. Категории инструментальных средств в языке Python

Категория	Примеры
Типы объектов	Списки, словари, файлы, строки
Функции	len, range, apply, open
Исключения	IndexError, KeyError
Модули	os, Tkinter, pickle, re
Атрибуты	__dict__, __name__, __class__
Внешние инструменты	NumPy, SWIG, Jython, IronPython и другие

Инструменты разработки крупных проектов

Как только вы овладеете основами языка, вы обнаружите, что ваши программы становятся существенно больше, чем примеры, с которыми вы экспериментировали до сих пор. Для разработки крупных программ и в Python, и в общем доступе имеется целый набор инструментов разработки. Некоторые из них вы видели в действии, некоторые я только упомянул. Чтобы помочь вам на вашем нелегком пути, я приведу краткое описание некоторых наиболее часто используемых инструментов:

PyDoc и строки документирования

Функция `help` и HTML-интерфейсы модуля PyDoc были представлены в главе 14. Модуль PyDoc реализует систему документирования для модулей и объектов и интегрирован со строками документирования. Это стандартная часть системы Python, поэтому за дополнительными подробностями обращайтесь к справочному руководству по библиотеке. Кроме того, в главе 14 даются подсказки с указанием на источники документации и другие информационные ресурсы по языку Python.

PyChecker

Python – это динамический язык программирования, поэтому некоторые ошибки сложно обнаружить, пока программа не будет выполнена (например, синтаксические ошибки можно выявить при запуске или во время импортирования файла). Это не такой большой недостаток – как и для большинства языков программирования это лишь означает, что прежде чем распространять свой программный код, его необходимо тестировать. Кроме того, динамическая природа языка Python, автоматический вывод сообщений об ошибках и модель исключений в языке Python позволяют быстрее и проще отыскивать и исправлять ошибки, чем в других языках программирования (например, в отличие от языка C, интерпретатор Python не вызывает крах системы при появлении ошибок).

Система PyChecker обеспечивает возможность выявления широкого круга наиболее часто встречающихся ошибок еще до того, как сценарий будет запущен. Она играет роль, похожую на ту, какую играет программа «lint» в разработке на языке C. Некоторые коллективы разработчиков проверяют свой программный код на языке Python с помощью PyChecker еще до его тестирования или распространения, чтобы выявить все скрытые проблемы. В действительности даже стандартная библиотека языка Python регулярно проверяется с помощью PyChecker перед выпуском. PyChecker – это сторонний пакет, распространяемый с открытыми исходными текстами. Найти его можно по адресу <http://www.python.org> или на веб-сайте проекта Vaults of Parnassus.

PyUnit (он же unittest)

В пятой части книги мы видели, как в файлы модулей добавляется программный код самопроверки, который использует результат проверки `__name__ == '__main__'`. Дополнительно для нужд тестирования в состав Python входят два инструмента. Первый, PyUnit (в руководстве по библиотеке называется `unittest`), обеспечивает комплект классов, с помощью которых можно определить и настроить варианты тестов и указать ожидаемые результаты. Он напоминает платформу JUnit в языке Java. Это сложная основанная на классах, система, подробное описание которой вы найдете в справочном руководстве по библиотеке Python.

Doctest

Модуль `doctest`, входящий в состав стандартной библиотеки, реализует второй и более простой подход к регрессивному тестированию. Он основан на использовании строк документирования в языке Python. В первом приближении, чтобы воспользоваться модулем `doctest`, следует скопировать результаты тестирования в интерактивном сеансе в строки документирования в файле с исходным текстом. После этого модуль `doctest` извлечет эти строки документирования, вычленив из них описание тестов с ожидаемыми результатами и повторно выполнит тесты, чтобы сравнить полученные результаты с ожидаемыми. Функциональные возможности `doctest` могут использоваться разными способами, о чем подробнее рассказывает в справочном руководстве по библиотеке Python.

Интегрированные среды разработки

В главе 3 мы уже рассматривали интегрированные среды разработки для языка Python. Такие интегрированные среды, как IDLE, обеспечивают графический интерфейс для редактирования, запуска, отладки и просмотра программ на языке Python. Некоторые мощные интегрированные среды разработки, такие как Eclipse и Komodo, поддерживают решение дополнительных задач разработки, включая интеграцию с системами контроля версий, интерактивные построители графического интерфейса, создание файлов проектов и многих других. Список интегрированных сред разработки и строителей графических интерфейсов для языка Python вы найдете в главе 3, а также на сайтах <http://www.python.org> и Vaults of Parnassus.

Профилировщики

Поскольку язык Python является высокоуровневым и динамическим языком программирования, интуитивные представления о производительности, которые следуют из опыта работы с другими языками программирования, неприменимы к программному коду на языке Python. Чтобы выявить узкие места в программе, вам необходимо добавить логику, выполняющую замеры временных интервалов с помощью инструментов, определяемых в модулях `time` или `timeit`, или запустить свой программный код под управлением модуля `profile`. Мы уже видели модули измерения времени в действии, когда сравнивали скорость работы итерационных инструментов в главе 17.

Модуль `profile` – это модуль стандартной библиотеки, который реализует профилирование исходных текстов программ на языке Python. Он выполняет строку, которую вы ему передадите (например, импорт файла или вызов функции), и затем по умолчанию выводит в стандартный поток вывода отчет, в котором собрана информация о производительности – количество вызовов каждой функции, время, потраченное каждой функцией, и многое другое. Модуль `profiler` допускает дополнительную настройку, например, он может

сохранять полученную информацию в файле для последующего анализа с помощью модуля `pstats`.

Отладчики

Стандартная библиотека языка Python включает модуль отладчика исходных текстов с именем `pdb`. Этот модуль работает подобно отладчику командной строки в языке C (например, `dbx`, `gdb`): вы импортируете модуль, запускаете программный код вызовом функции из модуля `pdb` (например, `pdb.run("main()")`) и затем вводите команды отладчика в интерактивном режиме. Кроме того, модуль `pdb` включает полезную функцию для проведения послеаварийного анализа — `pdb.pm`, которая позволяет выполнять отладку после появления исключения. Многие интегрированные среды разработки, такие как IDLE, включают интерфейсы «указал и щелкнул», поэтому `pdb` в наши дни используется относительно редко — советы по использованию отладчика с графическим интерфейсом в IDLE вы найдете в главе 3.¹

Варианты распространения

В главе 2 были представлены инструменты, используемые для упаковки программ на языке Python. Такие инструменты, как `py2exe`, `PyInstaller` и `freeze`, могут упаковывать байт-код и виртуальную машину с интерпретатором Python в «фиксированные двоичные файлы», которые способны выполняться как самостоятельные программы, не требуют установки Python и полностью скрывают программный код. Кроме того, в главе 2 и в пятой части книги мы узнали, что программы на языке Python могут распространяться в виде исходных текстов (`.py`) или в виде байт-кода (`.pyc`), а также о существовании программных ловушек, обеспечивающих возможность реализации специализированных приемов работы с пакетами, таких как автоматическое извлечение файлов из архивов в формате `.zip` и шифрование байт-кода. Мы также познакомились с модулем `distutils`, входящим в состав стандартной библиотеки, который обеспечивает упаковку модулей и пакетов на языке Python и расширений, написанных на языке C, — за дополнительной информацией обращайтесь к справочным руководствам по языку Python.

¹ Честно говоря, отладчик интегрированной среды IDLE тоже используется не слишком часто. Большинство практикующих программистов предпочитают отлаживать свой программный код вставкой инструкций `print` в критических точках. Поскольку промежуток времени от изменения до выполнения в языке Python очень короткий, добавление инструкций `print` выполняется быстрее, чем ввод команды отладчика `pdb` или запуск сеанса отладки в среде разработки с графическим интерфейсом. Еще один прием отладки, который вообще не требует никаких действий, основан на использовании сообщений, которые выводит интерпретатор в случае ошибки. Обычно эти сообщения дают достаточно информации, чтобы проанализировать и исправить ошибку.

Недавно появившаяся в языке Python система подготовки дистрибутивов «eggs» представляет собой другую альтернативу, которая позволяет учитывать зависимости, – дополнительную информацию о ней ищите в Сети.

Способы оптимизации

В главе 2 была описана система Psycodo, позволяющая оптимизировать программы по скорости выполнения. Она предоставляет динамический компилятор, выполняющий трансляцию байт-кода в двоичный машинный код. Кроме того имеется система Shedskin, которая представляет собой транслятор исходных текстов с языка Python на язык C++. Иногда вам могут встретиться файлы *.pyo* с оптимизированным байт-кодом, которые создаются при запуске интерпретатора Python с ключом командной строки `-O` (обсуждается в главе 18), но так как этот способ обеспечивает весьма скромное увеличение производительности, он обычно не используется. Наконец, для повышения производительности можно отдельные части своих программ перенести на компилирующийся язык программирования, такой как C, – подробнее о расширениях на языке C рассказывается в книге «Programming Python» и в стандартных руководствах по языку Python. Вообще говоря, скорость работы интерпретатора Python постоянно увеличивается, поэтому старайтесь использовать самую свежую его версию, когда это возможно (например, версия Python 2.3 обеспечила прирост производительности на 15–20 процентов по сравнению с версией 2.2).

Другие советы по разработке крупных проектов

Кроме всего прочего, в этой книге мы познакомились с различными особенностями языка, удобство которых особенно ярко проявляется при работе с крупными проектами. Среди них: пакеты модулей (глава 20); исключения на основе классов (глава 28); псевдочастные атрибуты класса (глава 25); строки документирования (глава 14); файлы с настройками пути поиска модулей (глава 18); сокрытие имен, импортируемых инструкцией `from *` с помощью списков `__all__` и имен в формате `_X` (глава 21); добавление программного кода самопроверки с использованием приема `__name__ == '__main__'` (глава 21); использование общих правил проектирования при создании функций и модулей (главы 16, 17 и 21) и т. д.

Узнать о других общедоступных разнообразных инструментах разработки можно на страницах веб-сайта Vaults of Parnassus.

В заключение

Эта глава завершила часть книги, описывающую исключения (и всю книгу в целом) кратким обзором типичных случаев использования исключений и инструментальных средств разработки. Поскольку это конец книги, глава заканчивается единственным контрольным вопросом.

Как обычно, обязательно выполните упражнения к этой части, чтобы закрепить знания, полученные в последних нескольких главах. В следующих далее приложениях даются подсказки по установке Python и ответы к упражнениям.

Чтобы получить представление о том, что читать после этой книги, просмотрите список литературы, рекомендуемой для дальнейшего чтения в предисловии. Вы достигли того момента, когда язык Python действительно становится увлекательным, но на этом данная книга заканчивается. Теперь вы неплохо подготовлены к изучению других книг и ресурсов, которые помогут вам решать задачи прикладного программирования, такие как создание программ с графическим интерфейсом, разработка веб-сайтов, интерфейсов к базам данных и многие другие. Удачи вам в вашем путешествии и, конечно, «всегда смотрите на светлую сторону жизни»!

Закрепление пройденного

Контрольные вопросы

1. (Этот вопрос уже задавался в контрольных вопросах к главе 1 – видите, я же говорил, что это будет просто.) Почему слово `spam` так часто появляется в примерах в этой книге?

Ответы

1. Язык Python получил свое название в честь английской комик-группы Монти 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 пытаются заказать блюдо в кафетерии, 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 выполнить несколько упражнений на применение исключений, чтобы попрактиковаться в основах. Исключения действительно являются очень простым инструментом – если вы пользуетесь ими, значит вы владеете ими в полной мере.

Решения вы найдете в приложении В, в разделе «Часть VII, Исключения и инструменты».

1. `try/except`. Напишите функцию с именем `oops`, которая при вызове явно возбуждает исключение `IndexError`. Затем напишите другую функцию, которая вызывает функцию `oops` внутри инструкции `try/`

`except`, которая перехватывает ошибку. Что произойдет, если изменить функцию `oops` так, чтобы вместо `IndexError` она возбуждала исключение `KeyError`? Где располагаются имена `KeyError` и `IndexError`? (Подсказка: вспомните, что все простые несоставные имена находятся в одной из четырех областей видимости согласно правилу LEGB.)

2. *Объекты исключений и списки.* Измените функцию `oops`, которую вы только что написали так, чтобы она возбуждала ваше собственное исключение с именем `MyError` и передавала бы дополнительные данные вместе с исключением. Вы можете определить свое исключение на основе строки или класса. Затем расширьте инструкцию `try` в функции, которая вызывает функцию `oops`, так чтобы кроме исключения `IndexError` она перехватывала бы еще и это исключение, извлекала бы дополнительные данные и выводила бы их на экран. Наконец, если вы использовали исключение на основе строки, вернитесь назад и измените исключение, так чтобы оно было основано на классе, — что теперь будет передаваться обработчику в виде дополнительных данных?
3. *Обработка ошибок.* Напишите функцию `safe(func, *args)`, которая запускает указанную функцию с помощью функции `apply` (или с использованием более нового синтаксиса вызова `*name`), перехватывает любые исключения, возникающие в ходе выполнения этой функции и выводит информацию об исключении с использованием атрибутов `exc_type` и `exc_value` модуля `sys` (или с помощью более современной функции `sys.exc_info`). Затем с помощью своей функции `safe` запустите функцию `oops` из упражнения 1 или 2. Поместите функцию `safe` в модуль с именем `tools.py` и передайте ей функцию `oops` в интерактивном режиме. Какие сообщения об ошибках вы получили? Наконец, расширьте свою функцию `safe` так, чтобы при возникновении исключения она выводила содержимое стека вызовов с помощью встроенной функции `print_exc`, расположенной в стандартном модуле `traceback` (за дополнительной информацией обращайтесь к руководству по библиотеке языка Python).
4. *Примеры для самостоятельного изучения.* В конце Приложения В я добавил несколько примеров сценариев, разработанных в ходе выполнения упражнений моими студентами, чтобы вы могли самостоятельно изучить и опробовать их, попутно изучая набор стандартных руководств по языку Python. Эти примеры не содержат описаний, и в них используются инструменты из стандартной библиотеки языка Python, которые вам потребуется отыскать. Для многих читателей эти примеры помогут увидеть, как концепции, которые мы обсуждали в этой книге, объединяются 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 «Programming Python», и в Сети.

