O'REILLY®

3-е издание
Рассмотрены версии
Python 2.7, 3.5 и новинки версии 3.6

Python
Справочник

ПОЛНОЕ ОПИСАНИЕ ЯЗЫКА

Алекс Мартелли,
Анна Рейвенскрофт, Стив Холден

Python

Справочник

ПОЛНОЕ ОПИСАНИЕ ЯЗЫКА

3-Е ИЗДАНИЕ

Python

IN A NUTSHELL

THIRD EDITION

Alex Martelli, Anna Ravenscroft, and Steve Holden

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Python

Справочник

ПОЛНОЕ ОПИСАНИЕ ЯЗЫКА

3-Е ИЗДАНИЕ

Алекс Мартелли, Анна Рейвенскрофт,
Стив Холден

ДИАЛЕКТИКА

Москва · Санкт-Петербург
2019

ББК 32.973.26-018.2.75
Х36
УДК 681.3.07

Компьютерное издательство "Диалектика"
Перевод с английского канд. хим. наук *А.Г. Гузикевича*
Под редакцией *В.Р. Гинзбурга*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Мартелли, Алекс, **Рейвенскрофт**, Анна, **Холден**, Стив

М29 Python. Справочник. Полное описание языка, 3-е издание. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 896 с. : ил. — Парал. тит. англ.

ISBN 978-5-6040723-8-7 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Алекс Мартелли, Анна Рейвенскрофт, Стив Холден

Python. Справочник. Полное описание языка

3-е издание

Подписано в печать 20.09.2018. Формат 70x100/16
Гарнитура Times
Усл. печ. л. 72,24. Уч.-изд. л. 50,6
Тираж 500 экз. Заказ № 9109

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-6040723-8-7 (рус.)
ISBN 978-1-449-39292-5 (англ.)

Оглавление

Содержание

Об авторах

Алекс Мартелли занимается программированием более 30 лет, из которых последние 15 лет в основном связаны с Python. Он автор первых двух изданий книги *Python in a Nutshell* и один из авторов первых двух изданий книги *Python Cookbook*. Алекс — номинированный член организации PSF (Python Software Foundation), обладатель наград *2002 Activators' Choice Award* и *2006 Frank Willison Memorial Award* за большой вклад в сообщество Python. Он активно публикуется на сайте Stack Overflow и часто выступает с докладами на технических конференциях. Вместе с женой Анной в течение 12 лет живет в Силиконовой долине, все это время работая в компании Google. В настоящее время отвечает за долгосрочную техническую поддержку облачной платформы Google Cloud.

Анна Мартелли Рейвенскрофт — опытный лектор и специалист по обучению персонала для самых разных областей, включая разработку веб-приложений для медицинских и образовательных сайтов, а также автор и рецензент технических книг, статей и презентаций. Не будучи профессиональным программистом, Анна — большой энтузиаст Python и активно участвует в движении сторонников открытого ПО: она номинированный член организации PSF и обладатель награды *2013 Frank Willison Memorial Award* за большой вклад в сообщество Python, а также один из авторов второго издания книги *Python Cookbook*. Живет в Силиконовой долине вместе с мужем Алексом в компании двух собак, кота и восьми кур.

Стив Холден, отметивший полувековой юбилей своей карьеры программиста выпуском этой книги, на протяжении последних 20 лет использует Python в качестве своего основного языка программирования. В 2002 году Стив написал книгу *Python Web Programming*. В 2003–2005 годах он председательствовал на первых трех конференциях PyCon, проходивших в США. В течение ряда лет был одновременно директором и председателем правления организации Python Software Foundation, номинированным членом которой продолжает оставаться в настоящее время. В 2007 году Стив был удостоен награды *Frank Willison Memorial Award* за большой вклад в сообщество Python. В настоящее время живет в Лондоне, где работает техническим директором в небольшом стартапе.

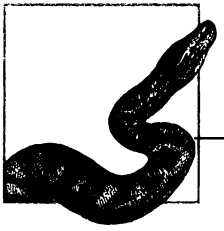
Об изображении на обложке

Животное на обложке книги — это африканский *иероглифовый питон*, или *скальный питон* (лат. *Python sebae*), представитель рода настоящих питонов. Питоны — семейство неядовитых змей, заглатывающих добычу целиком. Они обитают в тропических регионах Африки, Азии, Австралии и на некоторых островах Тихого океана. Питоны живут преимущественно на земле, но при этом отлично плавают и лазают по деревьям. Как у самцов, так и у самок сохранились рудименты задних конечностей. Самцы используют эти рудименты (так называемые анальные шпоры) во время ухаживания за самками.

Питоны убивают свою жертву путем удушения. Впившись в добычу своими острыми зубами и удерживая ее на месте, питон обвивается вокруг туловища жертвы, сжимая ее все сильнее и сильнее при каждом ее выдохе. Питаются питоны главным образом млекопитающими и птицами. Случаи нападения на человека крайне редки.

Многие из животных, изображаемых на обложках книг издательства O'Reilly, находятся на грани исчезновения, и все они представляют ценность для нашей планеты. Чтобы узнать о том, каким может быть ваш личный вклад в их спасение, посетите сайт `animals.oreilly.com`.

Изображение на обложке — это гравюра XIX столетия, взятая из архива Dover Pictorial.



Предисловие

Язык программирования Python обладает рядом, казалось бы, противоречивых качеств: в нем сочетаются элегантность и прагматичность, простота и мощь. С одной стороны, это высокоуровневый язык, с другой — он позволяет оперировать байтами и битами. Его легко изучать новичкам и в то же время он эффективный инструмент в руках специалистов.

Эта книга предназначена как для программистов, уже знакомых с языком Python, так и для тех, кто только приступает к его изучению, но имеет опыт работы с другими языками программирования. Она представляет собой справочное руководство по языку Python, наиболее часто используемым компонентам его стандартной библиотеки и ряду наиболее популярных и полезных модулей и пакетов сторонних производителей. Книга охватывает широкий круг прикладных вопросов, в том числе сетевое и веб-программирование, обработку XML-документов, работу с базами данных и высокоскоростные вычисления. Внимание читателей акцентируется на кросс-платформенных возможностях Python, включая рассмотрение основных способов расширения Python и внедрения кода на языке Python в приложения, созданные с использованием других языков программирования.

Структура книги

Книга разделена на пять частей, имеющих следующую структуру.

Часть I. Начало работы с Python

Глава 1. Введение в Python

В этой главе приведены общие характеристики языка Python и его реализаций, а также даны рекомендации относительно получения справочной и другой информации, в том числе касающейся участия в работе сообщества Python. Глава завершается описанием способов загрузки и установки Python на компьютере.

Глава 2. Интерпретатор Python

Рассматривается такие темы, как интерпретатор Python, опции его командной строки, а также его использование для запуска программ на языке Python и работы в интерактивном режиме. Глава включает краткое описание текстовых редакторов, пригодных для редактирования исходного кода программ на языке Python, и вспомогательных программ, предназначенных для проверки исходного кода. Кроме того, описаны полноценные интегрированные среды разработки, в том числе IDLE, которая бесплатно поставляется вместе с Python, и рассмотрены способы запуска программ на языке Python из командной строки.

Часть II. Ядро и встроенные объекты Python

Глава 3. Язык программирования Python

Рассматриваются такие темы, как синтаксис языка Python, встроенные типы данных, выражения, команды, поток управления, а также определение и вызов функций.

Глава 4. Объектно-ориентированный Python

Рассматривается реализация принципов объектно-ориентированного программирования в языке Python.

Глава 5. Исключения

Рассматриваются такие темы, как использование исключений для обработки ошибок и особых ситуаций, а также способы протоколирования подобных событий.

Глава 6. Модули

В этой главе рассказывается о группировании кода Python в модули и пакеты, определении и импорте модулей, а также об установке пакетов Python сторонних производителей. Также обсуждается изоляция зависимостей проекта с помощью виртуальных сред.

Глава 7. Встроенные объекты и модули стандартной библиотеки

Обсуждаются встроенные типы данных и функции, а также некоторые из основных модулей стандартной библиотеки Python (набор модулей, предоставляющих функциональность, которая в других языках программирования встроена в сам язык).

Глава 8. Строки и байты

Рассматриваются возможности Python, касающиеся обработки строк, включая строки в кодировке Unicode, байтовые строки и строковые литералы.

Глава 9. Регулярные выражения

Обсуждаются средства поддержки регулярных выражений в Python.

Часть III. Библиотека Python и модули расширения

Глава 10. Работа с файлами и текстом

Рассматриваются модули стандартной библиотеки Python, предназначенные для обработки файлов и текста, а также платформозависимые расширения, предоставляющие широкие возможности форматирования при вводе-выводе текста. Кроме того, обсуждаются вопросы интернационализации и локализации приложений, а также специфическая задача определения интерактивных сеансов работы с командами в текстовом режиме средствами Python.