VIII

Приложения



Установка и настройка

В этом приложении описываются подробности установки и настройки в помощь 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 установка Python. Если он еще не установлен у вас на компьютере, вам необходимо получить и установить последнюю версию Python на свой компьютер и, возможно, настроить его. Для каждого компьютера эту процедуру необходимо выполнить всего один раз, а если вы запускаете фиксированные двоичные файлы (описываемые в главе 2), вам вообще не потребуется устанавливать интерпретатор.

Возможно, Python уже установлен?

Прежде чем приступать к установке, проверьте, возможно, на вашем компьютере уже установлена свежая версия Python. Если вы работаете в операционной системе Linux, Mac OS X или UNIX, вполне вероятно, что Python уже установлен на вашем компьютере. Введите в командной строке (иногда называется окном терминала) команду `python` и посмотрите, что произойдет. Можно попробовать поискать подкаталог *python* в обычных местах – в каталогах `/usr/bin`, `/usr/local/bin` и других.

В операционной системе Windows поищите пункт Python в меню кнопки Пуск (Start), расположенной в левом нижнем углу экрана. Если интерпретатор установлен, убедитесь, что он имеет версию 2.5 или выше. Несмотря на то, что для работы с этой книгой подойдет любая достаточно свежая версия, тем не менее, для запуска некоторых примеров вам потребуется версия не ниже 2.5.

Где получить Python

Если вы не нашли Python у себя на компьютере, вам придется установить его. Могу вас обрадовать, Python является программным обеспечением, распространяемым с открытыми исходным текстами, и его можно свободно загрузить из Сети, к тому же на большинстве платформ Python устанавливается достаточно просто.

Самую свежую и лучшую версию Python всегда можно получить на официальном 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 (достаточно распаковать и запустить) файлы дистрибутивов для Windows (чтобы запустить установку, достаточно щелкнуть мышью на ярлыке файла), пакеты в формате RPM для Linux (установка производится с помощью утилиты *rpm*), дистрибутивы с полными исходными текстами (которые требуется скомпилировать, чтобы установить интерпретатор) и другие. Ссылки на другие веб-страницы, с которых можно загрузить версии для некоторых платформ, таких как PalmOS, сотовые телефоны Nokia и Windows Mobile можно найти либо на сайте проекта Python.org, либо воспользовавшись поисковой службой Google.

Интерпретатор Python можно также обнаружить на компакт-дисках с дистрибутивами Linux, в составе некоторых программных продуктов и как приложение к некоторым книгам по языку Python. Обычно они немного отстают от текущей версии, но это отставание, как правило, не очень велико.

Кроме того, компания ActiveState распространяет Python в составе своего пакета *ActivePython*. В состав пакета входят CPython с расширениями для разработки программ в операционной системе Windows, интегрированная среда разработки PythonWin (описывается в главе 3) и другие часто используемые расширения. Дополнительную информацию о пакете ActivePython вы найдете на веб-сайте компании ActiveState.

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, однако установка 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 рекомендации по установке в некоторых основных платформах:

Windows

Для операционной системы Windows 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, поддержку библиотеки построения графических интерфейсов Tkinter и базы данных shelve, а также среду разработки IDLE с графическим интерфейсом. Обычно Python 2.5 устанавливается в каталог *C:\Python25*, 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-браузере и удаление. Большинство этих действий связаны с концепциями, которые детально рассматривались 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). Существует возможность собрать Python из исходных текстов в Windows, но обычно это не делается.

Одно замечание для пользователей Windows Vista: особенности системы безопасности текущей версии Windows Vista изменили некоторые из правил использования инсталляционных файлов MSI. Обращайтесь за помощью к тексту врезки «Установка Python 2.5 из установочного файла формата MSI в Windows Vista» в этом приложении, если инсталляционный файл Python не запускается или если установка не выполняется в правильный каталог на вашей машине.

Linux

Для операционной системы Linux интерпретатор Python доступен в виде одного или нескольких файлов RPM, которые распаковываются обычным способом (за подробностями обращайтесь к странице справочного руководства по RPM). В зависимости от того, какие пакеты RPM вы загрузили, в одном может находиться сам интерпретатор Python, а в других – дополнительная поддержка Tkinter и среда разработки IDLE. Так как Linux является UNIX-подобной операционной системой, к нему применимы рекомендации, которые даются в следующем параграфе.

UNIX

В операционных системах UNIX Python обычно компилируется из дистрибутива с исходными текстами на языке C. Обычно для этого требуется распаковать файл и запустить команды `config` и `make` – Python настроит процедуру сборки автоматически, в соответствии с системой, где выполняется сборка. Однако обязательно ознакомьтесь с содержимым файла *README*, где приводятся дополнитель-

ные замечания по процессу сборки. Поскольку Python является программным продуктом, распространяемым с открытыми исходными текстами, его исходный программный код может свободно использоваться и распространяться.

Процедура установки в других платформах может существенно отличаться. Так, установка «Pipru», версии Python для PalmOS, например, требует выполнения операции синхронизации вашего PDA, а Python для PDA Sharp Zaurus, работающего под управлением Linux, поставляется в виде одного или более файлов *.ipk*, которые достаточно просто запустить, чтобы выполнить установку. Дополнительные процедуры установки для дистрибутивов в виде исполняемых файлов и пакетов с исходными текстами прекрасно документированы, поэтому мы пропустим дальнейшие подробности.

Настройка Python

После установки Python вам может потребоваться выполнить некоторые настройки, 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 строки. В этом разделе мы коротко рассмотрим переменные окружения Python. Параметры командной строки, которые указываются при запуске программ на языке Python из командной строки системы, используются достаточно редко и часто играют узкоспециализированные роли – за дополнительной информацией по этому вопросу обращайтесь к другим источникам документации.

Установка Python 2.5 из установочного файла формата MSI в Windows Vista

В середине 2007 года, когда я писал эти строки, дистрибутив Python для Windows распространялся в виде инсталляционного файла *.msi*. Файлы этого формата прекрасно работают в Windows XP (достаточно просто выполнить двойной щелчок мышью на этом файле, чтобы запустить его), однако в текущей версии Windows 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 выбранного по умолчанию каталога *C:\Python25*. Интерпретатор вполне работоспособен и в корневом каталоге, но это неправильное место для установки.

Эта проблема обусловлена особенностями поведения системы безопасности в Windows Vista. Дело в том, что в действительности файлы MSI не являются настоящими исполняемыми файлами, поэтому они неправильно наследуют права администратора, даже если запускаются администратором. Файлы MSI запускаются программой-инсталлятором MSI, ассоциация с которой определена в реестре Windows.

Чтобы установить Python 2.5.1 на свой карманный компьютер OQO, работающий под управлением Windows Vista, мне пришлось воспользоваться командной строкой и выполнить установку с правами администратора. Вот как это делается: откройте меню кнопки Пуск (Start), выберите пункт Все программы (All Programs), выберите пункт Стандартные (Accessories), щелкните правой кнопкой на пункте Командная строка (Command Prompt), в контекстном меню выберите пункт Запуск с правами администратора (Run as administrator) и выберите кнопку Продолжить (Continue) в диалоге управления доступом. Теперь в окне Командная строка (Command Prompt) запустите команду `cd`, чтобы перейти в каталог, где находится файл MSI с дистрибутивом Python (например, `cd C:\user\downloads`), и затем запустите программу-инсталлятор MSI, выполнив команду вида: `msiexec /i python-2.5.1.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 отключение системы безопасности Vista, если вы решитесь на это). Возможно также, что сам установочный файл Python будет распространяться в другом формате, ликвидирующем эту проблему, например, в виде настоящего исполняемого файла. Прежде чем использовать какие-либо обходные пути, попробуйте сначала просто запустить установку щелчком мыши, возможно, она уже работает должным образом.

Переменные окружения Python

Переменные окружения, иногда известные как переменные командной оболочки или переменные DOS, определяются за пределами интерпретатора Python и потому могут использоваться для настройки его поведения каждый раз, когда он запускается на данном компьютере. Интерпретатор Python пользуется множеством переменных окружения, но лишь немногие из них используются достаточно часто; они приведены ниже. В табл. А.1. приводятся основные переменные окружения, используемые для настройки интерпретатора Python.

Таблица А.1. Переменные окружения, имеющие важное значение

Переменная	Назначение
PATH (или path)	Путь поиска файлов, используемый системой (используется при поиске исполняемого файла <i>python</i>)
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 просты в обращении и, тем не менее, приведу несколько рекомендаций:

- Переменная PATH определяет список каталогов, где операционная система будет пытаться отыскать исполняемые файлы программ. Обычно этот список должен включать каталог, где находится интерпретатор Python (файл *python* в операционной системе UNIX или файл *python.exe* в Windows).

Вам вообще не придется настраивать эту переменную, если вы работаете в каталоге, где находится интерпретатор Python, например, настройки в переменной PATH не имеют значения, если перед запуском какого-либо программного кода выполнить команду `cd C:\Python25` (чтобы перейти в каталог, где находится интерпретатор Python) или вместо команды `python` всегда выполнять команду `C:\Python25\python` (в команде присутствует полный путь к исполняемому файлу). Кроме того, переменная окружения PATH в основном используется для запуска команд из командной строки – эта

Установка поддержки Tkinter в Linux

Среда разработки IDLE, описанная в главе 2, представляет собой программу на языке Python, которая использует библиотеку Tkinter для создания графического интерфейса. Библиотека Tkinter – это набор инструментальных средств для построения графического интерфейса; она является стандартным компонентом дистрибутива Python для Windows и некоторых других платформ. Однако в некоторых дистрибутивах Linux эта библиотека не входит в стандартный комплект устанавливаемых компонентов. Чтобы добавить в интерпретатор Python поддержку графического интерфейса в операционной системе Linux, попробуйте запустить команду `yum tkinter`, которая автоматически установит все необходимые библиотеки. Эта команда должна работать в дистрибутивах Linux (и в некоторых других системах), где имеется программа установки пакетов *yum*.

переменная не имеет значения при запуске программ щелчком мыши на ярлыке или из интегрированной среды разработки.

- Переменная окружения `PYTHONPATH` играет похожую роль: интерпретатор Python использует переменную `PYTHONPATH` во время поиска файлов модулей, когда они импортируются программами. (Дополнительную информацию о переменной `PYTHONPATH` вы найдете в главе 18.) Эта переменная содержит список каталогов в формате, зависящем от типа используемой платформы, – в UNIX каталоги в списке отделяются двоеточием, а в Windows – точкой с запятой. Обычно этот список должен включать только каталоги с вашими исходными текстами.

Вам не потребуется настраивать эту переменную, если не импортировать модули, находящиеся в других каталогах, потому что интерпретатор всегда автоматически пытается отыскать модули в домашнем каталоге программы. Настраивать эту переменную придется, только если какой-либо модуль должен импортировать другой модуль, расположенный в другом каталоге. Как упоминалось в главе 18, недавно появилась альтернатива переменной `PYTHONPATH` – файлы `.pth`.

- Если в переменной `PYTHONSTARTUP` указано полное имя файла с программным кодом на языке Python, интерпретатор будет запускать этот файл автоматически всякий раз, когда запускается интерактивный сеанс работы с интерпретатором, 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. это позволяет сэкономить время на импортировании вручную.
- Если вы предполагаете использовать набор инструментальных средств построения графического интерфейса Tkinter, вам может понадобиться записать в две переменные из табл. А.1 имена каталогов библиотек Tcl и Tk (похоже на `PYTHONPATH`). Однако в Windows (где поддержка Tkinter устанавливается вместе с интерпретатором Python) это не требуется, и обычно не требуется, если Tcl и Tk установлены в стандартные каталоги.

Обратите внимание: эти настройки окружения (так же, как и файлы `.pth`) являются внешними по отношению к интерпретатору Python, поэтому совершенно неважно, когда будет выполнена их настройка. Они могут быть настроены как до, так и после установки Python, главное, что это должно быть сделано перед *запуском* интерпретатора.

Как установить параметры конфигурации

Способ установки переменных окружения, имеющих отношение к Python, и устанавливаемые значения зависят от типа компьютера, с которым вы работаете. 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 весьма полезных модулей в каталогах *utilities* и *package1* где-то в компьютере, и вам необходимо иметь возможность импортировать их из файлов модулей, расположенных 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 следует одним из возможных способов настроить путь поиска модулей, чтобы включить в него каталог, содержащий файл *spam.py*. Ниже приводится несколько 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 командной оболочки *csh* для установки пути поиска модулей можно добавить 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 модулей должен выполняться в двух каталогах. Однако, если вы используете командную оболочку *ksh*, настройки можно выполнить в файле *.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 похожий) синтаксис.

Переменные DOS (Windows)

Если вы используете MS-DOS или старую версию Windows, вам может потребоваться 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\utilities;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* сохраняются постоянно и являются глобальными для всех программ.

Другие способы в Windows

В наиболее свежих версиях Windows имеется возможность устанавливать значение переменной окружения *PYTHONPATH* и других переменных с помощью графического интерфейса и тем самым избежать редактирования файлов и перезагрузки компьютера. В Windows 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 – это обычно пользовательская переменная). Вам не потребуется перезагружать компьютер, но необходимо будет перезапустить интерпретатор Python, если к моменту внесения изменений он уже был запущен (он воспринимает настройки пути только во время запуска).