Глава 11. Базы данных и постоянное хранение

Рассматриваются механизмы сериализации и хранения данных, а также предоставляемые в Python интерфейсы доступа к базам данных DBM и реляционным базам данных (использующим язык запросов SQL) и, в частности, к удобной базе данных SQLite, поставляемой вместе со стандартной библиотекой Python.

Глава 12. Работа со значениями даты и времени

Рассматриваются способы обработки дат и времени с использованием как стандартной библиотеки Python, так и популярных расширений от сторонних производителей.

Глава 13. Управление процессом выполнения

В этой главе рассказывается о том, как эффективно управлять выполнением программ на Python, включая выполнение динамически генерируемого кода и сборку мусора. Также рассматриваются некоторые внутренние типы Python и специфические вопросы регистрации функций "очистки", освобождающих неиспользуемые ресурсы по завершении работы программы.

Глава 14. Потоки и процессы

Рассматриваются средства Python, обеспечивающие одновременное выполнение программ как за счет использования нескольких потоков, выполняющихся в одном процессе, так и нескольких процессов, выполняющихся на одном компьютере. Также обсуждаются способы доступа к окружению процесса и обращения к файлам посредством механизма их отображения в памяти.

Глава 15. Математические вычисления

Эта глава посвящена рассмотрению средств математической обработки данных, предоставляемых как стандартной библиотекой Python, так и сторонними расширениями. В частности, обсуждается использование дробных десятичных чисел с фиксированной точностью и рациональных дробей вместо используемых по умолчанию чисел с плавающей точкой. Кроме того, рассмотрены способы генерации и использования псевдослучайных и истинно случайных чисел и приемы ускоренной обработки числовых массивов и матриц.

Глава 16. Тестирование, отладка и оптимизация кода

Эта глава познакомит вас со средствами и подходами, позволяющими проверять корректность работы программ (т.е. убеждаться в том, что они делают именно то, для чего предназначены), находить и исправлять программные ошибки, контролировать быстродействие программ и повышать их производительность. Обсуждается смысл понятия "предупреждение" и рассматривается модуль библиотеки Python, предназначенный для работы с предупреждениями.

Часть IV. Сетевое и веб-программирование

Глава 17. Основы работы с сетями

Рассмотрены основы работы с сетевыми протоколами в Python.

Глава 18. Асинхронные архитектуры

В этой главе анализируются альтернативные варианты асинхронного (управляемого событиями) программирования и, в частности, современные архитектуры, основанные на сопрограммах, средства низкоуровневого мультиплексирования и соответствующие модули стандартной библиотеки Python.

Глава 19. Модули для работы с клиентскими сетевыми протоколами

Обсуждаются модули стандартной библиотеки Python, предназначенные для написания клиентских сетевых программ и, в частности, для работы с различными сетевыми протоколами на стороне клиента, ответственными за отправку и получение электронной почты и обработку URL-адресов.

Глава 20. Работа с протоколом HTTP

Обсуждаются способы создании веб-приложений с помощью популярных облегченных фрёймворков сторонних производителей на основе использования стандартного WSGI-интерфейса Python для доступа к веб-серверам.

Глава 21. Электронная почта, MIME и другие сетевые кодировки

Рассматриваются способы обработки в Python сообщений электронной почты и других документов, структурируемых и кодируемых для обмена по сети.

Глава 22. Структурированный текст: HTML

Рассматриваются популярные сторонние модули расширения Python, предназначенные для обработки, изменения и создания HTML-документов.

Глава 23. Структурированный текст: XML

Рассматриваются модули библиотеки Python и популярные расширения, предназначенные для обработки, изменения и создания XML-документов.

Часть V. Расширение, распространение, миграция v2/v3

Глава 24. Модули расширения и внедрение классического Python в другие программы

Описывается процесс создания модулей расширения Python с использованием C API и Cython.

Глава 25. Распространение расширений и программ

Рассматриваются популярные инструменты и модули, предназначенные для пакетирования и распространения расширений, модулей и приложений Python.

Глава 26. Переход с версии 2.x на версию 3.x и сосуществование различных версий

Обсуждается профессиональные методики написания кода для версий Python v2 и v3, а также наиболее эффективные способы переноса существующего кода из версии v2 в версию v3 и меры по обеспечению портируемости кода между версиями.

Соглашения, принятые в книге

На протяжении всей книги используются следующие соглашения.

Соглашения о ссылках

Там, где это уместно, каждый необязательный параметр функции или метода указывается вместе с его значением по умолчанию с использованием синтаксиса Python *имя=значение*. Встроенные функции не нуждаются в именованных параметрах, поэтому имена параметров для них несущественны. Для некоторых необязательных параметров важен сам факт их наличия или отсутствия, а не их значение по умолчанию. В подобных случаях необязательность параметра обозначается заключением его в квадратные скобки (`[]`). При наличии нескольких необязательных аргументов могут использоваться вложенные квадратные скобки.

Соглашения о версиях

Книга охватывает как версию Python 3.5, обозначаемую в тексте как "v3", так и версию Python 2.7, обозначаемую как "v2". В упоминаниях о новинках, появившихся в версии Python 3.6 (выпущенной буквально перед отправкой книги в печать), эта версия обозначается как "3.6". Выполняя примеры в версии v2, помещайте в начале каждого из них инструкцию `from __future__ import print_function` (врезка "Инструкции импорта `from __future__`" в главе 26).

Типографские соглашения

Курсив
: Применяется для выделения имен файлов, названий программ, URL-адресов и новых терминов.

Полужирный курсив
: Используется для выделения новинок, появившихся в версии Python 3.6.

`Моноширинный шрифт`
: Используется для представления кода примеров и всех элементов, которые могут появляться в коде, включая ключевые слова, а также имена методов, функций, классов и модулей.

`Моноширинный курсив`
: Представляет текст, вместо которого пользователь должен ввести фактические значения, используемые в коде примеров.

`Моноширинный полужирный`
: Используется для представления команд, вводимых в системной командной строке.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

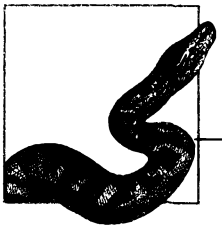
Наши электронные адреса:

E-mail: `info@dialektika.com`

WWW: `www.dialektika.com`

I

Начало работы с Python



1 Введение в Python

Универсальный язык программирования Python известен уже давно: его создатель, Гвидо ван Россум, приступил к разработке Python в 1990 году. Этот стабильный и зрелый язык обладает целым рядом привлекательных качеств: он сверхвысокоуровневый, динамичный, объектно-ориентированный и кросс-платформенный. Python выполняется на всех основных аппаратных платформах под управлением самых разных операционных систем, предоставляя полную свободу в отношении их выбора.

Python обеспечивает высокую производительность на всех фазах жизненного цикла разработки программного обеспечения (ПО): фазах анализа, проектирования, прототипирования, написания, тестирования, отладки и доводки программного кода, а также подготовки документации, развертывания и, разумеется, сопровождения готового приложения. Год от года популярность Python неуклонно возрастает. В наши дни знание языка Python служит дополнительным плюсом к характеристике любого программиста, поскольку Python проник практически во все возможные ниши и может играть важную роль в качестве компонента любого программного решения.

В Python удивительным образом сочетаются элегантность, простота, практичность и подлинная мощь. Вы очень быстро повысите свою продуктивность с помощью Python благодаря стройности и систематичности его архитектуры, богатой стандартной библиотеке и множеству доступных пакетов и инструментов, предложенных сторонними производителями. Python легко изучить, что облегчает жизнь новичкам, но при этом он обладает достаточно мощными возможностями, которые удовлетворят даже самых требовательных профессионалов.

Язык Python

Ни в коей мере не будучи минималистичным языком, Python в то же время отличается предельной экономичностью, и это вполне объяснимо. Если для выражения

какой-либо идеи в языке уже предусмотрен один способ, то добавление других, аналогичных ему, средств в лучшем случае принесет лишь скромные дополнительные преимущества, которые не окупятся сопутствующим этому усложнением языка. Чем сложнее язык, тем труднее его изучить и освоить (а также эффективно и безукоризненно реализовать). Усложнения и специфические особенности языка снижают темпы разработки программного обеспечения и затрудняют его сопровождение, особенно в случае крупных проектов, предполагающих участие многих разработчиков, которые вынуждены работать с кодом, первоначально написанным их коллегами.