Если вы опытный пользователь Windows, вы можете также настроить путь с помощью редактора реестра Windows. Выберите пункт меню «Старт→Выполнить...» («Start→Run...») и введите команду regedit. Если этот инструмент редактирования установлен у вас на компьютере, вы сможете с его помощью отыскать записи, имеющие отношение к Python, и выполнить необходимые изменения. Это достаточно сложная процедура, при выполнении которой легко ошибиться, поэтому если вы не знакомы с реестром, я рекомендую использовать другие возможности.

Файлы путей

Наконец, если для настройки пути поиска модулей вы решили использовать файл *.pth*, а не переменную окружения PYTHONPATH, в операционной системе Windows можно создать текстовый файл со следующим содержанием (файл *C:\Python25\mypath.pth*):

```
c:\pycode\utilities
d:\pycode\package1
```

Его содержимое будет отличаться для разных платформ, а каталог его размещения может отличаться как в зависимости от платформы, так и в зависимости от версии Python. Интерпретатор отыскивает эти файлы автоматически во время запуска.

Имена каталогов в файлах пути могут быть абсолютными или относительными по отношению к каталогу, где находится файл пути. Допускается использовать несколько файлов *.pth* (все каталоги, перечисленные в них, будут добавлены в путь поиска), а сами файлы *.pth* могут размещаться в любых каталогах, которые проверяются автоматически, в зависимости от используемой платформы и версии Python. Например, Python 2.5 пытается отыскать такие файлы в каталогах *C:\Python25* и *C:\Python25\Lib\site-packages* в операционной системе Windows и в каталогах */usr/local/lib/python2.5/site-packages* and */usr/local/lib/site-python* в Unix и Linux.

Поскольку эти настройки часто бывают необязательными и эта книга не описывает командные оболочки операционных систем, я оставляю освещение подробностей другим источникам информации. За более подробными сведениями обращайтесь к страницам справочного руководства своей командной оболочки или к другой документации. Если вы испытываете затруднения в определении того, какие настройки вам следует выполнить, обратитесь за помощью к своему системному администратору или другому опытному товарищу.

В

Решения упражнений

Часть I. Введение

Упражнения находятся в главе 3, в разделе «Упражнения к первой части».

1. *Взаимодействие.* Предположим, что настройка Python выполнена правильно, тогда сеанс взаимодействия с интерактивной оболочкой должен выглядеть примерно так, как показано ниже (вы можете запустить интерактивную оболочку любым способом, по своему усмотрению – в IDLE, из системной командной строки и т. д.):

```
% 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 взаимодействия с командной оболочкой операционной системы должны выглядеть, 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→Run Module (Запустить→Запустить модуль) и т. д.

3. *Модули.* Следующий пример сеанса взаимодействия иллюстрирует запуск модуля при его импортировании:

```
% python
>>> import module1
```

```
Hello module world!  
>>>
```

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 каталоге 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 интерпретатор имел доступ к файлу с исходным программным кодом, он автоматически создаст файл *.pyc*, содержащий байт-код скомпилированной версии модуля. Подробнее о модулях рассказывается в пятой части книги.

4. *Сценарии*. Предположим, что ваша платформа поддерживает интерпретацию комбинации символов `#!`, решение этого упражнения должно выглядеть примерно так, как показано ниже (на вашем компьютере в строке `#!` может потребоваться 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 исключения – по умолчанию при возникновении исключений интерпретатор завершает работу программы и выводит сообщение об ошибке вместе содержимым стека вызовов на экран. Информация из стека показывает, в каком месте программы произошло исключение. В седьмой части книги вы узнаете, как перехватывать исключения с помощью инструкции `try` и как обрабатывать их. Вы также увидите, что в состав Python входит полноценный отладчик исходного программного кода, обладающий возможностями анализа ошибок после возникновения исключения. А пока обратите внимание, что в случае появления ошибок Python выводит вполне информативные сообщения (вместо того чтобы просто аварийно завершать программу вообще без каких-либо сообщений):

```
% python  
>>> 1 / 0  
Traceback (innermost last):  
  File "<stdin>", line 1, in ?  
ZeroDivisionError: integer division or modulo  
(ZeroDivisionError: целочисленное деление или деление по модулю)  
>>>
```

```
>>> x
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: x
```

6. Прерывание программы. Если ввести следующие инструкции:

```
L = [1, 2]
L.append(L)
```

тем самым вы создадите циклическую структуру данных. В версиях ниже, чем Python 1.5.1, интерпретатор еще не обладает возможностью определять случаи заикливания в объектах, и он будет выводить бесконечный поток [1, 2, [1, 2, [1, 2, [1, 2, и т. д., пока вы не нажмете комбинацию клавиш, выполняющих прерывание программы (которая, с технической точки зрения, возбуждает исключение прерывания с клавиатуры и приводит к выводу сообщения по умолчанию). Начиная с версии Python 1.5.1, интерпретатор уже умеет определять случаи заикливания и вместо бесконечной последовательности выведет [[...]].

Причина заикливания пока для вас неочевидна; чтобы ее понять, необходимы знания, которые вы получите во второй части книги. Однако в двух словах замечу, что операции присваивания в языке Python всегда создают ссылки на объекты (которые можно представить себе, как неявное следование по указателям). После выполнения первой инструкции из тех, что представлены выше, имя `L` превратится в именованную ссылку на объект списка, состоящий из двух элементов. Списки в языке Python в действительности являются массивами ссылок на объекты, обладающими методом `append`, который изменяет сам массив, добавляя ссылку на другой объект. В данном случае вызов метода `append` добавляет в конец списка `L` ссылку на сам список `L`, что приводит к заикливанию, как показано на рис. В.1. В некоторых программах, где выполняется обход произвольных объектов, вам, возможно, самим придется опреде-

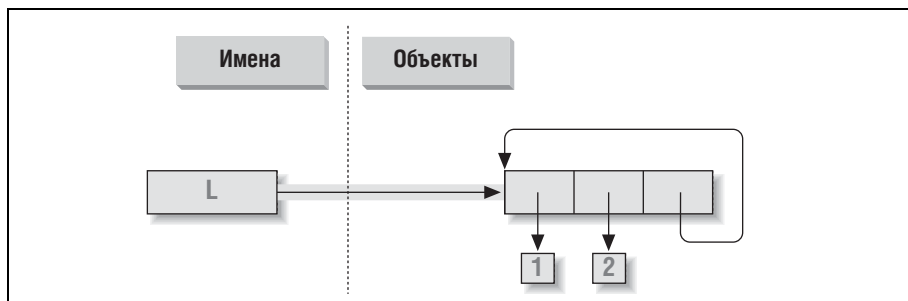


Рис. В.1. Циклический объект, созданный за счет добавления списка к самому себе. По умолчанию интерпретатор добавляет ссылку на оригинальный список, а не его копию

лять такие заикливания и внимательно изучать действия программы, чтобы избежать заикливания.

Хотите верить, хотите нет, но циклические структуры данных иногда могут быть полезны (но не тогда, когда выводятся на экран!).

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 некоторыми комментариями к ним. Обратите внимание, что здесь иногда используется символ `;`, чтобы поместить несколько инструкций в одну строку. Символ `;` — это разделитель инструкций:

```
# Числа
>>> 2 ** 16                # 2 в степени 16
65536
>>> 2 / 5, 2 / 5.0        # / усекает целые числа, а с плавающей точкой - нет
(0, 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'

# 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
[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 новую запись
>>> 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}
>>> D.keys(), D.values(), D.has_key((1,2,3)) # Методы
(['w', 'z', 'y', (1, 2, 3), 'x'], [0, 3, 2, 4, 1], 1)

# 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 корректирует срезы, выходящие за пределы, так, чтобы они соответствовали границам последовательности (пределы устанавливаются равными нулю и длине последовательности, если это необходимо).

Извлечение последовательности в обратном порядке, когда нижняя граница 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 выражение `L[3:3]` – пустая позиция со смещением 3). Срезы в языке Python всегда извлекаются слева направо, даже при использовании отрицательных индексов (они сначала будут преобразованы в положительные индексы за счет добавления длины последовательности). Обратите внимание, что в Python 2.5 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]
>>> 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 смещению приведет к сохранению объекта пустого списка в этой позиции, но присваивание пустого списка срезу приведет к удалению этого среза. Операция 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
(TypeError: неверный тип аргумента встроенной операции)
```

4. *Кортежи.* Значения X и Y поменяются местами. Когда слева и справа от оператора присваивания (=) появляются кортежи, интерпретатор выполнит присваивание объектов справа объектам слева в соответствии с их позициями. Это объяснение, пожалуй, самое простое для понимания, хотя целевые объекты слева и не являются настоя-

щим кортежем несмотря на то, что они именно так и выглядят, – это просто набор независимых имен, которым выполняется присваивание. Элементы справа – это кортеж, который распаковывается в процессе выполнения операции присваивания (кортеж играет роль временного хранилища присваиваемых данных, необходимого для достижения 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, строки и т. д. Это действительно словарь, несмотря на то, что некоторые из его ключей выглядят как целочисленные смещения. Допускается 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 несуществующему ключу (`D['d']`) приводит к появлению ошибки. Присваивание по несуществующему ключу (`D['d']='spam'`) создает новый элемент словаря. С другой стороны, выход индекса за границы списка тоже вызывает появление ошибки; ошибкой считается и присваивание по индексу, выходящему за границы списка. Имена переменных напоминают ключи словаря – прежде чем обратиться к переменным, им необходимо присвоить значение. Имена переменных, если понадобится, 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
(IndexError: выход индекса за границы списка)
>>> L[2] = 3
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
(IndexError: присваивание по индексу за границами списка)
```

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 последовательностями.
- Метод `append` может применяться только к спискам, но не к строкам, а метод `keys` может применяться только к словарям. Метод `append` предполагает, что целевой объект является изменяемым, поскольку изменяется сам объект; строки относятся к неизменяемым типам.
- Операции извлечения среза и конкатенации всегда возвращают новый объект того же типа, что и объекты-операнды.

```
>>> "x" + 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
(TypeError: неверный тип аргумента встроенной операции)
>>>
>>> {} + {}
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: bad operand type(s) for +
(TypeError: неверный тип операнда(ов) для +)
>>>
>>> [].append(9)
>>> "".append('s')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: attribute-less object
(AttributeError: объект без атрибутов)
>>>
>>> {}.keys()
[]
>>> [].keys()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: keys
(AttributeError: ключи)
```

```
>>>
>>> [][:]
[]
>>> ""[:]
```

8. *Индексирование строк.* Так как строки представляют собой коллекции односимвольных строк, каждый раз, когда извлекается элемент строки, возвращается строка, из которой также допускается извлекать элементы по индексам. Выражение `s[0][0][0][0][0]` всего лишь извлекает первый символ снова и снова. Этот прием неприменим к спискам (списки могут хранить объекты произвольных 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 решения. Присваивание по индексу недопустимо, потому что строки являются неизменяемыми объектами:

```
>>> s = "spam"
>>> s = s[0] + '1' + s[2:]
>>> s
'slam'
>>> s = s[0] + '1' + s[2] + s[3]
>>> s
'slam'
```

10. *Вложенные структуры.* Ниже приводится пример такой структуры:

```
>>> me = {'name':('mark', 'e', 'lutz'), 'age':'?', 'job':'engineer'}
>>> me['job']
'engineer'
>>> me['name'][2]
'lutz'
```

11. *Файлы.* Ниже приводится пример одного из способов создания файла и чтения информации из него в языке Python (`ls` – это команда UNIX, в Windows 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
```

12. *Функция dir*. Ниже показано, что будет получено в случае списков. Для словарей будет получена та же информация, но имена методов будут другими. Обратите внимание, действие функции `dir` в Python 2.2 было расширено – она дополнительно выводит имена, начинающиеся и заканчивающиеся символами подчеркивания, которые реализуют операторы выражений и могут быть унаследованы подклассами, как описывается в шестой части книги. Атрибут `__methods__` был удален в версии Python 2.2 из-за его противоречивой реализации – используйте вместо него функцию `dir` для извлечения списка атрибутов:

```
>>> [].__methods__
['append', 'count', 'index', 'insert', 'remove', 'reverse', 'sort',...]
>>> dir([])
['append', 'count', 'index', 'insert', 'remove', 'reverse', 'sort',...]
```

Часть III. Инструкции и синтаксис

Упражнения находятся в главе 14, 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]

>>> map(ord, S)
[115, 112, 97, 109]
```

2. *Символы обратного следа.* Пример выводит символ подачи звукового сигнала (\a) 50 раз – здесь предполагается, что ваш компьютер в состоянии воспроизвести его, поэтому при запуске этого фрагмента не под управлением IDLE вы можете получить серию звуковых сигналов (или один длинный сигнал, если ваш компьютер обладает достаточно высоким быстродействием). Эй! Я предупреждал вас!
3. *Сортировка словарей.* Ниже приводится один из возможных вариантов решения этого упражнения (вернитесь к главе 8, если что-то показалось 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 возможность организовать итерации по ключам словаря без вызова метода `keys` (например, `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 = D.keys()
>>> 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):      # В наиболее свежих версиях Pythons
...     print key, '=>', D[key]
```

4. *Программирование альтернативной логики.* Ниже приводятся несколько вариантов решения этого упражнения. Ваши решения могут несколько отличаться – цель этого упражнения состоит в том, чтобы дать вам возможность поэкспериментировать с созданием альтернативных ветвей исполнения программы, поэтому приветствуются любые правильные решения:

```
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 = i+1
else:
    print X, 'not found'

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'

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'

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'

X = 5
L = map(lambda x: 2**x, range(7))
print L

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 17, 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
>>> 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 инструкцию `print`, потому что выполнение программного кода в модуле — это совсем не то же самое, что выполнение программного кода, который вводится в интерактивной оболочке, — интерпретатор не выводит автоматически 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* показаны два варианта реализации функции `adder`. Самое сложное здесь заключается в инициализации значения суммы пустым значением в зависимости от типа передаваемых аргументов. Первое решение выясняет, является ли первый аргумент целым числом, и извлекает пустой срез первого аргумента (предполагается, что он является последовательностью), если он не является целым числом. Во втором решении в качестве начального значения используется первый аргумент, а затем выполняется сканирование аргументов, начиная со второго, почти как в одном из вариантов функции `min`, показанной в главе 16.

Второе решение является более предпочтительным. Но в обоих вариантах предполагается, что все аргументы принадлежат к одному и тому же типу, а, кроме того, предполагается, что функция не будет 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 предусмотреть специальный программный код для обработки словарей, за что вам причитаются дополнительные баллы:

```
def adder1(*args):
    print 'adder1',
    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',
    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. **Передача аргументов по ключу.** Ниже приводится мое решение первой части этого упражнения (файл *mod.py*). Для обхода аргументов, передаваемых по ключу, в заголовке функции следует использовать форму списка аргументов `**args` и цикл в теле функции (например, `for x in args.keys():` и использовать `args[x]`) или использовать `args.values()`, чтобы реализовать сложение по позициям 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

```

5. (и 6.) Ниже приводится мое решение упражнений 5 и 6 (файл *dicts.py*). Однако это решение пригодно только в качестве упражнения, потому что в Python 1.5 у словарей появились методы `D.copy()`

и `D1.update(D2)`, которые реализуют операции копирования и сложение (слияние) словарей. (За дополнительной информацией обращайтесь к справочному руководству по библиотеке Python или к книге «Python Pocket Reference», выпущенной издательством O'Reilly.) Операция извлечения среза `X[:]` не применима к словарям, потому что они не являются последовательностями (подробности в главе 8). Кроме того, не забывайте, что в случае присваивания (`e = d`) просто создается вторая ссылка на один и тот же объект словаря – изменения 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. **Дополнительные примеры на сопоставление аргументов.** Ниже приводится пример диалога с интерактивной оболочкой, который вы должны получить в ходе выполнения этого упражнения, с некоторыми комментариями, 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   # Сопоставление выполняется по ключам
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 позиционные

% 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 аргументы объединяются
1 (2, 3)                    # в кортеж

>>> f3(1, x=2, y=3)          # Дополнительные ключи объединяются
1 {'x': 2, 'y': 3}          # 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

```

7. **Снова простые числа.** Ниже приводится пример нахождения простых чисел, оформленный в виде функции в модуле (файл *primes.py*), которую можно вызывать несколько раз. Я добавил проверку, чтобы отсечь отрицательные числа, 0 и 1. Кроме того, я заменил оператор / на //, чтобы это решение правильно работало в Python 3.0, где будут внесены изменения в действие оператора /, о чем рассказывалось в главе 5, и чтобы добавить поддержку чисел с плавающей точкой. В настоящее время в решении можно использовать как оператор //, так и оператор /, но в будущем оператор / будет завершаться ошибкой (раскомментируйте инструкцию `from` и замените оператор // на /, чтобы увидеть разницу между версиями Python 2.2 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 будущем / будет терпеть неудачу
        while x > 1:
            if y % x == 0:     # нет остатка?
                print y, 'has factor', x
                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 модулю 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 использованию — вместо вывода результата она могла бы возвращать значения, но для экспериментов вполне достаточно и такой реализации. Кроме того, она не строго соответствует математическому определению простых чисел, которыми могут быть только целые числа, и не обладает достаточной эффективностью. Эти улучшения я оставляю для самостоятельной реализации читателям, склонным к математике. Подсказка: цикл `for` через `range(y, 1, -1)` будет выполняться немного быстрее, чем цикл `while` (на самом деле, в Python 2.2 скорость увеличивается почти в два раза), но самым узким местом здесь является сам алгоритм. Для измерения производительности альтернативных реализаций воспользуйтесь встроенным модулем `time` и отрывком кода, подобным следующей универсальной функции `timer` (за дополнительной информацией обращайтесь к справочному руководству по библиотеке):

```

def timer(reps, func, *args):
    import time
    start = time.clock()
    for i in xrange(reps):
        apply(func, args)
    return time.clock() - start

```

8. *Генераторы списков.* Ниже приводится вариант программного кода, который 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]

>>> 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]
```

Часть V. Модули

Упражнения находятся в главе 21, в разделе «Упражнения к пятой части».

1. *Основы импортирования.* Решение этого упражнения выглядит гораздо проще, чем можно было бы подумать. Когда вы закончите, у вас должен получиться файл (*mymod.py*) и сеанс взаимодействия с интерактивной оболочкой, как показано ниже. Не забывайте, что интерпретатор Python может читать содержимое файла целиком в список строк, а получить длину каждой 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 целиком и потому не в состоянии работать с патологически большими файлами, которые не смогут уместиться в памяти компьютера. Чтобы обеспечить более высокую устойчивость, можно организовать построчное чтение содержимого файла с помощью итератора и накапливать длины строк в ходе итераций:

```
def countLines(name):
    tot = 0
    for line in open(name): tot += 1
    return tot

def countChars(name):
    tot = 0
```

```
for line in open(name): tot += len(line)
return tot
```

В операционной системе UNIX полученный результат можно проверить с помощью команды `wc`. В Windows можно щелкнуть на файле правой кнопкой мыши и посмотреть его свойства. Обратите внимание, что результат, возвращаемый сценарием, может несколько отличаться от того, что сообщает Windows, – с целью обеспечения переносимости интерпретатор Python преобразует пары символов `\r\n`, отмечающих конец каждой строки, в один символ `\n`, в результате чего теряется по одному байту на каждую строку. Чтобы результат сценария в точности соответствовал тому, что сообщает Windows, файл необходимо открыть в режиме двоичного доступа (`'rb'`) или прибавлять к общему результату число строк.

Чтобы реализовать часть упражнения «для честолюбивых» (передавать 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 работает точно так же, как функция `fseek` в языке C (которая, собственно, и вызывается внутренней реализацией метода): метод `seek` переустанавливает текущую позицию в файле в указанное смещение. После вызова метода `seek` последующие операции ввода/вывода будут выполняться относительно новой позиции. Чтобы переместиться в начало файла, не закрывая и не открывая его повторно, можно вызвать метод `file.seek(0)`; все вызовы метода `read` выполняются чтение из текущей позиции в файле, поэтому, чтобы начать повторное чтение, необходимо переместить текущую позицию в начало 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
```

3. `__main__`. Если вы не допустили ошибок в модуле, он сможет работать в любом из режимов (в режиме самостоятельной программы или в режиме импортируемого 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)
```

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* доступны (то есть были импортированы) на верхнем уровне модуля *myclient*, потому что инструкция *from* просто присваивает их именам в импортирующем модуле (выглядит так, как если бы определения функций *def* находились в модуле *myclient*). Например, в следующем файле приводится альтернативный вариант реализации:

```
import myclient
myclient.countLines(...)

from myclient import countChars
countChars(...)
```

Если бы в модуле *myclient* вместо инструкции *from* использовалась инструкция *import*, вам пришлось бы использовать полные имена функций из модуля *mymod*:

```
import myclient
myclient.mymod.countLines(...)

from myclient import mymod
mymod.countChars(...)
```

Вообще говоря, можно определить *модуль-коллектор*, который будет импортировать все имена из других модулей, чтобы сделать их дос-

тупными в виде единого модуля. Если воспользоваться следующим программным кодом, вы покончите 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 целого числа, а в интерактивной оболочке будет существовать только одно имя `somename`:

```
# Файл: mod1.py
somename = 42

# Файл: collector.py
from mod1 import *      # Выборка всех имен переменных
from mod2 import *      # из инструкций присваивания и превращение их
from mod3 import *      # в собственные имена этого модуля

>>> 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 настройке каталога и необходимого файла `__init__.py` в консоли Windows – вы должны учитывать особенности своей платформы (например, использовать команды `mv` и `vi` вместо `move` и `edit`). Эти действия можно выполнять в любом произвольном каталоге (просто так вышло, что я запускал команды, находясь 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 находиться 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 инициализационном файле выполняется только при первой операции импорта:

```
C:\python25> mkdir mypkg
C:\Python25> move mymod.py mypkg\mypkg\mypkg.py
C:\Python25> edit mypkg\__init__.py
...добавление инструкции print...
C:\Python25> python
>>> import mypkg.mymod
initializing mypkg
>>> mypkg.mymod.countLines('mypkg\mypkg.py')
13
>>> from mypkg.mymod import countChars
>>> countChars('mypkg\mypkg.py')
346
```

6. **Повторная загрузка.** В этом упражнении вам просто предлагает поэкспериментировать 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`.

Если говорить более подробно, все происходит следующим образом: когда модуль `recur2` импортируется первым, в результате рекурсивного импорта модуля `recur2` из модуля `recur1` модуль `recur2` извлекается целиком. К моменту, когда модуль `recur2` импортируется модулем `recur1`, его пространство имен еще не заполнено, но так как импортирование выполняется инструкцией `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 сбоев. Когда импорт в модуле `recur2` возобновляется, вторая инструкция `from` обнаруживает в модуле `recur1` имя `Y` (модуль был выполнен полностью), поэтому в такой ситуации не возникает ошибок. Запуск файла как отдельного 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 равносильно предварительному импортированию модуля `recur2` в интерактивной оболочке, поскольку `recur2` – это первый модуль, импортируемый в `recur1`.

Часть VI. Классы и ООП

Упражнения находятся в главе 26, в разделе «Упражнения к шестой части».

1. *Наследование.* Ниже приводится решение этого упражнения (файл `adder.py`) вместе с несколькими примерами действий в интерактивной оболочке. Метод перегрузки оператора `__add__` присутствует только в суперклассе и вызывает конкретные методы `add`, определяемые 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})
{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
(TypeError: ни __add__, ни __radd__ не определен для этих операндов)

```

Обратите внимание, что в последнем тесте возникла ошибка при попытке использовать экземпляр класса справа от оператора +. Если вы захотите исправить эту ошибку, реализуйте метод `__radd__`, как описывается в главе 24, в разделе «Перегрузка операторов».

Если вы начнете сохранять текущее значение в экземпляре, вы можете также переписать метод `add`, чтобы он принимал единственный 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 передаются в виде аргументов, эта версия определенно является более объектно-ориентированной. И как только вы доберетесь до этого момента, вы наверняка обнаружите, что вообще можно избавиться от методов `add` и просто определить методы `__add__` в двух подклассах.

2. *Перегрузка операторов.* В программном коде решения (файл *mylist.py*) используется несколько методов перегрузки операторов, о которых в книге рассказывается не слишком много, однако в них нет ничего сложного. Операция копирования начального значения в конструкторе имеет большое значение, потому что оно может быть представлено изменяемым объектом – едва ли есть разумная причина, чтобы изменять или обладать ссылкой на объект, который, вполне возможно, используется где-то за пределами класса. Метод `__getattr__` перехватывает попытки обращения к обернутому списку. Подсказки, которые помогут упростить реализацию, вы найдете в разделе «Расширение типов наследованием» в главе 26:

```
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 т. д.
        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,
```

```
% 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 значение, выполняя добавление в конец вместо извлечения среза, потому что в противном случае может получиться объект, не являющийся списком и потому не обладающий методами списка, такими как `append` (например, операция извлечения среза строки возвращает другую строку, но не список). Если бы начальное значение было объектом 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 использования 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 попрактиковаться (подробнее о множествах рассказывается в главе 5).

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 = MyListSub.calls + 1 # Счетчик, единый для класса
        self.adds = self.adds + 1 # Подсчет экземпляров
        return MyList.__add__(self, other)

    def stats(self):
        return self.calls, self.adds # All adds, my adds

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. Обратите внимание, что операторы будут извлекать атрибуты с помощью метода `__getattr__` — он должен возвращать значение, чтобы операторы могли работать:

```
>>> class Meta:
...     def __getattr__(self, name):
...         print 'get', name
...     def __setattr__(self, name, value):
...         print 'set', name, value
...
>>> x = Meta()
>>> x.append
get append
>>> x.spam = "pork"
set spam pork
>>>
>>> x + 2
get __coerce__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function
(TypeError: попытка вызова объекта, не являющегося функцией)
>>>
>>> x[1]
get __getitem__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function
(TypeError: попытка вызова объекта, не являющегося функцией)
>>> x[1:5]
get __len__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function
(TypeError: попытка вызова объекта, не являющегося функцией)
```

5. **Объекты множеств.** Ниже приводится сеанс взаимодействия, который должен у вас получиться. Комментарии описывают, какие методы вызываются.