Python — простой, но не упрощенный язык программирования. В соответствии с философией Python поведение языка не должно зависеть от контекста. В Python соблюдается ряд принципов, согласно которым язык не должен предусматривать удобные "обходные пути", частные случаи, специальные исключения, а также вводить излишне тонкие различия или привлекать изощренные методы скрытой оптимизации. Подобно любому другому тщательно продуманному продукту, хороший язык должен демонстрировать разумный баланс между этими общими принципами, с одной стороны, и вкусом, здравым смыслом и практичностью — с другой. Python — универсальный язык программирования: благодаря своим качествам он находит применение практически в любой области разработки ПО. Не существует такой сферы, в которой Python не мог бы стать частью удачного решения. Здесь слово "часть" очень важное. Многие разработчики обходятся одним только языком Python, но это вовсе не означает, что он не может сочетаться с другими языками: программы на языке Python могут совместно работать со множеством других программных компонентов, и поэтому он идеально подходит для сопряжения компонентов, написанных на разных языках. Одна из целей создания Python заключалась в том, чтобы он "хорошо уживался" с другими языками программирования.

Python — *сверхвысокоуровневый язык*. Это означает, что он использует более высокий уровень абстракции и дистанцирован от аппаратной платформы в большей степени, чем такие классические компилируемые языки, как C, C++ или Fortran, традиционно относящиеся к категории высокоуровневых. По сравнению с ними Python проще, быстрее обрабатывается (как человеком, так и программными средствами) и отличается более строгой систематичностью. Это способствует повышению производительности труда программистов и делает Python весьма привлекательным инструментом разработки. Хорошие компиляторы для классических языков могут генерировать двоичный код, который будет выполняться быстрее аналогичного кода, написанного на Python. Но в большинстве случаев производительность приложений, написанных на Python, достаточно высока. В тех случаях, когда это не так, можно улучшить быстродействие программы при сохранении всех преимуществ повышения продуктивности разработки, используя методы, описанные в разделе "Оптимизация" главы 16.

Некоторые из более поздних языков, такие как Java и C#, являются несколько более высокоуровневыми по сравнению с такими классическими языками, как C и Fortran, и имеют общие характеристики как с ними (например, необходимость использования объявлений), так и со сверхвысокоуровневыми языками наподобие

Python (например, компиляция в переносимый байт-код в типичных реализациях и сборка мусора, избавляющая программиста от необходимости брать на себя ответственность за управление памятью). Если вы считаете, что ваша продуктивность повышается, когда вы используете Java или C#, а не C или Fortran, то попробуйте еще больше повысить ее, обратившись к Python (возможно, в виде реализации Jython или IronPython, о которых пойдет речь в разделе "Реализации Python").

Если говорить о высокоуровневости, то Python сопоставим с такими мощными сверхвысокоуровневыми языками, как JavaScript, Ruby и Perl. Но во всем, что касается простоты и систематичности языка, Python не имеет себе равных.

Python — объектно-ориентированный язык программирования, но при этом позволяет разрабатывать код не только с использованием объектно-ориентированного и процедурного подходов, но и с включением некоторых элементов функционального программирования, допуская сочетание всех перечисленных стилей в соответствии с нуждами конкретного приложения. Объектно-ориентированные возможности Python концептуально аналогичны возможностям C++, но проще в использовании.

Стандартная библиотека Python и модули расширения

Понятие "Python" включает в себя не только язык Python: стандартная библиотека и другие модули расширения почти так же важны для эффективного использования Python, как и сам язык. Стандартная библиотека Python — это богатая коллекция тщательно спроектированных, стабильных, написанных исключительно на языке Python модулей, специально предназначенных для многократного использования. Входящие в нее модели позволяют решать такие задачи, как представление данных, обработка текста, взаимодействие с операционной и файловой системами и веб-программирование. Поскольку эти модули написаны на Python, они работают на всех поддерживаемых платформах.

Модули расширения, будь то из стандартной библиотеки или из других источников, обеспечивают коду на Python доступ к функциональности, предоставляемой базовой операционной системой и другими программными компонентами — графическими пользовательскими интерфейсами (GUI), базами данных и сетями. Кроме того, расширения обеспечивают достижение максимального быстродействия при выполнении таких задач, сопряженных с интенсивными вычислениями, как синтаксический анализ XML-файлов и обработка крупных числовых массивов. В то же время модули расширения, написанные на языках, отличных от Python, не всегда обеспечивают автоматическую кросс-платформенную совместимость того же уровня, что и код на чистом Python.

Вы сможете создавать специализированные модули расширения на низкоуровневых языках для достижения максимальной производительности в небольших, но требующих интенсивных вычислений приложениях на основе прототипов, первоначально написанных на языке Python. Кроме того, с помощью таких средств, как Cython и CFFI (Common Foreign Function Interface), модули расширения Python

можно использовать в качестве оберток для существующих библиотек C/C++, о чем будет говориться в разделе "Расширение Python без использования C API" главы 24. И наконец, последнее по счету, но не по значимости замечание: вы можете внедрять Python в приложения с кодом на других языках, открывая сценариям Python доступ к функциональности существующих приложений посредством модулей расширения Python, специфических для каждого приложения.

В этой книге описаны многие модули, происходящие как из стандартной библиотеки, так и из других источников и находящие применение в таких областях, как сетевое программирование, базы данных, работа с текстовыми и двоичными файлами и взаимодействие с операционной системой.

Реализации Python

В настоящее время имеются четыре промышленные реализации Python (CPython, Jython, IronPython, PyPy) и одна высокопроизводительная реализация, находящаяся на ранней стадии разработки, Pyston (`https://blogs.dropbox.com/tech/2014/04/introducing-pyston-an-upcoming-jit-based-python-implementation/`), которая далее не рассматривается. О целом ряде других, в большей степени экспериментальных, реализаций рассказано в одном из последующих разделов.

Эта книга посвящена главным образом CPython, наиболее широко используемой реализации, которая для простоты и будет обычно подразумеваться при ссылках на Python. Однако различие между языком и его реализацией является очень важным.

CPython

Классический Python (также известный как CPython и часто называемый просто Python) — это самая последняя стабильная и полностью промышленная реализация Python. Ее можно считать "эталонной реализацией" языка. CPython включает компилятор, интерпретатор и набор встроенных модулей и дополнительных расширений, которые все написаны на стандартном C. CPython можно использовать на любой платформе, обеспечивающей работу компилятора C в соответствии с требованиями стандарта ISO/IEC 9899:1990 (т.е. на любой из современных популярных платформ). О том, как загрузить и установить CPython, будет рассказано в разделе "Установка". За исключением некоторых разделов, в отношении которых в необходимых случаях делаются специальные оговорки, весь излагаемый в книге материал относится к реализации CPython. Она поддерживает как Python *v3* (версия 3.5 или выше), так и Python *v2* (версия 2.7).

Jython

Jython (`www.jython.org`) — это реализация Python для любой виртуальной машины Java (JVM), совместимой с версией Java 7 или выше. Такие JVM доступны для всех популярных современных платформ. На момент выхода книги Jython поддерживал

версию v2, но не v3. Работая с Jython, вы сможете использовать все библиотеки и фреймворки Java. Оптимальное использование Jython требует определенного знакомства с основными классами Java. Вам не придется писать код на Java, но документация и примеры для библиотек Java сформулированы в терминах Java, и, для того чтобы вы могли читать и понимать их, Java-классы не должны быть для вас чем-то новым. Вы также будете использовать вспомогательные инструменты Java для таких задач, как манипулирование *.jar*-файлами и подписывание аплетов. В этой книге мы будем работать с Python, а не с Java. Чтобы освоить Jython, вам следует обратиться к книге *The Definitive Guide to Jython (Open Source Version)*, которая доступна на сайте Jython.org (`http://www.jython.org/jythonbook/en/1.0/`).

IronPython

IronPython (`ironpython.net`) — это реализация Python (на момент выхода книги — лишь версии v2, но не v3) для разработанной компанией Microsoft общеязыковой исполняющей среды (CLR), более известной под названием .NET, которая в настоящее время отнесена к категории ПО с открытым исходным кодом (`https://github.com/dotnet/coreclr`) и портирована на платформы Linux и macOS. С IronPython вы сможете использовать все библиотеки и фреймворки CLR. Кроме собственной реализации Microsoft существует кросс-платформенная реализация CLR, известная под названием Mono (`www.mono-project.com`) и работающая, наряду с Windows, под управлением других операционных систем. Оптимальное использование IronPython требует знакомства с основными библиотеками CLR. Вам не придется писать код на C#, но документация и примеры для существующих библиотек нередко сформулированы в терминах C#, и, для того чтобы вы могли читать и понимать их, язык C# не должен быть для вас чем-то новым. Вы также будете использовать вспомогательные инструменты CLR для таких задач, как создание сборок CLR. В этой книге мы будем работать с Python, а не с CLR. Чтобы использовать IronPython, вам следует дополнить чтение книги оригинальной онлайн-документацией IronPython и, если в этом возникнет необходимость, воспользоваться другими доступными ресурсами, посвященными .NET, CLR, C#, Mono и т.п.

PyPy

PyPy (`pypy.org`) — это быстрая и гибкая реализация Python, написанная с использованием подмножества самого Python, которая способна транслировать код для ряда низкоуровневых языков и виртуальных машин с применением таких передовых методов, как *вывод типов*. Наибольшим достоинством PyPy является способность генерировать двоичный машинный код с помощью технологии JIT во время выполнения программы на Python. PyPy реализует версию v2, но на момент выхода книги готовилась к выпуску альфа-версия поддержки Python 3.5. PyPy обеспечивает значительные преимущества в отношении быстродействия и управления памятью, и выпуск его производственной версии не за горами.

Выбор между CPython, Jython, IronPython и PyPy

Если на вашей платформе, как и на большинстве других, может выполняться любая из реализаций CPython, Jython, IronPython и PyPy, то на какой из них вам стоит остановить свой выбор? Прежде всего, не выбирайте заранее ни одну из них: загрузите и установите все. Они могут безболезненно сосуществовать на одном компьютере, и все они бесплатны. Их установка будет стоить вам определенных затрат времени на загрузку реализаций и небольшого объема дискового пространства, но взамен вы получите возможность их непосредственного сравнения между собой.

Главное, что отличает одну реализацию от другой, — это окружение, в котором они могут выполняться, а также библиотеки и фреймворки, которые они могут использовать. Если вам нужна среда виртуальной машины, то наилучшим выбором будет Jython. Если вам необходима CLR (она же — .NET), воспользуйтесь преимуществами IronPython. Если вам требуется настраиваемая версия Python или же на первый план выходит высокая производительность для выполнения "долгоиграющих" программ, остановитесь на PyPy.

Если же вы работаете в традиционной среде, то вам отлично подойдет CPython. В отсутствие каких-либо веских причин для того, чтобы отдать предпочтение какому-то одному варианту, начните со стандартной эталонной реализации CPython, которая получает наиболее широкую поддержку в виде надстроек и расширений сторонних производителей и предоставляет наиболее предпочтительный вариант Python — версию v3 (точнее, версию 3.5 со всеми ее возможностями)[1].

Иными словами, для экспериментов, обучения или апробации идей используйте реализацию CPython как наиболее зрелую и получающую широкую поддержку. Если же речь идет о разработке и развертывании приложений, то наилучший выбор зависит от того, какие модули расширения вы собираетесь использовать и как вы хотите распространять свои программы. Приложения CPython часто работают быстрее приложений Jython или IronPython, особенно если вы используете такие модули расширения, как NumPy (обсуждается в разделе "Обработка массивов" главы 15). В то же время PyPy, благодаря JIT-компиляции в машинный код, зачастую обеспечивает еще большее быстродействие. Возможно, вам имело бы смысл сравнить быстродействие вашего кода в CPython и PyPy.

Наиболее зрелая реализация — CPython. Она существует дольше других, в то время как Jython, IronPython и PyPy являются более новыми реализациями и в меньшей степени проявили себя в деле. Похоже, что разработка CPython ведется более быстрыми темпами, чем разработка Jython, IronPython и PyPy. Например, на момент выхода книги обеспечивалась поддержка двух веток Python: v2 — поддержка версии 2.7 со

[1] Иногда мы будем обсуждать некоторые возможности версии Python 3.6, которая вышла буквально во время написания этой книги. Несмотря на то что она может быть еще недостаточно зрелой для того, чтобы вы могли использовать ее в производственных выпусках своих программ, знание того, что вам предложит Python 3.6, достигнув зрелости, не будет для вас лишним.

всеми вышеперечисленными реализациями; v3 — поддержка версий 3.5 и 3.6 доступна в CPython, а разработка поддержки версии 3.5 в PyPy прошла альфа-стадию.

Вместе с тем Jython может использовать в качестве модуля расширения любой класс Java, откуда бы он ни поступил: из стандартной библиотеки Java, библиотеки сторонних производителей или библиотеки, которую разработали вы сами. Точно так же IronPython может использовать любой класс CLR, независимо от его происхождения: из стандартной библиотеки CLR или в виде кода на языке C#, Visual Basic .NET или любом другом CLR-совместимом языке. Совместимость PyPy с большинством стандартных библиотек CPython, а также с написанными на языке C расширениями обеспечивается использованием Cython (раздел "Cython" в главе 24) и CFFI (раздел "Расширение Python без использования C API" в главе 24).

Приложение, созданное с помощью Jython, является полноценным Java-приложением со всеми свойственными Java преимуществами и недостатками развертывания и выполняется на любой целевой машине с подходящей JVM. Возможности работы с пакетами аналогичны возможностям Java. Точно так же приложение, созданное с помощью IronPython, полностью совместимо со спецификациями .NET.

Jython, IronPython, PyPy и CPython все являются надежными реализациями Python, близкими между собой в отношении удобства использования и производительности. Поскольку работа с каждой из JVM- или CLR-платформ связана с определенными издержками, и в то же время они предоставляют множество полезных библиотек, фреймворков и инструментов, любая из реализаций может предложить решающие практические преимущества в конкретных сценариях развертывания. Имеет смысл ознакомиться со слабыми и сильными сторонами каждой из них, а затем выбрать оптимальный вариант в зависимости от обстоятельств.

Другие разработки, реализации и дистрибутивы

Популярность Python стала причиной появления многочисленных групп и отдельных лиц, проявивших интерес к его развитию и предоставивших возможности и реализации, находящиеся вне фокуса внимания основной команды разработчиков.

Для программистов, впервые столкнувшихся с Python, его установка нередко доставляла определенные хлопоты. В настоящее время почти все Unix-системы включают реализацию версии v2, а последние выпуски Ubuntu включают v3 в качестве "системного Python". Если вы намерены всерьез заниматься разработкой программ на Python, то первое, что вы должны сделать, — это *не пытаться конфигурировать версию Python, установленную вашей операционной системой*! Независимо от любых других факторов, Python все в большей степени используется самой операционной системой, поэтому вмешательство в его установку чревато неприятностями.

В связи с этим подумайте о независимой установке одной или нескольких реализаций Python, которые вы сможете свободно использовать в своих разработках, будучи полностью уверенным в том, что никакие ваши действия не смогут нанести вред операционной системе. Мы также рекомендуем изолировать проекты один

от другого за счет *виртуальных окружений* (раздел "Окружения Python" в главе 6), тем самым позволяя им использовать зависимости, которые в противном случае могли бы привести к возникновению конфликтов (например, если двум проектам требуются различные версии одного и того же модуля).

Популярность Python привела к появлению многочисленных активных сообществ, и в экосистеме языка наблюдается высокая активность. В последующих разделах кратко охарактеризованы наиболее интересные из последних разработок, и отсутствие описания какого-либо проекта является исключительно следствием нехватки времени и места, а не нашим отрицательным отношением к этому проекту.

Pyjion

Pyjion (`https://github.com/Microsoft/Pyjion`) — это проект компании Microsoft с открытым исходным кодом, главной целью которого является добавление в CPython программного интерфейса (API), предназначенного для управления JIT-компиляторами. К числу других целей относится создание JIT-компилятора для ядра CLR-среды Microsoft, которая, несмотря на то что является одним из центральных элементов среды .NET, теперь отнесена к категории ПО с открытым исходным кодом, и фреймворка для разработки JIT-компиляторов. В связи с этим данный проект не является реализацией Python, но мы все же упоминаем о нем, поскольку он открывает возможность транслировать байт-код CPython в высокоэффективный код для множества различных сред. Интеграция Pyjion в CPython будет возможна в соответствии с рекомендацией **PEP 523**[2] (`https://www.python.org/dev/peps/pep-0523/`), реализованной в версии Python 3.6.

IPython

IPython улучшает стандартный CPython, делая его более мощным и удобным для интерактивного использования. IPython расширяет возможности интерпретатора, допуская сокращенный синтаксис вызова функций и расширяемую функциональность, вводимую символом процента (`%`) и известную под названием *магические команды* (magics). Он также предоставляет возможность временного выхода в командную оболочку операционной системы (shell escapes) и получения результатов выполняемых в ней команд. Вы сможете использовать вопросительный знак для получения информации о любом объекте из документации (два вопросительных знака означают использование расширенной документации); все стандартные возможности Python также остаются доступными.