```
% 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,                  # __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 изменить 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
    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:
            for other in others:
                if x not in other: break
            else:
                res.append(x)
        return Set(res)

    def union(*args):
        res = []
        for seq in args:
            for x in seq:
                if not x in res:
                    res.append(x)
        return Set(res)

```

Ваш сеанс взаимодействия с интерактивной оболочкой должен выглядеть примерно так, как показано ниже. Обратите внимание, что пересечение двух множеств можно находить как с помощью оператора `&`, так и с помощью метода `intersect`, но для случая трех и более множеств обязательно нужно вызывать метод `intersect` — оператор `&` является двухместным. Кроме того, обратите внимание, что можно было класс `MultiSet` назвать просто `Set`, чтобы сделать это изменение более прозрачным, если бы в файле `multiset` в качестве имени наследуемого 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 показано, как я изменил класс `Listner`, и результат повторного запуска теста. Чтобы дополнительно отобразить унаследованные атрибуты, необходимо реализовать метод, аналогичный текущему методу `attrnames`, и предусмотреть в нем рекурсивный обход ссылок `__bases__` в каждом классе. Поскольку, начиная с версии Python 2.2, функция `dir` включает в результат унаследованные атрибуты, можно было бы просто организовать цикл по ее результатам: например, `for x in dir(self)` и использовать `getattr(self, x)`. Однако такой прием не поможет, если вам потребуются вывести структуру дерева классов (как в примере *classtree.py* в главе 24):

```
class Listner:
    def __repr__(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):
        result = ""
        first = 1
        for super in self.__class__.__bases__: # Вверх на один уровень
```



```

        if not first:
            result = result + ", "
        first = 0
        result = result + super.__name__      # имя, не repr(super)
    return result

C:\python\examples> python testmixin.py
<Instance of Sub(Super, Lister), address 7841200:
  name data3=42
  name data2=eggs
  name data1=spam
>

```

- 7. Композиция.** Мое решение с комментариями, взятыми из описания, приводится 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
    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 способ реализации 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 сотрудников. Обратите внимание, что в классе *Animal* выполняется обращение к методу *self.speak*, это приводит к выполнению независимого поиска метода *speak* в дереве наследования, начиная с подкласса. Протестируйте работу этого дерева классов, следуя описанию упражнения. Попробуйте дополнить эту иерархию новыми классами и создать 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 упражнения (файл *parrot.py*). Обратите внимание на то, как работает метод *line* в суперклассе *Actor*: он дважды обращается к атрибутам аргумента *self*, дважды отсылая интерпретатор к экземпляру, и тем самым дважды иницирует 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):
    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 29, 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 программного кода: если изменить функцию `oops` так, чтобы вместо `IndexError` она возбуждала исключение `KeyError`, то обработчик в инструкции `try` не сможет перехватывать исключение (оно достигнет верхнего уровня и приведет к появлению сообщения об ошибке по умолчанию). Имена `KeyError` и `IndexError` определены во встроенной области видимости. Импортируйте модуль `__builtin__` 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 (для начала исключение определено в виде строки):

```

MyError = 'hello'

def oops():
    raise MyError, 'world'

def doomed():
    try:
        oops()
    except IndexError:
        print 'caught an index error!'
    except MyError, data:
        print 'caught error:', MyError, data

```

```

    else:
        print 'no error caught...'

if __name__ == '__main__':
    doomed()

% python oops.py
caught error: hello world

```

Чтобы идентифицировать исключение как класс, я просто изменил первую часть файла, как показано ниже, и сохранил его под именем *oop_oops.py*:

```

class MyError: pass

def oops():
    raise MyError()

...остальная часть осталась без изменений...

```

Как и в случае любого другого исключения на основе класса, в виде дополнительных данных передается сам экземпляр — теперь сообщение об ошибке содержит имя класса и указание на его экземпляр (<...>).

```

% python oop_oops.py
caught error: __main__.MyError <__main__.MyError instance at 0x00867550>

```

Не забывайте, что внешний вид сообщения можно улучшить, если определить метод `__repr__` или `__str__` в своем классе так, чтобы он возвращал нужную строку. Подробнее об этом рассказывается в главе 24.

3. **Обработка ошибок.** Ниже приводится мое собственное решение этого упражнения (файл *safe2.py*). Я добавил тесты непосредственно в файл, чтобы не проводить их в интерактивной оболочке, но результаты от этого не меняются.

```

import sys, traceback

def safe(entry, *args):
    try:
        apply(entry, args)      # Перехватывать любые исключения
    except:
        traceback.print_exc()
        print 'Got', sys.exc_type, sys.exc_value

import oops
safe(oops.oops)

% python safe2.py
Traceback (innermost last):
  File "safe2.py", line 5, in safe
    apply(entry, args)          # Перехватывать любые исключения
  File "oops.py", line 4, in oops
    raise MyError, 'world'

```

```
hello: world
Got hello world
```

Сегодня я реализовал бы эту функцию, как показано ниже, используя новый синтаксис `*args` и функцию `exc_info`:

```
def safe(entry, *args):
    try:
        entry(*args)                # Перехватывать любые исключения
    except:
        traceback.print_exc()
        print 'Got', sys.exc_info()[0], sys.exc_info()[1]
```

- 4. Далее приводятся несколько примеров для самостоятельного изучения на досуге; еще больше примеров программ на языке Python вы найдете в последующих книгах и в Сети:**

Поиск наибольшего файла в единственном каталоге

```
dirname = r'C:\Python25\Lib'
import os, glob

allsizes = []
allpy = glob.glob(os.path.join(dirname, '*.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 в дереве каталогов

```
import sys
if sys.platform[:3] == 'win':
    dirname = r'C:\Python25\Lib'
else:
    dirname = '/usr/lib/python'
import os, glob

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()
print allsizes[:2]
print 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, subHere, 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:])
```

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=''),
               dict(script='test2.py', args='spam')]

for testcase in testscripts:
```

```

        cmdline = '%(script)s %(args)s' % testcase
        output = os.popen(cmdline).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: кнопка, изменяющая цвет и размер
from Tkinter import *
import random
fontsize = 25
colors = ['red', 'green', 'blue', 'yellow', 'orange',
          'white', 'cyan', 'purple']

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"""
        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,
позволяет 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) # Get hdrs only
        print '-'*80
        print '[%d: octets=%d, size=%s]' % (msgnum, octets, msgsize)
        for line in hdrlines: print line

        if raw_input('Print?') in ['y', 'Y']:
            for line in server.retr(msgnum)[1]: print line # Получить сообщ.
        if raw_input('Delete?') in ['y', 'Y']:
            print 'deleting'
            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 браузером 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

# Сценарий для заполнения базы данных shelve объектами Python

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

Специальные символы

: символ двоеточия, 276, 377
#, комментарии, 365
% (остаток/формат) оператор, 147
& (побитовое И) оператор, 146
* (умножение) оператор, 147
/ (деление) оператор, 147
<< (сдвиг влево) оператор, 146
== (проверка равенства) оператор, 175, 257
^ (побитовое исключающее ИЛИ) оператор, 146
| (побитовое ИЛИ) оператор, 146
..., приглашение к вводу, 76
>> (сдвиг вправо) оператор, 146
>>>, приглашение к вводу, 74, 77

A

abs, функция, 159
__add__, метод, 587, 607, 608
__all__, переменная, 534
all, функция, 348
and, оператор, 146, 323
any, функция, 348
Apache, 47
apply, функция, 447
apply-подобный синтаксис вызова, 448
ArithmeticError, исключение (встроенное), 742
assert, инструкция, 275, 701, 725
 проверка соблюдения ограничений, 725
AssertionError, исключение, 725
AttributeError, исключение, 616

B

__base__, атрибут, 591

__bases__, атрибут, 628
bool, тип данных, 163
Boolean, 136
break, инструкция, 275, 332, 334
__builtin__, модуль, 159, 401, 404

C

C, язык программирования, интеграция, 50
C++, язык программирования, 40, 50
__call__, метод, 608, 620
chr, функция, 453
__class__, атрибут, 591, 628
class, инструкция, 275, 561, 578, 595
 атрибуты данных, 597
 вложенные инструкции, 596
 общая форма, 596
 пример, 596
 пространство имен, 596
 сравнение с именами в модуле, 596
 сравнение с функциями, 596
classtree, функция, 630
__cmp__, метод, 608
COM, поддержка в MS Windows, 44
contextlib, модуль, 730
continue, инструкция, 275, 332, 334
CPython, 65
csh, командная оболочка, 784

D

Dabo, 43
__debug__, флаг, 725
def, инструкция, 275, 387, 389, 599
 return, yield, 386
 и lambda, 441
del, инструкция, 275
__del__, метод, 608, 623

`__dict__`, атрибут, 590, 627, 649
`dict`, функция-конструктор, 356
`dir`, функция, 90, 121, 366
`distutils`, 498
Django, 44
`__doc__`, атрибут, 367, 657
Doctest, 769

E

Eclipse, интегрированная среда
 разработки, 98, 769
`else`, инструкция, 335
`__enter__`, метод, 729
`enumerate`, функция, 356
`env`, команда в UNIX, 82
`__eq__`, метод, 608
`eval`, функция, 64, 158, 432
 использование функции `eval` для
 строк в объектах, 249
excerpt, предложение инструкции `try`,
 710
Exception, исключение (встроенное),
 742
`exec`, инструкция, 275
`exec`, функция, 64
`execfile(module.py)`, встроенная
 функция, 92
`__exit__`, метод, 729

F

False, предопределенное имя, 163
FIFO, файлы, 252
`filter`, функция, 331, 454
`find`, метод, 205
`for`, циклы, 337
 `readlines`, метод, 346
 вложенные, 341
 внутри функции, 393
 и генераторы списков, 458
 итераторы, 345
 кортежи, 340
 переменные цикла, 340
 общий формат, 337
 строки, 339
 типичные варианты использования,
 339
`for/else`, инструкция, 275
`freeze`, 69, 770
`from *`, инструкция, 503, 550
`from`, инструкция, 275, 503, 549

и `import`, 506
модули, 485
потенциальные проблемы, 507
рекурсия, 553

G

`__getattr__`, метод, 608, 615, 648
`__getattribute__`, метод, только в
 классах нового стиля, 676
`__getitem__`, метод, 608
 реализует доступ к элементам по
 индексу, 609
`global`, инструкция, 275, 386, 388, 406
GTK, 43

H

`help`, функция, 121, 205, 369, 372
`hex`, функция, 158
HTML (Hyper Text Markup Language –
гипертекстовый язык разметки), 44

I

`__iadd__`, метод, 607, 608
IDLE, интегрированная среда
 разработки, 769
`if`, инструкции, 315
 множественное ветвление, 316
 общая форма, 315
 простые примеры, 316
`if/elif/else`, инструкция, 274
`if/else`, трехместное выражение, 325
`import as`, расширение, 539
`import`, инструкция, 87, 275, 488, 490,
 503
 `sys.path`, список, 494
 выбор файла модуля, 495
 выполняется только один раз, 504
 домашний каталог, 492
 дополнительные возможности
 выбора модуля, 496
 запуск, 497
 и `from`, 506
 и пространства имен, 91
 использование расширений имен
 файлов, 379
 каталоги стандартной библиотеки,
 492
 когда необходимо использовать, 508
 компиляция, 496

- модули, 485
- переменная окружения PYTHON-
PATH, 492
- примечания к использованию, 91
- содержимое файлов с расширением .
pht, 492
- __import__, функция, 496, 549
- in, оператор проверки на вхождение,
146, 331, 342, 348
- IndexError, исключение, 704
- Informix, система управления базами
данных, 45
- __init__, метод, 570, 588, 608
- __init__.py, файлы, 523
- int, функция, 158, 196, 286
- intersect, функция, 394
- IronPython, 44, 66
- is not, оператор, 146
- __iter__, метод, 608, 609
- iter, функция, 346, 461

J

- join, метод, 205, 207, 349
- Jython, 43, 65

K

- Komodo, интегрированная среда
разработки, 99, 769
- ksh, командная оболочка, 784

L

- lambda-выражения, 415, 440, 441
 - вложенные, 445
 - где используются, 442
 - и def, 441
 - операторы, 146
 - тело, 441
- lambda, инструкция, 386
- LEGB, правило, 401, 406, 511
- __len__, метод, 608
- len, функция, 117, 227, 387
- Linux, 47
- list, функция, 174, 207, 243, 349
- __lt__, метод, 608

M

- map, функция, 331, 348, 354, 449, 465
 - и генераторы списков, 459

- и функциональное
программирование, 449
- math, модуль, 116
- MFC, 43
- mod_python, 44
- __mul__, метод, 587
- MySQL, система управления базами
данных, 45

N

- __name__, атрибут, 535
- .NET, платформа, 44
- NLTK, пакет, 46
- None, объект, 259
- not in, оператор, 146
- not, оператор, 146
- NotImplementedError, исключение, 606
- NULL, указатель в языке C, 259
- NumPy, расширение, 45, 145

O

- oct, функция, 158
- ODBC, 45
- open, функция, 135, 136, 387
- __or__, метод, 608
- or, оператор, 146, 323
- Oracle, система управления базами
данных, 45
- ord, функция, 453
- OverflowError, исключение
(встроенное), 742

P

- Parrot, 70
- pass, инструкция, 275, 332, 333
- PATH, переменная окружения, 73, 782
- Perl, язык программирования, 47, 51
- pickle, модуль, 45, 249, 647
- PMW, 43, 69
- pop, метод, 230
- PostgreSQL, система управления базами
данных, 45
- pow, функция, 450
- print, инструкция, 74, 80, 274, 307
 - в интерактивном режиме, 76
 - и sys.stdout, 311
 - перенаправление потока вывода, 308
 - программа Hello World, 308
 - расширение print >> file, 309

- числа, 152
- Psyco, 771
 - динамический компилятор, 67
- .py, расширение имен файлов, 78, 95
- py2exe, 69, 770
- PyChecker, 768
- PyDev, модуль расширения, 98
- PyDoc, 121, 365, 768
 - help, функция, 365, 369, 372
 - отчеты в формате HTML, 365, 372
- pygame, 46, 69
- PyGTK, библиотека создания графического интерфейса, 69
- PyInstaller, 69, 770
- Pylons, 44
- PyPI, веб-сайты, 46
- PyPy, 70
- PySol, программа, 46
- Python
 - архитектура программы, 487
 - импортирование и атрибуты, 487
 - как организована программа, 487
 - в сравнении с другими языками программирования, 54
 - структура, 273
- Python 3.0, 447, 450, 452
 - nonlocal, инструкция, 411
 - инструкция print, 312
 - исключения, 712
 - классы, 669
 - сильные стороны языка, 46
 - смешивание типов, 222
- Python for .NET, 66
- PythonCard, интегрированная среда разработки, 99
- PYTHONPATH, переменная окружения, 492, 782
 - каталоги, 492
- PYTHONSTARTUP, переменная окружения, 782
- PythonWin, интегрированная среда разработки, 99
- PyUnit, 768