IPython получил признание в научных кругах и среди специалистов по обработке данных. Постепенно видоизменяясь (через разработку инструмента IPython Notebook, который впоследствии был подвергнут рефакторингу и переименован в Jupyter Notebook), он превратился в интерактивную среду программирования,

[2] PEP (Python Enhancement Proposal) — предложение по развитию Python. Процесс PEP (`https://www.python.org/dev/peps/`) является основным механизмом отбора проектных решений для включения в Python. — *Примеч. ред.*

позволяющую работать с отдельными фрагментами кода, перемежая их комментариями в стиле *грамотного программирования* (https://ru.wikipedia.org/wiki/Грамотное_программирование), и выводить результаты работы кода в графическом виде с помощью таких подсистем, как matplotlib и bokeh. Пример графического вывода, созданного с помощью matplotlib, представлен на рис. 1.1. Получая щедрое финансирование на протяжении нескольких последних лет, проект Jupyter/IPython может служить примером одной из наиболее ярких историй успеха Python.

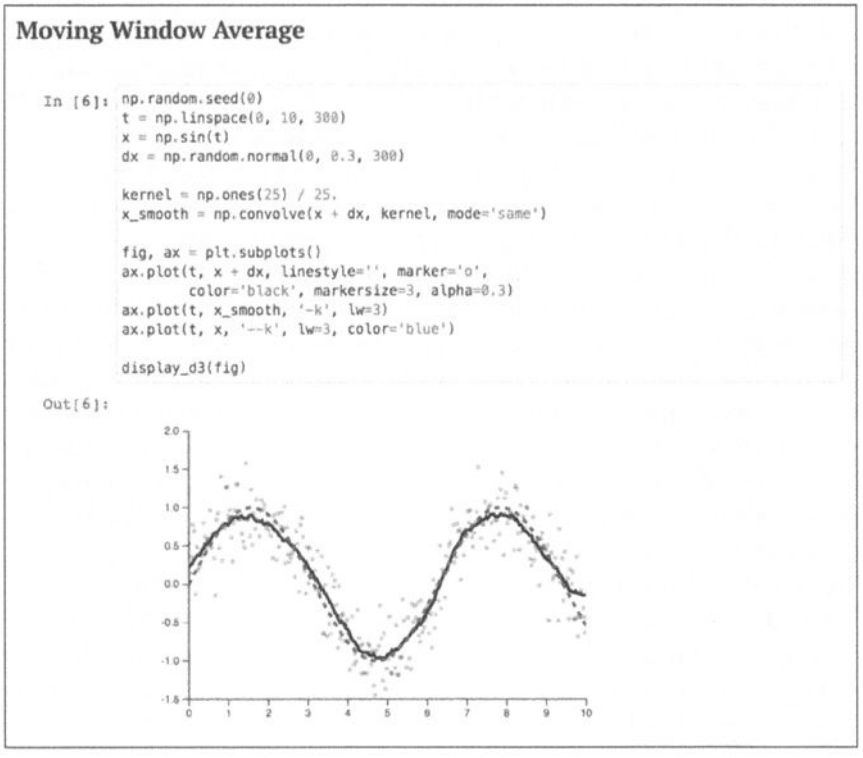


Рис. 1.1

MicroPython: а ваш ребенок уже стал программистом?

Всеобщая тенденция к миниатюризации не обошла стороной и Python, превратив его в своеобразное хобби. Компьютеры, целиком размещающиеся на миниатюрной плате, наподобие *Raspberry Pi* (www.raspberrypi.org) и *BeagleBoard* (beagleboard.org), предоставляют возможность выполнять Python в полноценной среде Linux. Ярус, располагающийся ниже этого уровня, охватывает интересный класс устройств — *микроконтроллеры*, которые представляют собой программируемые чипы с конфигурируемыми устройствами. Это расширяет сферу применения как любительских, так и профессиональных проектов, например, за счет того, что упрощается создание аналоговых и цифровых датчиков, которые приложения смогут

использовать для измерения уровня освещенности и температуры при минимальном дополнительном оборудовании.

На момент выхода книги типичными примерами таких микроконтроллеров могли служить *Adafruit Feather* (`https://www.adafruit.com/feather`) и *WiPy* (`https://www.pycom.io/product/wipy-2-0/`), но постоянно появляются (и исчезают) все новые устройства. Благодаря проекту MicroPython эти устройства, часть из которых предлагает улучшенную функциональность наподобие Wi-Fi, теперь можно программировать с помощью Python.

MicroPython (`micropython.org`) — это реализация Python 3, способная создавать байт-код или исполняемый машинный код (хотя последняя возможность может вообще не понадобиться многим пользователям). Она реализует синтаксис версии v3, но не включает большую часть стандартной библиотеки. Специальные модули обеспечивают возможность конфигурирования и управления работой различных элементов встроенного оборудования, а доступ к библиотеке Python для работы с сокетами позволяет этим устройствам выступать в качестве клиентов веб-служб. Код может запускаться событиями таймера и внешних устройств.

Обычно доступ к интерпретатору в таких устройствах осуществляется через последовательный USB-порт или посредством протокола Telnet. (Случаи реализации SSH[3] нам пока что неизвестны, поэтому вы должны позаботиться о защите этих устройств: *прямой доступ к ним через Интернет без принятия надлежащих мер безопасности должен быть исключен!*). Вы сможете запрограммировать последовательность команд процесса начальной загрузки при включении питания устройства с помощью Python, создав файл *boot.py* в памяти устройства, и этот файл может выполнять код любой степени сложности на языке Python.

Благодаря MicroPython язык Python может стать полноценной платформой Интернета вещей[4]. На момент выхода книги устройства *BBC micro:bit*, способные выполнять программы на языке Python, раздавались в Великобритании всем 7-летним школьникам (всего около миллиона устройств). А вы уверены в том, что ваш ребенок еще не знаком с языком Python?

Anaconda и Miniconda

Одним из наиболее успешных дистрибутивов Python за последние годы является Anaconda от компании Continuum Analytics (`https://www.continuum.io/anaconda-overview`). Помимо стандартной библиотеки дистрибутив включает огромное количество предварительно сконфигурированных и протестированных модулей с открытым

[3] SSH (Secure Shell — безопасная оболочка) — сетевой протокол, обеспечивающий безопасную передачу данных в незащищенной среде практически любого другого сетевого протокола. — *Примеч. ред.*

[4] Интернет вещей (Internet of Things, IoT) — концепция, в соответствии с которой различные бытовые и промышленные приборы, а также "вещи" (например, лекарственные препараты), снабженные радиочастотными метками, могут взаимодействовать между собой посредством коммуникационных сетей (`https://ru.wikipedia.org/wiki/Интернет вещей`). — *Примеч. ред.*

исходным кодом. Возможно, вы обнаружите, что он содержит все необходимые зависимости, которых вам будет достаточно для работы во многих случаях. В UNIX-подобных системах он устанавливается в один каталог: для активации программы достаточно добавить каталог *bin* Anaconda в переменную `PATH` командной оболочки.

Anaconda — дистрибутив Python, основанный на технологии управления пакетами под названием *conda*. Родственная реализация, Miniconda, предоставляет доступ к тем же библиотекам, но не загруженным предварительно, что делает ее более удобной для создания настраиваемых окружений. Conda не использует стандартные виртуальные окружения, но содержит эквивалентные средства, позволяющие изолировать проекты Python друг от друга и использовать в каждом из них собственные зависимости и версию Python.

Nuitka: конвертируйте код Python в C++

Nuitka — альтернатива интерпретатору Python, позволяющая транслировать код Python v2 и v3 в код C++ с использованием удобных средств оптимизации кода, в настоящее время обеспечивающих повышение быстродействия программ более чем в два раза по сравнению с CPython. В дальнейшем предполагается усовершенствование возможностей автоматического определения типов и еще большее увеличение быстродействия программ. Для получения более подробной информации посетите сайт Nuitka (`nuitka.net`).

Grumpy: транскомпилятор "Python в Go"

Grumpy — экспериментальный компилятор, преобразующий код на языке Python в код на языке Go, но поддерживающий только версию v2. После этого код Go компилируется в исполняемый двоичный код, что обеспечивает возможность импорта модулей Go (но не обычных расширений Python) и высокую масштабируемость кода. Grumpy оптимизирован для обслуживания крупных параллельных нагрузок со множеством легковесных потоков и, как сообщалось в блоге Google Open Source (`https://opensource.googleblog.com/2017/01/grumpy-go-running-python.html`), в один прекрасный день может заменить CPython в качестве среды выполнения, используемой для обслуживания миллионов запросов в секунду в сети YouTube. Grumpy поддерживается на GitHub (`https://github.com/google/grumpy`).

Transcrypt: конвертируйте код Python в JavaScript

Попытки поиска возможностей использования Python в роли браузерного языка предпринимались неоднократно, но JavaScript прочно удерживает свои позиции. Transcrypt (`www.transcrypt.org`) — это пакет Python, который устанавливается с помощью утилиты командной строки `pip` и преобразует код Python в код JavaScript, выполняющийся в браузере. При этом вы получаете полный доступ к объектной модели документа (DOM) в браузере, что позволяет вашему коду динамически манипулировать содержимым окна и использовать библиотеки JavaScript.

Несмотря на то что система Transcrypt создает минифицированный код, она предоставляет полные карты кода, что обеспечивает возможность отладки программ со ссылками на исходный код Python, а не на сгенерированный JavaScript-код. Вы сможете писать обработчики событий браузера на языке Python и свободно включать их в HTML- и JavaScript-код. Возможно, Python никогда не станет первоклассным браузерным языком, но наличие системы Transcrypt означает, что теперь никакой потребности в этом нет.

Существует еще один (более зрелый, но очень активно развивающийся) проект, нацеленный на создание сценариев для веб-страниц с помощью Python: Brython (`brython.info`), не говоря уже о нескольких других аналогичных проектах. Если вам необходимо принять решение относительно выбора одного из них, то имеет смысл предварительно ознакомиться с обсуждением этого вопроса на сайте Stack Overflow (`http://stackoverflow.com/questions/30155551/python-in-browser-how-to-choose-between-brython-pypy-js-skulpt-and-transcrypt`).

Вопросы лицензирования и оплаты

На CPython распространяется действие лицензии Python Software Foundation License (PSFL) Version 2, совместимой с GNU Public License (GPL), но позволяющей, подобно лицензиям BSD/Apache/MIT, использовать Python для разработки любого проприетарного, свободного или с открытым исходным кодом программного обеспечения. Лицензии Jython, IronPython и PyPy столь же либеральны. Все программные продукты, предоставляемые для загрузки на основных сайтах Python, Jython, IronPython и PyPy, являются бесплатными. Эти лицензии никоим образом не ограничивают условия лицензирования и оплаты, устанавливаемые в отношении программного обеспечения, разрабатываемого с использованием охватываемых ими инструментов, библиотек и документации.

Но не все программные продукты, имеющие отношение к Python, не требуют лицензионных расходов. Во многих случаях разработанный третьими сторонами исходный код на языке Python, а также инструменты и модули расширения, которые вы можете свободно загружать, имеют либеральные лицензии, аналогичные тем, которые имеет сам Python. Однако на некоторые продукты распространяется действие лицензий GPL или Lesser GPL (LGPL), ограничивающих условия лицензирования, которые вы вправе устанавливать для разработанных на их основе продуктов. Некоторые коммерческие модули и инструменты могут требовать оплаты с вашей стороны, либо безусловной, либо при условии, что вы собираетесь использовать их для получения прибыли.

Нет ничего лучше скрупулезного изучения условий лицензирования и оплаты. Прежде чем вкладывать время и силы в разработку любого программного инструмента или компонента, убедитесь в приемлемости для вас условий лицензирования. Часто, особенно в корпоративной среде, для выяснения этих вопросов может потребоваться консультация юриста. Если явно не оговорено иное, модули и инструменты,

обсуждаемые в этой книге, на момент ее написания могли рассматриваться как бесплатные программные продукты с открытым исходным кодом, на которые, как и на Python, распространяется действие либеральной лицензии. Однако соответствующая юридическая экспертиза нами не проводилась, а условия лицензирования с течением времени могут меняться, поэтому лишняя проверка никогда не помешает.

Разработка и версии Python

Разработкой, сопровождением и выпуском очередных версий языка Python занимается основная группа разработчиков под руководством Гвидо ван Россума, изобретателя, архитектора и "великодушного пожизненного диктатора Python" (Benevolent Dictator For Life, BDFL). Этот титул означает, что при принятии решений относительно включения новых элементов в язык Python и его стандартную библиотеку последнее слово остается за Гвидо. Авторские права на Python как объект интеллектуальной собственности принадлежат фонду Python Software Foundation (PSF) — некоммерческой организации, деятельность которой направлена на развитие языка и сообщества Python (раздел "Python Software Foundation"). Многие номинированные и рядовые члены PSF направляют свои предложения в базовый репозиторий исходного кода Python. В настоящее время в качестве такового используется распределенная система управления версиями Mercurial, но наблюдается постепенный переход к Git (`https://github.com/python`) в соответствии с рекомендациями, изложенными в руководстве Python Developer's Guide (`https://docs.python.org/devguide/`). Большинство коммиттеров Python являются членами или сотрудниками PSF.

Предложенные изменения подробно описываются в документах PEP (Python Enhancement Proposals), обсуждаются (иногда с привлечением консультантов) разработчиками Python и более широким сообществом пользователей Python, а затем окончательно утверждаются или отвергаются Гвидо или его уполномоченными представителями, которые принимают во внимание результаты обсуждения и голосования, но не обязаны строго следовать им. Сотни людей вносят свой вклад в разработку Python посредством подачи документов PEP, обсуждения, а также предоставления отчетов об ошибках и "заплат" к исходному коду, библиотекам и документации Python.

В настоящее время основная команда разработчиков выпускает младшие версии Python (3.*x* с нарастающими значениями *x*) с периодичностью примерно каждые 18 месяцев. Версия Python 2.7, первый выпуск которой пришелся на июль 2010 года, была последним выпуском линейки 2.*x*. Имеющиеся и последующие выпуски v2 включают и будут включать лишь изменения, связанные с устранением обнаруженных ошибок, а не добавлением новых возможностей, и изменяться в них будет лишь номер сборки. На момент выхода книги текущей сборкой была 2.7.13. Версия 2.8 уже никогда не будет выпущена, поэтому мы настоятельно рекомендуем переходить на версию v3.

Каждый младший выпуск v3 (в отличие от очередных сборок) добавляет новую функциональность, которая расширяет сферу возможностей Python и упрощает

его использование, одновременно обеспечивая обратную совместимость. Версия Python 3.0, нарушившая принцип сохранения обратной совместимости с целью исключения "унаследованных" возможностей, которые стали ненужными, и упрощения языка, была выпущена в декабре 2008 года. Версия Python 3.5 (обозначаемая нами как "v3") была выпущена в сентябре 2015 года, версия 3.6 — в декабре 2016 года, как раз тогда, когда написание этой книги близилось к концу.

Очередные версии выпускаются поэтапно. Каждая младшая версия 3.*x* сначала выпускается в виде альфа-версии, обозначаемой как 3.*x* a0, 3.*x* a1 и т.д. Вслед за альфа-версией выпускается по крайней мере одна бета-версия, 3.*x* b1, а вслед за бета-версиями — по крайней мере одна версия-кандидат (release candidate), 3.*x* rc1. К моменту выхода окончательной версии 3.*x* (3.*x*.0) она представляет собой вполне стабильный и надежный продукт, прошедший тестирование на всех основных платформах. Любой программист может принять участие в этом процессе, загружая версии Python, предоставляемые в конце каждого этапа разработки, испытывая их и направляя отчеты о всех обнаруженных проблемах.

Сразу после выпуска младшей версии внимание основной команды разработчиков частично переключается на следующую младшую версию. Однако обычно текущая младшая версия продолжает выходить в виде последующих сборок, которые нумеруются с помощью чисел, указываемых через точку вслед за младшим номером версии (т.е. 3.*x*.1, 3.*x*.2 и т.д.). Сборки не вносят никакой новой функциональности, но исправляют обнаруженные ошибки, портируют Python на новые платформы, улучшают документацию, оптимизируют код и добавляют инструментальные средства.

В этой книге рассматриваются версии Python 2.7 и 3.5 (и их последующие сборки) — наиболее стабильные и распространенные на момент ее написания (однако во врезках будет приводиться информация о новинках, появившихся в самой последней версии — Python 3.6). Версия Python 2.7 обозначается в книге как *v2*, а версия Python 3.5 (и все последующие выпуски) — как *v3*. В необходимых случаях мы указываем, какие части языка и библиотек доступны только в версии v3 и не могут использоваться в версии v2, а какие части v3 можно использовать также в v2, например с помощью синтаксиса `from __future__ import` (раздел "Инструкции импорта `from __future__`" в главе 26). Если говорится о возможностях, доступных в версии v3, то при этом подразумевается версия Python 3.5 (а также, предположительно, все будущие версии, включая 3.6, но не обязательно, скажем, 3.4). Если же речь идет о возможностях, доступных в v2, то это относится к версии Python 2.7.*x* (в настоящее время 2.7.13), но не обязательно к версии 2.6 или более ранним версиям. В отдельных случаях, касающихся включения в язык новых элементов, мы указывает также номер сборки, например 3.5.3.