Q

Qt, 43

R

__radd__, метод, 607, 608, 619
raise, инструкция, 275, 701, 722

общие формы, 747
передача дополнительных данных, 723
передача сигналов из функций по условию, 756
random, модуль, 116, 159
range, функция, 297, 332, 350

- изменение списков, 352
- обход части последовательности, 352
- счетные циклы, 350

raw_input, функция, 285, 312, 756
readlines, метод, 346
reduce, функция, 331
reload, функция, 88, 514, 552

- вместе с инструкцией from, 550
- модули, 485
- основы использования, 515
- подробнее, 516
- пример использования, 516
- примечания к использованию, 91

replace, метод, 205
__repr__, метод, 608, 618, 621, 650
repr, функция, 147, 153, 196
return, инструкция, 275, 388, 389
round, функция, 159
Run Module, пункт меню в среде IDLE, 92

S

self, 579
set, функция, 136, 161
__setattr__, метод, 608, 615
__setitem__, метод, 608
Shedskin, 68, 771
shelve, модуль, 647
SIP, 44
__slots__, атрибут, в классах нового стиля, 673
sorted, функция, 131, 348, 379, 465
SQLite, 45
SQLObject, 45
Stackless Python, 70
StandardError, исключение (встроенное), 742
stdout, объект, 307
StopIteration, исключение, 461, 611
__str__, метод, 608, 618, 621
str, функция, 147, 153
sum, функция, 348, 465
SWIG, 44

Swing, 43
Sybase, система управления базами данных, 45
SyntaxError, исключение, 715
sys, модуль, 307
sys.exc_info, функция, 729, 759
sys.modules, словарь, 545
sys.path, список, 494, 538
sys.stdout, объект, 311

T

Tcl, язык программирования, 47
TCL_LIBRARY, переменная окружения, 782
Tkinter, 43, 69
 GUI
 API, 562
 графический интерфейс, 93
TK_LIBRARY, переменная окружения, 782
True, предопределенное имя, 163
try, инструкция
 обработка ошибок, 288
 отладка с помощью инструкции try, 757
try/else, инструкция, 713
try/except, инструкция, 701, 708, 718
 вложение, 720
 вложенная, 752
 пример использования объединенной инструкции, 720
try/except/else, инструкция, 709
try/except/finally, инструкция, 275
try/finally, инструкция, 701, 708, 716
 вложенная, 753
 программирование завершающих действий, 716
tuple, функция, 243, 349, 422
TurboGears, 44
type, функция, 393

U

unittest, модуль, 768
update, метод, 230

V

Vaults of Parnassus, веб-сайты, 46

Vista, установка Python 2.5 из установочного файла формата MSI, 780

W

WebWare, 44
while, циклы, 330
 имитация циклов while языка C, 338
 общий формат, 331
 примеры, 331
while/else, инструкция, 275
win32all, пакет расширений для Windows, 531
WingIDE, интегрированная среда разработки, 99
with/as, инструкция, 275, 701, 726
 протокол управления контекстом, 728
wxPython, 43, 69

X

xmlrpclib, модуль, 46

Y

yield, инструкция, 275

Z

zip, функция, 350, 354
 конструирование словарей, 355
ZODB, 45
Zope, 44

A

абсолютный импорт, 540
абстрактные суперклассы, 605
автоматическое присвоение расширений файлам в операционной системе Windows, 79
автоматическое управление памятью, 49
агрегирование, 643
адаптация через наследование, 563
альтернативные реализации
 CPython, 65
 IronPython, 66
 Jython, 65
анонимные функции, 440
аргументы
 значения по умолчанию, 427

- и разделяемые ссылки, 419
- извлечение аргументов из коллекции, 429
- изменяемые объекты передаются по указателю, 419
- комбинирование ключей и значений по умолчанию, 430
- неизменяемые объекты передаются по значению, 419
- передача в функции, 388
- передача по ключу, 426, 436, 448
- передача через операцию присваивания, 423
- применение функций, 447
- произвольное количество, 428
- режимы сопоставления
 - переменное число аргументов, 424
 - по ключам, 424
 - по позиции, 423
 - по умолчанию, 424
- сбор аргументов в коллекцию, 428
- функция поиска минимума, 430
- аргументы командной строки, 196
- аргументы передаются через автоматическое присваивание объектов локальным именам, 418
- аргументы по ключу
 - передача, 448
- аргументы со значениями по умолчанию, 414
- ассоциативные массивы, 225
- атрибуты, 487, 767
 - def, инструкция, 597
 - имена, 624
 - классов, 603
 - псевдочастные, 665
 - для чего нужны, 666
 - экземпляров, 602
- атрибуты данных, 597

Б

- базы данных, 45
- байт-код, 40
 - компиляция, 62
- библиотеки утилит, 50
- битовые операторы, 146
- блок else в циклах, 332
- быстрое создание прототипов, 45

В

- ввод составных инструкций, 76
- веб-сценарии, 44
- взаимодействие, 469
- взаимосвязи типа «имеет», 643
- взаимосвязи типа «является», 641
- включение будущих возможностей языка, 534
- вложенные модули, 96
- вложенные области видимости
 - lambda, 445
- вложенные функции, области видимости, 410
 - и lambda-выражения, 415
- вложенные циклы, 454
- внешние инструменты, 767
- возбуждение и обработка собственных исключений, 722
- возможность запуска программ из текстового редактора, 101
- восьмеричные литералы, 144, 158
- вспомогательные модули, 145
- встраивание
 - кода на языке Python в C, 44
 - расширение типов, 662
- встроенные
 - инструменты, 49
 - типы объектов, 49
- встроенные типы
 - зачем нужны, 112
 - расширение, 661
 - встраиванием, 662
 - наследованием, 663
- встроенные функции
 - abs, 159
 - all, 348
 - any, 348
 - apply, 447
 - chr, 453
 - dict, 356
 - dir, 121, 366
 - enumerate, 356
 - eval, 158, 432
 - execfile(module.py), 92
 - False, имя, 163
 - filter, 454
 - help, 121
 - hex, 158
 - __import__, 496, 549
 - int, 158, 196, 286

- iter, 346, 461
- len, 117, 387
- list, 174, 243, 349
- map, 342, 348, 354, 449, 465
- __name__, атрибут, 535
- oct, 158
- open, 135, 136, 387
- ord, 453
- pow, 450
- random, модуль, 159
- range, 297, 332, 350
- raw_input, 285, 312, 756
- reload, 514
 - пример использования, 516
- repr, 147, 153, 196
- round, 159
- set, 136, 161
- sorted, 131, 348, 465
- str, 147, 153
- sum, 348, 465
- sys.path, список, 538
- True, имя, 163
- tuple, 243, 349, 422
- type, 393
- zip, 350, 354
- длинные целые, 156
- принудительное преобразование
 - типов, 149
 - числовые литералы, 145
- вызов метода, 599, 600
- вызовы, 386, 390
 - встроенные, 100
 - обработка, 620
- вызовы функций, 274
- выполнение
 - PVM, 63
 - виртуальная машина Python, 63
- выполнение программы, 60
 - компиляция в байт-код, 62
 - производительность, 63
 - скорость разработки, 64
- выражения-генераторы, 465
- высокая скорость разработки, 38
- выходные параметры, имитация, 422

Г

- Гвидо ван Россума (Guido van Rossum), 669
- генераторы, 459
 - методы send и next, 463

- пример, 461
- генераторы списков, 348, 357, 452
 - выражения-генераторы, 465
 - и map, 459
 - и кортежи, 243
 - и матрицы, 456
 - и циклы for, 458
 - основы, 358, 453
 - проверки и вложенные циклы, 454
 - расширенный синтаксис, 360
 - файлы, 359
- гибкость кода
 - нарушение, 137
- гибкость объектов, 253
- глобальная область видимости, 399
- глобальные переменные, минимизация
 - количества, 407
- границы блоков, 319, 320
- границы инструкций, 319
- групповое присваивание, 293, 294, 298
 - разделяемые ссылки, 298

Д

- декораторы функций, 623, 649, 681
 - пример, 683
- делегирование, 648
- деление
 - истинное, 153
 - классическое, 153
 - с округлением вниз, 153
- дескрипторы файлов, 136, 252
- деструктор, 623
- динамическая типизация, 49, 114, 166
 - отсутствие инструкций объявления, 166
 - переменные, объекты и ссылки, 167
 - разделяемые ссылки, 171
 - и изменяемые объекты, 173
 - и равенство, 175
 - сборка мусора, 170
 - типы, 169
- динамический компилятор (just-in-time, JIT), 67
- длинные целые, 156
- домашний каталог программы, 492
- дополняющее присваивание, 299
 - преимущества, 300
 - разделяемые ссылки, 301

З

завершающие действия
 с помощью инструкции try/finally, 716
заключительные операции, 703
замыкания, 412
записи, в виде классов, 639
запуск Python
 интерпретатор, 61
запутать программный код, как не, 444
зарезервированные слова, 302
значения по умолчанию, 474

И

иерархии
 типов, 260
 понятий, 273
избегать воздействий на изменяемые аргументы, 421
извлечение среза, 180
изменения
 внутри функции аргумента, который является изменяемым объектом, 419
 в соседних модулях, минимизация, 408
 значений имен в других файлах, 506
изменяемые объекты, 474
 в операциях присваивания, 378
изменяемые типы, 212
имена атрибутов, 403
имитация частных атрибутов экземпляра, 617
импортирование
 и перезагрузка, 86
 и атрибуты, 487
 и области видимости, 512
 пример, 90
 примечания к использованию, 91
индексация, 117, 180
инкапсуляция, 573, 637
инструкции, 273
 в форме выражений, 444
 не имеющие вложенных инструкций, 323
 присваивания, 274
 разделители, 282, 321
 специальные случаи, 282
инструкции выражений, 305
 типичные, 305

инструкции импортирования
 импортирование модулей по имени в виде строки, 548
 синтаксис относительного импорта, 539
инструкции присваивания, 292, 293
 групповые, 293, 298
 разделяемые ссылки, 298
 дополнительные варианты, 296
 дополняющие, 299
 преимущества, 300
 разделяемые ссылки, 301
 комбинированные, 293
 кортежей и списков, 293
 последовательностей, 293
инструменты разработки крупных проектов, 767
интеграция
 компонентов, 44
 с языком C, 50
интегрированные среды разработки, 769
 Eclipse, 98
 Komodo, 99
 PythonCard, 99
 PythonWin, 99
 WingIDE, 99
интерактивные циклы, 284
 вложенный программный код, 289
 математическая обработка данных пользователя, 285
 пример простого цикла, 284
 проверка ввода, 286
интерактивный режим, 72
 ввод составных инструкций, 76
 изменение строки приглашения к вводу, 76
 использование, 75
 отступы, 76
 тестирование программного кода, 74
интернационализация, 187
Интернет-модули, 44
интерпретатор, 58
 соглашения по именованию, 303
интерфейсы, программные, 43
интроспекция, 545
информация о состоянии, 138
исключения, 701, 767
 AssertionError, 725
 except, предложение инструкции try, пустое, 710
 IndexError, 704

- SyntaxError, 715
 - вложенные обработчики
 - исключений, 751
 - вложение в потоке управления, 753
 - синтаксическое вложение, 754
 - возбуждение и обработка
 - собственных исключений, 722
 - встроенные
 - ArithmeticError, 742
 - Exception, 742
 - OverflowError, 742
 - StandardError, 742
 - перехват, 715
 - заключительные операции, 703
 - идиомы, 756
 - sys.exc_info, 759
 - запуск тестов в рамках единого процесса, 758
 - исключения не всегда являются ошибками, 756
 - отладка с помощью внешних инструкций try, 757
 - передача сигналов из функций по условию, 756
 - использование, 751
 - на основе классов, 735
 - в сравнении с исключениями на основе строк, 736
 - определение текста исключения, 743
 - передача данных и поведения в экземплярах, 744
 - преимущества, 738
 - пример, 736
 - на основе строк, 734
 - назначение, 703
 - необычное управление потоком выполнения, 703
 - обработка ошибок и особых ситуаций, 703
 - обработчик исключений по умолчанию, 705
 - повторное возбуждение исключений с помощью инструкции raise, 724
 - пример, 704
 - развитие, 702
 - советы по применению исключений, 760
 - избегайте пустых предложений excerpt, 761
 - используйте категории, 763
 - что должно быть обернуто, 760
 - типичные проблемы, 764
 - уведомления о событиях, 703
 - искусственный интеллект, 46
 - исполняемые сценарии в UNIX, 81
 - источники документации, 364
 - dir, функция, 365, 366
 - __doc__, атрибут, 367
 - help, функция, 365
 - PyDoc
 - help, функция, 369, 372
 - отчеты в формате HTML, 372
 - веб-ресурсы, 365, 376
 - комментарии #, 365
 - отчеты в формате HTML, 365
 - печатные издания, 365, 376
 - стандартный набор руководств, 365, 375
 - строки документирования, 365, 367, 368
 - определяемые пользователем, 367
 - стандарты, 368
 - исходный код, компиляция, 62
 - итераторы, 342, 459
 - StopIteration, исключение, 345
 - выражения-генераторы, 465
 - другие контексты итераций, 347
 - других встроенных типов, 346
 - и встроенные типы, 464
 - и сканирование файлов, 343
 - итерационный протокол, 345
 - несколько итераторов в одном объекте, 613
 - определяемые пользователем, 349, 610
 - файлов, 344
 - итерации
 - альтернативы, 466
 - хронометраж, 466
 - перегрузка, 609
 - протокол, 132
 - итерационный протокол, 345
- ## К
- кавычки, 122
 - каналы, 136
 - каталоги стандартной библиотеки, 492
 - категории, на основе классов, 763