Материал книги не ориентирован на устаревшие версии Python, такие как 2.5 или 2.6; эти версии существуют уже более пяти лет, и их не следует использовать в разработке. Однако они могут давать о себе знать, если внедрены в приложения, для которых вам приходится писать свои сценарии. К счастью, возможности Python в отношении обратной совместимости в пределах старших версий довольно неплохие:

версия v2 способна адекватно обрабатывать программы, написанные для Python 1.5.2 и более поздних версий. В то же время, как уже отмечалось, Python 3 создал брешь в обратной совместимости, хотя при соблюдении необходимых мер предосторожности можно написать код, который будет работать в обеих версиях, v2 и v3. Более подробную информацию о миграции устаревшего кода в версию v3 вы найдете в главе 26. Кроме того, код и документация всех прежних выпусков Python доступны в Интернете (`https://www.python.org/doc/versions/`).

Где искать информацию о Python

Богатейшим источником информации о Python является Интернет. В качестве отправной точки лучше всего использовать главную страницу сайта Python (`www.python.org`), где вы найдете массу ссылок, которые вас могут заинтересовать. Если вас интересует Jython, то вы в обязательном порядке должны посетить сайт `www.jython.org`. Сайт IronPython находится по адресу `ironpython.net`, а сайт PyPy — по адресу `pypy.org`.

Документация

Python, Jython, IronPython и PyPy снабжены неплохой документацией. Руководства доступны во многих форматах, их можно просматривать и выводить на печать, и в них предусмотрены средства поиска. Руководства CPython можно читать в онлайн-режиме (мы часто ссылаемся на них как на "онлайн-документацию"): документация доступна как для v2 (`https://docs.python.org/2.7/`), так и для v3 (`https://docs.python.org/3/`). Кроме того, загружаемые файлы с документацией предлагаются в различных форматах как для v2 (`https://docs.python.org/2.7/download.html`), так и для v3 (`https://docs.python.org/3.5/download.html`). На странице документации Python (`https://www.python.org/doc/`) вы найдете ссылки для доступа к огромному количеству всевозможных документов. На странице документации Jython (`http://www.jython.org/docs/`) приведены ссылки как на документы, специфические для Jython, так и на документы общего характера, касающиеся Python. Также существуют страницы документации для IronPython (`http://ironpython.net/documentation/`) и PyPy (`http://doc.pypy.org/en/latest/`). Вам также будет полезна онлайн-документация категории FAQ (часто задаваемые вопросы), существующая для Python (`https://docs.python.org/2/faq/`), Jython (`https://wiki.python.org/jython/JythonFaq`), IronPython (`http://ironpython.codeplex.com/wikipage?title=FAQ`) и PyPy (`http://doc.pypy.org/en/latest/faq.html`).

Документация Python для начинающих программистов

Большинство документов и книг по Python (в том числе и эта) рассчитано на читателей, уже имеющих определенный уровень подготовки в области разработки программного обеспечения. Однако Python — довольно удобный язык для начинающих, поэтому из данного правила существуют исключения. Читателям, не

являющимся опытными программистами, можно порекомендовать начать свое знакомство с Python, используя следующие онлайновые источники.

- *Non-Programmers Tutorial For Python* (Josh Cogliati). Существует в двух версиях: для v2 (`http://oopweb.com/Python/Documents/easytut/VolumeFrames.html`) и v3 (`https://en.wikibooks.org/wiki/Non-Programmer%27s_Tutorial_for_Python_3`).
- *Learning to Program* (Alan Gauld). Охватывает обе версии, v2 и v3 (`alan-g.me.uk`)
- *Think Python, 2nd Edition* (Allen Downey). В основном относится к версии v3, но содержит дополнительные примечания, касающиеся версии v2 (`http://greenteapress.com/wp/think-python-2e/`).

Великолепным ресурсом для изучения Python (ориентирован как на начинающих, так и на не очень опытных программистов) является онлайновое Wiki-руководство "Beginners' Guide" (`https://wiki.python.org/moin/BeginnersGuide`), включающее множество ссылок и рекомендаций. Это руководство курируется сообществом, поэтому можно рассчитывать на то, что по мере появления и улучшения имеющихся книг, курсов, инструментов и т.п. оно будет постоянно обновляться.

Модули расширения и исходный код Python

Неплохой отправной точкой для получения расширений и исходного кода Python является каталог пакетов PyPI (от англ. **P***ython* **P***ackage* **I***ndex*), который на момент выхода книги включал свыше 95000 пакетов[5], снабженных описаниями и указателями (`https://pypi.python.org/pypi`).

Дистрибутив стандартного Python содержит исходный код на языке Python в стандартной библиотеке и каталогах *Demos* и *Tools*, а также исходный код на языке C для многих встроенных модулей расширения. Даже если вас не интересует сборка Python из исходного кода, мы рекомендуем загрузить и распаковать дистрибутивный исходный код Python для его изучения или, если вас это больше устраивает, просмотра в онлайновом режиме (`https://github.com/python/cpython/tree/master/Lib`).

Для многих модулей и инструментов Python, рассматриваемых в этой книге, существуют выделенные сайты, ссылки на которые приведены в соответствующих главах.

Книги

Несмотря на то что Интернет является богатейшим источником информации, книги по-прежнему занимают свою нишу (если бы это было не так, мы не писали бы данную книгу, а вы не читали бы сейчас ее). Языку Python посвящено множество

[5] Вполне вероятно, что к тому моменту, когда вы будете читать эти строки, он будет насчитывать свыше 100000 пакетов.

книг. Список книг, рекомендуемых нами к прочтению, приводится ниже, хотя некоторые из них относятся не к текущим, а к прежним версиям Python.

- Если вы только учитесь программировать, то книга *Pyton для чайников, 2-е издание* (John Paul Mueller), может послужить отличным введением в программирование в целом и в элементарный Python в частности.
- Если вы уже знакомы с программированием, но только приступаете к изучению Python, то вам подойдет книга *Head First Python, 2nd Edition* (Paul Barry). Как и во всех книгах серии *Head First* (`http://shop.oreilly.com/category/series/head-first/formula.do`), предметное изложение сопровождается графикой и юмором.
- Книги *Dive Into Python* и *Dive Into Python 3* (Mark Pilgrim) отлично подойдут тем, кто уже имеет достаточно большой опыт программирования на других языках, но хотел бы быстро изучить Python на конкретных примерах. Эти книги также доступны для бесплатной загрузки в нескольких форматах как для версии v2 (`www.diveintopython.net`), так и для версии v3 (`www.diveintopython3.net`).
- Книга *Beginning Python: From Novice to Professional* (Magnus Lie Hetland) наряду с подробным изложением основ языка Python содержит полное описание разработки 10 проектов, которые завершаются созданием готовых приложений, относящихся к различным областям применения.
- Книга *Python Essential Reference* (David Beazley) представляет собой полный справочник по языку Python и его стандартным библиотекам.
- В качестве краткого справочного руководства можно использовать книгу *Python. Карманный справочник, 5-е издание* (Марк Лутц).

Сообщество Python

Одной из сильных сторон Python является наличие стабильного, дружественного и благосклонного сообщества. Программисты и разработчики Python общаются друг с другом на конференциях, хакатонах (часто именуемых *спринтами* в сообществе Python) и в локальных пользовательских группах, активно обсуждают общие интересы и оказывают друг другу помощь в списках рассылки и социальных сетях.

Python Software Foundation

Деятельность фонда Python Software Foundation (PSF), обладателя прав на Python как объект интеллектуальной собственности, в значительной мере направлена на "поддержку разветвленного международного сообщества программистов на языке Python и содействие его развитию". PSF спонсирует пользовательские группы, конференции и спринты, предоставляет гранты на разработку проектов, программы поддержки и образование. Фонд PSF включает десятки номинированных членов

(получивших это право за весомый вклад в развитие Python; в их число входит вся команда разработчиков ядра Python и авторы этой книги), сотни членов, вложивших в развитие Python время, труд и деньги, и десятки корпоративных спонсоров. В соответствии с новой *моделью открытого членства* любой, кто использует и поддерживает Python, может стать членом PSF. Для получения справки относительно порядка предоставления членства в PSF посетите сайт `https://www.python.org/psf/membership/`. О том, каким может быть ваш вклад в развитие Python, если у вас есть такие намерения, можно прочитать в руководстве *Python Developer's Guide* (`https://docs.python.org/devguide/`).