- категории типов, 211, 252
 - отображения, 212
 - последовательности, 212
 - числа, 211
- квадратные скобки, 282
- классификация объектов, 252
- классы, 562, 565, 567
 - `__bases__`, атрибут, 628
 - атрибуты, 578, 603
 - возможности
 - адаптация через наследование, 563
 - композиция, 562
 - множество экземпляров, 563
 - наследование, 562
 - перегрузка операторов, 563
 - вызовы методов классов, 567
 - и их хранение, 647
 - и модули, 659
 - как записи, 639
 - множественное наследование, 649
 - модули, 584
 - нового стиля, 668
 - `__getattr__`, метод, 676
 - `__slots__`, атрибут, 673
 - другие расширения, 673
 - ромбоидальное наследование, 669
 - пример, 670
 - явное разрешение конфликтов имен, 671
 - свойства, 674
 - статические методы и методы класса, 673
 - операции присваивания внутри инструкции `class`, 578
 - определяемые пользователем, 138
 - основы программирования, 577
 - вызов объектов, 579
 - множество экземпляров, 577
 - `self`, 579
 - конкретные элементы, 579
 - наследование, 579
 - поведение по умолчанию, 578
 - пример, 579
 - переопределение операторов языка Python, 586
 - пример, 587
 - переопределение операторов языка Python, 586
 - пример, 587
 - программирование, 595
 - пример, 631
 - псевдочастные атрибуты, 665
 - для чего нужны, 666
 - смеси, 650
 - создание деревьев классов, 568
 - типичные проблемы, 684
 - изменение атрибутов, 684
 - методы, классы и вложенные области видимости, 687
 - многослойное обертывание, 689
 - множественное наследование, 685
 - шаблоны проектирования, 637, 661
 - перегрузка сигнатур вызова, 638
 - ключевые аргументы, 436
 - код
 - компиляция, 62
 - командная строка интерпретатора Python, 77
 - командные оболочки, 39
 - комбинируемая инструкция присваивания, 293
 - комбинируемое присваивание, 294
 - комментарии, 78, 150, 319
 - компилируемые расширения, 767
 - компиляция
 - Psyco, динамический компилятор, 67
 - Shedskin, 68
 - в байт-код, 62
 - комплексные числа, 144, 157
 - композиция, 562, 643
 - пример, 643
 - компоненты
 - интеграция, 44
 - конец строки, 277
 - конкатенация, 118, 180
 - строка, 190
 - концепции проектирования
 - модулей, 543
 - функций, 469
 - копии и ссылки, 254
 - короткая схема вычислений, 324
 - кортежи, 90, 113, 134, 240
 - массив ссылок на объекты, 240
 - в действии, 241
 - доступ к элементам по смещению, 240
 - запятые и круглые скобки, 241
 - и генераторы списков, 243
 - литералы кортежей и операции, 241

- неизменяемые последовательности, 240
- преобразования и неизменяемость, 242
- упорядоченные коллекции объектов произвольных типов, 240
- уровни вложенности, 240
- циклы `for`, 340
 - переменные цикла, 340
- косвенный вызов, 471
- круглые скобки, 148, 277, 283, 378

Л

- лексическая область видимости, 399
- литералы
 - длинных целых чисел, 144
 - чисел, 143
 - чисел с плавающей точкой, 143
 - шестнадцатеричные и восьмеричные, 144
- логические значения, 163, 327
- логические операторы, 146, 155
- локальная область видимости, 399
- локальные имена, 472
- локальные переменные, 395

М

- максимизировать многократное использование программного кода, 386
- математическая обработка данных пользователя, 285
- математические и научные вычисления, 45
- математические функции, 145
- матрицы и генераторы списков, 456
- метапрограммы, 544
- методы, 119, 599
 - `find`, 205
 - `join`, 205, 207
 - `replace`, 205
 - возможности, 601
 - вызов, 204, 599, 600
 - вызов конструкторов суперклассов, 601
 - несвязанные, 655
 - пример, 600
 - проверка наличия подстроки в конце, 209
 - проверка содержимого, 209

- связанные, 658
 - экземпляра, 655
- способы вызова методов, 600
- строки, 204
 - экземпляра, 680
- методы `send` и `next`, 463
- методы класса, 673, 677
 - и методы экземпляра, 680
 - использование, 679
 - несвязанные, 677
- минимизировать избыточность программного кода, 386
- мнимая часть, 157
- многослойное обертывание в классах, 689
- множества, 136, 161
- множественное ветвление, 316
- множественное наследование, 649, 685
- множество экземпляров, 563
- модель отношений реального мира, 641
- модули, 77, 485, 767
 - `from *`, инструкция, 503
 - `from`, инструкция, 485, 503
 - потенциальные проблемы, 507
 - `import as`, расширение, 539
 - `import`, инструкция, 485, 503
 - выполняется только один раз, 504
 - `reload`, функция, 485
 - `sys.path`, список, 494
 - вложенные, 96
 - выбор файла модуля, 495
 - домашний каталог программы, 492
 - дополнительные возможности выбора, 496, 533
 - и использование инструкции `import`, 508
 - и классы, 584, 659
 - и пространства имен, 91
 - изменение значений имен в других файлах, 506
 - изменение пути поиска, 538
 - имена файлов, 501
 - использование, 502
 - каталоги стандартной библиотеки, 492
 - концепции проектирования, 543
 - взаимозависимость, 543
 - интроспекция, 545
 - согласованность, 543
 - метапрограммы, 544
 - назначение, 486

- определение, 501
- основы программирования, 501
- переменная окружения PYTHON-
PATH, 492
- повторная загрузка, 514
- повторное использование
программного кода, 486
- присваивание, 505
- пространства имен, 509
 - вложенные, 513
 - дополнение имен атрибутов, 511
- путь поиска модулей, 491
- разделение системы пространств
имен, 486
- реализация служб или данных для
совместного пользования, 486
- синтаксис относительного импорта,
539
- смешанные режимы использования,
535
 - тестирование модулей, 536
- содержимое файлов с расширением .
pht, 492
- создание, 501
- сокрытие данных, 533
 - __all__, переменная, 534
 - предотвращение копирования,
534
- типичные проблемы, 547
 - from *, инструкция, 550
 - from, инструкция, 549
 - reload, функция, 550, 552
 - импортирование модулей по
имени в виде строки, 548
 - порядок следования инструкций,
547
 - рекурсивный импорт
инструкцией from, 553
 - тестирование в интерактивной
оболочке, 551
 - эквивалентность инструкций import
и from, 506
- модули расширений, 502
- модульное программирование, 49

Н

- наследование, 562, 582, 600, 602, 637,
641
 - __init__, конструктор, 738
- абстрактные суперклассы, 605

- иерархия, 561
- и атрибуты классов, 582
- и обращение object.attribute, 582
- множественное, 649, 685
- модель поиска в дереве
наследования, 603
- переопределение унаследованных
методов, 603
- подклассы и суперклассы, 582
- приемы организации
взаимодействия классов, 604
- пример, 583, 670
- ромбоидальное, 669
 - пример, 670
 - явное разрешение конфликтов
имен, 671
- создание дерева атрибутов, 602
- специализация унаследованных
методов, 603
- суперклассы, 582
- экземпляров, 582
- явное разрешение конфликтов имен,
671
- настройка переменных окружения, 781
- неизменяемость (строки), 119
- необычное управление потоком
выполнения, 703
- несвязанные методы класса, 655, 677
- несоставные инструкции, 323
- неформатированные строки, 180, 181,
184
 - подавление экранирования, 184

О

- обертывание, 760
- области видимости, 398, 624
 - lambda-выражения, 415
 - LEGB, правило, 401
 - аргументы со значениями по
умолчанию, 414
 - вложенные, 687
 - встроенная, 401, 404
 - глобальные, 400
 - замыкания, 412
 - и вложенные функции, 410
 - изменения в соседних модулях,
минимизация, 408
 - локальные, 400
 - объемлющие, 410
 - объемлющих инструкций def, 401

- основы, 399
- охранение состояния объемлющей
 - области видимости с помощью аргументов по умолчанию, 414
- пример, 403
- присваивание именам, 400
- произвольное вложение, 418
- рекурсия, 400
- фабричные функции, 412
- обработка
 - особых ситуаций, 703
 - ошибок, 703
 - проверкой ввода, 286
- обработчик исключений по умолчанию, 705
- обработчики событий, 621
- объединение разделенных инструкций, 323
- объектно-ориентированный язык сценариев, 38
- объекты, 168
 - вызов, 579
 - два стандартных поля, 169
 - и функции, 471
 - несвязанные методы класса, 655
 - связанные методы экземпляра, 655
 - система хранения, 45
 - состояние и поведение, 578
 - строковые методы, 204
 - числа, 142, 150
- объекты исключений, 733
 - на основе классов, 733
 - на основе строк, 733
- объемлющие инструкции `def`, 401
- объемлющие области видимости, 410
- ООП (объектно-ориентированное программирование), 46, 561, 564
 - взаимосвязи типа «имеет», 643
 - взаимосвязи типа «является», 641
 - вызовы методов классов, 567
 - зачем нужны классы, 562
 - адаптация через наследование, 563
 - композиция, 562
 - множество экземпляров, 563
 - перегрузка операторов, 563
 - инкапсуляция, 573
 - классы, 565, 567
 - многократное использование программного кода, 571
 - платформы, 574

- поиск унаследованных атрибутов, 564
- создание деревьев классов, 568
- фабрики, 653
 - зачем нужны, 654
- шаблоны проектирования, 574
- экземпляры, 565, 567
- операторы выражений, 145
 - группировка подвыражений, 148
 - определение старшинства, 147
 - перегрузка, 149
 - смешивание операторов, 147
 - смешивание типов, 148
 - числа, 150
- операции
 - над строками, 190
- операции присваивания
 - внутри инструкции `class`, 578
 - внутри функции, 418
- оригинальный модуль `string`, 209
- остаток от деления, оператор, 147
- открытое программное обеспечение, 47
- отладчики, 770
 - `IDLE`, 97
- отображения, 212
- отрицания оператор, 147
- отрицательные смещения в строках, 192
- отступы, 320, 377
 - конец, 278
 - правила оформления, 278
- отсутствие инструкций объявления, 166
- очереди, 136
- ошибки при работе с функциями, 472
 - значения по умолчанию, 474
 - локальные имена, 472
 - переменные цикла в объемлющей области видимости, 476
 - функции, не возвращающие результат, 476

П

- пакеты модулей, 521
 - инструкции `from` и `import`, 526
 - когда используется операция импортирования, 527
 - основы операции импортирования, 521
 - `__init__.py`, файлы, 523
 - инициализация пакета, 524

- инициализация пространства имен модуля, 524
- настройка пути поиска, 522
- поведение инструкции `from *`, 524
- точечный путь, 522
- пример импортирования, 525
- перегрузка операторов, 253, 563, 606
- имитация частных атрибутов экземпляра, 617
- итераторы
 - несколько итераторов в одном объекте, 613
 - определяемые пользователями, 610
- ключевые идеи, 607
- общие методы, 608
 - `__add__`, 608
 - `__call__`, 608
 - `__cmp__`, 608
 - `__del__`, 608
 - `__eq__`, 608
 - `__getattr__`, 608
 - `__getitem__`, 608
 - `__iadd__`, 608
 - `__init__`, 608
 - `__iter__`, 608
 - `__len__`, 608
 - `__lt__`, 608
 - `__or__`, 608
 - `__radd__`, 608
 - `__repr__`, 608
 - `__setattr__`, 608
 - `__setitem__`, 608
 - `__str__`, 608
- основные идеи, 586
- перегрузка сигнатур вызова, 638
- передача аргументов, 398, 418
- переменные, 168
 - DOS, 784
 - и основные выражения, 150
 - использование, 167
 - окружения, 781
 - создание, 167
 - типы, 167
 - цикла, 416
- переносимый ANSI C, 48
- переносимый прикладной программный интерфейс баз данных, 45
- перехват обращения к атрибутам, 615
- перечислимые типы данных, 297
- платформы, 574
- поведение по умолчанию, 578, 714
- повторение
 - последовательности, 263
 - строки, 118
- повторное возбуждение исключений с помощью инструкции `raise`, 724
- повторное использование программного кода, 486
- подавление экранированных последовательностей, 184
- подвыражения, 148
- подклассы, 565
 - замещение унаследованных атрибутов, 603
 - расширение встроженных типов, 663
- поиск унаследованных атрибутов, 564
- полиморфизм, 118, 138, 149, 392, 637
 - `intersect`, функция, 394
- положительные смещения в строках, 191
- получение среза, 117
- пользовательский интерфейс IDLE, 92
 - Tkinter, 93, 96
 - вложенные модули, 96
 - дополнительные возможности, 97
 - запуск в Windows, 93
 - запуск редактируемых файлов, 94
 - запуск сценариев, 96
 - многопоточные программы, 96
 - настройка, 96
 - отладчик, 97
 - очистка экрана, 96
 - ошибки соединения, 97
 - повторное выполнение команд, 94
 - подсветка синтаксиса, 94
 - файлы с исходным текстом, 94
- поразрядный сдвиг, чисел, 155
- последовательности, 116, 179, 212
 - отображение функций на, 449
- почему программисты используют Python, 35
 - высокая скорость разработки, 36
 - интеграция компонентов, 36
 - качество программного обеспечения, 35
 - переносимость программ, 36
 - поддержка библиотек, 36
 - удовольствие, 37
- правила именования переменных, 301
- предложения инструкции `try`, 710
- формы, 710

- предупреждения, 756
- преобразование строк, 147
- приглашение к вводу, 74
- приемы организации взаимодействия классов, 604
- приемы программирования циклов, 349
- приложения баз данных, 45
- пример использования объединенной инструкции try, 720
- присваивание, 505
 - классификация имен, 625
 - именам, 400
 - кортежей и списков, 293, 294
 - последовательностей, 293, 294
 - дополнительные варианты, 296
- пробелы, 319
- проверка
 - ввода, 286
 - и вложенных циклов, 454
 - ошибок, 730
 - соблюдения ограничений, 725
- проверка истинности, 257, 323
 - короткая схема вычислений, 324
 - оператор and, 325
 - оператор or, 324
 - операторы and и or, 323
 - понятия Истина и Ложь, 259
- программа Hello World, 308
- программирование
 - интерпретатор, 58
 - модульное, 49
 - системное, 43
 - строковые методы, 204
- программные ловушки импорта, 496
- программный код
 - вложенный, 289
 - многократное использование, 571
 - повторное использование, 486
 - самый простой класс на языке Python, 589
 - типичные ошибки
 - программирования, 377
- производительность, 63
- произвольное вложение областей видимости, 418
- пространства имен, 88, 91, 398, 486, 509, 624, 694
 - вложенные, 513
 - дополнение имен атрибутов, 511
 - имена атрибутов, 624
 - классификация имен, 625

- классы, 596
- простые имена, 624
- разделение системы пространств имен, 486
- словари, 627
- составные имена атрибутов, 624
- ссылки, 630
- пространства имен, объявление, 406
- профилировщики, 769
- процедурная декомпозиция, 386
- псевдочастные атрибуты, 665
 - для чего нужны, 666
- пустые предложения except, 710, 761
- пустые строки, 180, 319

Р

- равенство, 257
- разделение системы пространств имен, 486
- разделяемые ссылки, 171
 - и аргументы, 419
 - и изменяемые объекты, 173
 - и равенство, 175
- расширение встроенных типов, 661
 - встраиванием, 662
 - наследованием, 663
- расширения на языке Python, 766
- расширения файлов, 79
- режимы сопоставления, 423, 435
 - переменное число аргументов, 424
 - по ключам, 424
 - по позиции, 423
 - по умолчанию, 424
- рекурсия, 400, 630
- ромбоидальное наследование, 669
 - пример, 670
 - явное разрешение конфликтов имен, 671

С

- сборка мусора, 49, 130, 170
- свободное программное обеспечение, 47
- свойства класса, 674
- связанные методы экземпляра, 655, 658
- связность, 469
- сдвига операторы, 146
- символы,
 - крупные наборы, 187
 - подчеркивания, 534
- синтаксис относительного импорта, 539

синтаксические правила, 318
системная командная строка и файлы, 77
 исполняемые сценарии в UNIX, 81
 использование, 79
сканирование файлов, 343
словари, 113, 127, 224
 len, функция, 227
 pop, метод, 230
 update, метод, 230
 базовые операции, 227
 вложенность, 129
 дополнительные методы, 229
 доступ к элементам по ключу, а не по индексу, 225
 другие способы создания, 235
 замечания по использованию, 231
 изменение, 228
 имитация гибких списков, 232
 интерфейсы словарей, 236
 использование в качестве записей, 234
 итерации и оптимизация, 132
 как избежать появления ошибок обращения к несуществующему ключу, 233
 ключи, 128
 литералы и операции, 226
 неупорядоченные коллекции произвольных объектов, 225
 операции над последовательностями, 231
 операции отображения, 128
 отсутствующие ключи, 133
 переменная длина, гетерогенность и произвольное число уровней вложенности, 225
 пространств имен, 627
 сортировка по ключам, 130
 структуры разреженных данных, 233
 таблица языков, 230
 таблицы ссылок на объекты (хеш-таблицы), 226
словари для хранения записей, 639
сложение, правостороннее, 619
службы или данные для совместного пользования, 486
смешивание типов, операторы выражений, 148
соглашения по именованию, 303

содержимое файлов с расширением .pht, 492
создание дерева атрибутов, 602
сокеты, 136, 252
составные инструкции, 315
составные типы, 471
специальные режимы сопоставления, 423
специальный случай оформления блока, 283
списки, 113, 123, 215
 базовые операции, 218
 вложенные, 125
 генераторы списков, 126
 доступ к элементам по смещению, 216
 изменение, 220
 индексы, 219
 литералы и операции, 217
 массивы ссылок на объекты, 216
 матрицы, 219
 методы, 221
 специфичные для типа, 124
 операции над последовательностями, 124
 относятся к категории изменяемых объектов, 216
 присваивание по индексам, 220
 присваивание срезам, 220
 проверка выхода за границы, 125
 срезы, 219
 упорядоченные коллекции объектов произвольных типов, 215
 часто используемые операции, 224
способы доступа к глобальным переменным, 409
способы оптимизации, 771
сравнение, 257
 операторы, 146
 языков, 54
средства оптимизации, 67
 Psyco, динамический компилятор, 67
 Shedskin, 68
ссылки, 168
 и копии, 254
 на пространства имен, 630
стандартная библиотека, 490
 каталоги, 492
статически вложенные области видимости, 411

статические методы, 673, 677
 использование, 679
стратегический режим, 562
строгая типизация, 114
строки, 113, 116, 179
 dir, функция, 121
 help, функция, 121
 базовые операции, 190
 в апострофах и в кавычках, 181
 в кавычках, 122, 180
 в тройных кавычках, 186
 вызов методов, 180
 другие способы представления, 122
 извлечение среза, 192
 изменение, 199
 индексирование, извлечение среза,
 длина, 180, 191
 инструменты преобразования, 195
 конкатенация, 118, 180
 литералы, 180
 в апострофах, 181
 в кавычках, 181
 в тройных кавычках, 181
 в Юникоде, 181
 экранированные
 последовательности, 181, 182
 Юникод, 187
 методы, 204
 вызов функции, 204
 извлечение атрибутов, 204
 специфичные для типа, 119
наиболее типичные литералы строк и
 операции, 180
неизменяемость, 119
неформатированные, 180, 184
обход в цикле, проверка на
 вхождение, 180
операции над последовательностями,
 116
повторение, 118
поиск, 180
 по шаблону, 123
 с заменой, 180
преобразование регистра символов,
 180
проверка содержимого, 180
пустые, 180
разбиение на подстроки по символу-
 разделителю, 180
расширенная операция извлечения
 среза, 194

 символы обратного следа, 183
 удаление пробелов, 180
форматирование, 180, 200
 дополнительные возможности,
 201
 спецификаторы формата, 202
 строки из словаря, 203
 циклы for, 339
строки документирования, 319, 367,
 368, 657, 768
 определяемые пользователем, 367
 основное преимущество, 658
 стандарты, 368
структура организации программ, 487
структуры
 данных, 112
 разреженных данных, 233
суперклассы, 565, 604
 абстрактные, 605
 классы-смеси, 650
 конструкторы, 601
 наследование, 582
 расширение методов, 603
суффиксы
 комплексные числа, 157
сценарии, 77
счетчик ссылок, 169

Т

тактический режим, 562
текст
 изменение строк, 199
 разбор, 207
 строковые методы, 204
текущий рабочий каталог, 493
тестирование
 в интерактивной оболочке, 551
 запуск тестов в рамках единого
 процесса, 758
 программного кода в интерактивном
 режиме, 74
типичные проблемы, 764
 перехват ненужных исключений,
 766
 соответствие строковым
 исключениям, 764
типы объектов, 49, 111, 767
 базовые, 113
 встроенные, 112
 изменяемые, 212

- в модулях стандартной библиотеки, 262
- ловушки, 262
 - неизменяемые типы, 264
 - повторение последовательности, 263
 - присваивание, 262
 - циклические структуры данных, 264
- объекты и переменные, 169
- преобразование, 148
- сравнение, 258
- точность представления чисел и длинные целые, 144

У

- уведомления о событиях, 703
- удобство в использовании, 39
- указатель типа, 169
- унарные операторы, 147
- универсальные функции, 433
- упакованные двоичные данные,
 - сохранение в файлах и интерпретация, 250
- управление контекстом, 726
 - протокол, 728
- управляющий язык, 39
- упражнения
 - решения, 786
 - from/from *, 804
 - __man__, 805
 - try/except, 816
 - аргументы, 798
 - взаимодействие, 786
 - вложенное импортирование, 805
 - вложенные структуры, 794
 - генераторы списков, 802
 - дополнительные примеры на сопоставление аргументов, 800
 - импорт пакетов, 806
 - индексирование и извлечение среза, 790
 - индексирование словарей, 792
 - индексирование строк, 794
 - индексирование, извлечение среза и инструкция del, 791
 - классификация животных в зоологии, 814
 - ключи словарей, 792
 - композиция, 814

- кортежи, 791
- методы метакласса, 811
- модули, 786
- наследование, 807
- неизменяемые типы, 794
- обработка ошибок, 817
- общие операции, 793
- объекты исключений и списки, 816
- объекты множеств, 811
- основы импортирования, 803
- основы циклов, 795
- основы, 789
- основы, 797
- ошибки, 787
- перегрузка операторов, 809
- передача аргументов по ключу, 799
- переменное число аргументов, 798
- повторная загрузка, 806
- подклассы, 810
- прерывание программы, 788
- примеры для самостоятельного изучения, 818
- программирование
 - альтернативной логики, 796
- программы, 786
- связи в дереве классов, 813
- символы обратного следа, 796
- снова простые числа, 801
- сортировка словарей, 796
- сценарии, 787
- сценка с мертвым попугаем, 815
- файлы, 794
- функция dr, 795
- циклический импорт, 807
- установка и настройка Python, 59, 777
 - где получить, 778
 - настройка, 780
 - настройка переменных окружения, 781
 - переменные DOS, 784
 - проверка наличия предыдущей установки, 777
 - установка, 778
 - установка интерпретатора, 777
 - файлы путей, 785
- установка поддержки Tkinter в Linux, 782
- утилиты сторонних разработчиков, 50

Ф

фабрики, 653
 зачем нужны, 654
фабричные функции, 412
файлы, 113, 135, 244
 FIFO, 252
 в действии, 247
 сохранение и интерпретация
 упакованных двоичных данных
 в файлах, 250
 сохранение объектов с помощью
 модуля pickle, 249
 вызов метода close, 246
 выполнение, 61
 выталкивание выходных буферов на
 диск, 245
 дескрипторы, 252
 другие инструменты, 251
 другие средства, напоминающие
 файлы, 136
 закрытие файла вручную, 245
 запись всех строк из списка в файл,
 245
 запись строки в файл, 245
 изменение текущей позиции в файле,
 245
 каналы, 252
 операции над файлами, 245
 открытие, 244
 с доступом по ключу, 136, 252
 создают пространства имен, 509
 сохранение и интерпретация
 объектов Python в файлах, 247
 чтение
 следующей текстовой строки, 245
 следующих N байтов, 245
 файла целиком в единственную
 строку, 245
 файла целиком в список строк,
 245
файлы путей, 785
фигурные скобки, 282
фиксированные двоичные файлы, 68
фиксированные исполняемые двоичные
 файлы, 101
функции, 385, 398, 767
 apply, 447
 filter, 451
 lambda, 440
 reduce, 451

 анонимные, 440
 вызов, 379, 390
 выполняющие непосредственные
 изменения в объектах, 378
 для работы с числами, 159
 инструкции и выражения, имеющие
 отношение к функциям, 386
 как объекты, 471
 концепции проектирования, 469
 косвенный вызов, 471
 обратного вызова, 446
 обработчики, 442
 определение, 385
 передача сигналов из функций по
 условию, 756
 пересечение последовательностей,
 393
 вызов, 394
 локальные переменные, 395
 определение, 393
 полиморфизм, 394
 цикл for внутри функции, 393
 применение к аргументам, 447
 расширенные возможности, 440
 создание функций, 387
 def, инструкция, 387, 389
 global, инструкция, 388
 return, инструкция, 388, 389
 аргументы, 388
 функции обратного вызова, 658
функциональное программирование,
 451
 filter, функция, 451
 reduce, функция, 451
функциональные интерфейсы, 621
функция поиска минимума, 430

Х

хеши, 225

Ц

целые числа, 143
 литералы, 143
циклические структуры данных, 264
циклы, 330
 break, инструкция, 332, 334
 continue, инструкция, 332, 334
 else, инструкция, 335
 for, 337
 pass, инструкция, 332, 333

- readlines, метод, 346
- while, 330
 - общий формат, 331
 - примеры, 331
- блок else в циклах, 332
- имитация циклов while языка C, 338
- интерактивные, 284
 - вложенный программный код, 289
 - математическая обработка данных пользователя, 285
 - проверка ввода, 286
 - простой пример, 284
- итераторы, 342
- общий формат, 333
- параллельный обход, 354
- приемы программирования, 349
 - изменение списков, 352
 - обход части последовательности, 352
- счетные циклы, 350

Ч

- числа, 113, 142, 211
 - базовые операторы, 150
 - инструкция print, 152
 - комплексные числа, 157
 - литералы, 143
 - с плавающей точкой, 143
 - с фиксированной точностью, 137, 160
 - смешивание числовых типов в выражении, 148
 - форматы отображения, 152

Ш

- шаблоны проектирования, 574, 637, 661
- шестнадцатеричные
 - литералы, 158
- шестнадцатеричные литералы, 144

Э

- экземпляры, 565, 567
- __class__, атрибут, 628
- атрибуты, 602
 - имитация частных атрибутов, 617
- множество, 563
- передача данных и поведения в экземплярах, 744

- экранированные последовательности, подавление, 184

Я

- языки
 - сравнение, 54
- ярлыки, щелчок, 83
- raw_input, функция, 84
- в Windows, 83
- ограничения, 86

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-138-7, название «Изучаем Python, 3-е издание» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.