Конференции по Python

Конференции по Python проходят по всему миру. К числу мероприятий с широкой тематикой относятся такие международные и региональные конференции, как PyCon (`https://us.pycon.org/2016/`) и EuroPython (`https://ep2015.europython.eu/en/`), а также конференции преимущественно локального характера, такие как PyOhio (`www.pyohio.org`) и Pycon Italia (`https://www.pycon.it/en/`). К числу узкотематических конференций относятся SciPy (`conference.scipy.org`), PyData (`https://pydata.org/events.html`) и DjangoCon (`https://2016.djangocon.us/`). За конференциями часто следуют спринты по кодированию, когда контрибуторы Python в течение нескольких дней совместно работают над каким-либо конкретным проектом с открытым исходным кодом, что попутно открывает для них дополнительные возможности для установления более близкого личного знакомства. Со списком проводимых конференций можно ознакомиться на сайте Community Workshops (`https://www.python.org/community/workshops/`). Видеоматериалы предыдущих конференций можно найти на сайте Pyvideo (`pyvideo.org`).

Пользовательские группы

На всех континентах, за исключением Антарктиды, созданы локальные группы пользователей, со списком которых можно ознакомиться на сайте `https://wiki.python.org/moin/LocalUserGroups`. Кроме того, более чем в 70 странах проводятся митапы (клубные встречи в неформальной обстановке) сторонников Python. Женщины, работающие с Python, создали собственную международную группу PyLadies (`www.pyladies.com`), имеющую локальные отделения.

Специальные группы

Обсуждение некоторых специализированных тем, связанных с Python, проводится посредством списков рассылки, организуемых специальными группами по интересам (Special Interest Groups, SIGS). Их список приведен на странице SIGS (`https://www.python.org/community/sigs/`) вместе с указателями на соответствующую информацию общего и узкоспециального характера. На момент выхода книги активными были более десяти SIG. Вот некоторые из них:

- *cplusplus-sig* — связывание C++ и Python (https://www.python.org/community/sigs/current/cplusplus-sig/);
- *mobile-sig* — Python на мобильных устройствах (https://www.python.org/community/sigs/current/mobile-sig/);
- *edu-sig* — Python в образовании (https://www.python.org/community/sigs/current/edu-sig/).

Социальные сети

Если вас интересует RSS-лента блогов, связанных с Python, посетите сайт Planetpython (planetpython.org) или следите за хеш-тегом @planetpython в Твиттере, где также стоит отслеживать хеш-тег @ThePSF. IRC-сеть FreeNode предоставляет несколько каналов, связанных с Python, из которых основным является *#python*. На Facebook можно присоединиться к группе Python Software Developers (https://www.facebook.com/groups/2249498416/). В Google+ существует неофициальное сообщество Python (https://plus.google.com/communities/103393744324769547228). Дополнительную информацию вы сможете получить в LinkedIn, открыв страницу Python (https://www.linkedin.com/topic/python). С техническими вопросами, касающимися Python, и ответами на них можно ознакомиться на сайте stackoverflow.com под различными тегами, включая *python* (http://stackoverflow.com/questions/tagged/python), *python-2.7* (http://stackoverflow.com/questions/tagged/python-2.7), *python-3.x* (http://stackoverflow.com/questions/tagged/python-3.x) и др. Тема Python активно поддерживается на сайте Stack Overflow, где можно найти много полезных обсуждений.

Списки рассылки и группы новостей

На странице https://www.python.org/community/lists/ приведены ссылки на списки рассылки и новостные группы, связанные с Python. В Usenet такой группой является *comp.lang.python* (https://groups.google.com/forum/#!forum/comp.lang.python). Эта новостная группа также доступна в виде списка рассылки, на который можно подписаться, заполнив соответствующую форму (https://mail.python.org/mailman/listinfo/python-list). Связанные с Python официальные объявления публикуются на доске объявлений по адресу https://groups.google.com/forum/#!forum/comp.lang.python.announce или в эквивалентном списке рассылки по адресу https://mail.python.org/mailman/listinfo/python-announce-list. Чтобы получить индивидуальную помощь, касающуюся решения конкретной проблемы, отправьте письмо по адресу help@python.org. По вопросам изучения или обучения Python обратитесь по адресу tutor@python.org или присоединитесь к списку рассылки по адресу https://mail.python.org/mailman/listinfo/tutor. Если у вас возникли какие-либо проблемы в связи с портированием Python между версиями v2 и v3 или вы хотите обсудить эту тему, присоединитесь к списку рассылки по адресу https://mail.python.org/mailman/listinfo/python-porting.

Для получения еженедельных сводок новостей и статей, связанных с Python, подпишитесь на еженедельник *Python Weekly* (`www.pythonweekly.com`).

В 2015 году в целях поддержки своей новой модели открытого членства фонд Python Software Foundation создал ориентированный на сообщество список рассылки, на который можно подписаться (`https://mail.python.org/mailman/listinfo/psf-community`).

Установка

Классическую версию Python (CPython), а также версии JVM (Jython), .NET (IronPython) и PyPy можно установить на большинстве платформ. Используя подходящую среду разработки (C для CPython, Java для Jython, .NET для IronPython; в случае PyPy, которая сама написана на языке Python, необходимо иметь лишь установленный CPython), вы сможете установить конкретную версию Python из соответствующего дистрибутива исходного кода. Кроме того, для установки Python на популярных платформах можно (и это крайне желательно!) использовать альтернативный вариант установки из предварительно собранных бинарных дистрибутивов.

Установка Python при наличии предустановленной версии

Если ваша платформа поставляется с предустановленной версией Python, лучше всего установить еще одну, независимую обновленную версию. При этом исходную версию не следует удалять или заменять — используйте параллельный вариант установки. Это будет гарантией того, что вы не вмешаетесь в установку программного обеспечения, являющегося частью платформы: такое ПО может быть рассчитано на работу с конкретной версией Python, поставляемой вместе с платформой.

Установка CPython с использованием бинарного дистрибутива выполняется быстрее, существенно уменьшает количество выполняемых вами операций на некоторых платформах и является единственно возможным вариантом, если у вас отсутствует подходящий компилятор C. Установка с помощью дистрибутива исходного кода предоставляет более широкие возможности контроля, обеспечивает бо́льшую гибкость и безусловно необходима в тех случаях, когда вам не удается подыскать подходящий бинарный дистрибутив для своей платформы. Но даже если вы устанавливаете Python с помощью бинарного дистрибутива, целесообразно загрузить также дистрибутив исходного кода, поскольку он включает демонстрационные примеры, которые могут отсутствовать в бинарных сборках.

Установка Python из бинарных дистрибутивов

Если вы используете одну из популярных платформ, обновленную до последней версии, то, скорее всего, ваша система уже содержит бинарные пакеты Python, готовые к установке. Как правило, такие пакеты являются самоустанавливающимися и запускаются либо непосредственно в виде исполняемых программ, либо с помощью системных инструментов, таких как менеджер пакетов RedHat (RPM) в некоторых версиях Linux или программа-установщик Microsoft (MSI) в Windows. Предварительно загрузив пакет, установите его, выполнив программу и выбрав такие параметры установки, как каталог (папку), в который должен быть установлен Python. В Windows выберите опцию Add Python 3.5 to PATH. В этом случае программа-установщик автоматически добавит путь к каталогу установки в переменную среды `PATH`, что упростит работу с Python из командной строки (раздел "Программа `python`" в главе 2).

Чтобы загрузить "официальный" бинарный дистрибутив Python, посетите сайт Python (`www.python.org`), щелкните на кнопке Downloads, а после открытия следующей страницы — на кнопке Python 2.7 или Python 3.x для загрузки бинарного дистрибутива, соответствующего вашей платформе.

Многие сторонние производители предлагают бесплатные программы-установщики для различных платформ. Существуют установщики как для дистрибутивов Linux на основе RPM-пакетов (RedHat, Fedora, Mandriva, SUSE и др.; `rpmfind.net`), так и для Debian и Ubuntu (`www.debian.org`). На странице Other Platforms (`https://www.python.org/download/other/`) вы найдете ссылки на бинарные дистрибутивы, предназначенные для установки на таких довольно экзотических в наши дни платформах, как OS/2, RISC OS, IBM AS/400 и т.п.

Anaconda (`https://www.continuum.io/downloads`) — это бинарный дистрибутив, включающий либо версию v2, либо версию v3 плюс менеджер пакетов *conda* (`https://conda.io/docs/`) и сотни расширений сторонних производителей, предназначенных для научных, математических и инженерных расчетов и анализа данных. Пакет Anaconda доступен для Linux, Windows и macOS. Также доступен пакет Miniconda (`https://conda.io/miniconda.html`), отличающийся от пакета Anaconda только тем, что в нем отсутствуют все вышеперечисленные расширения (однако возможна выборочная установка любого их подмножества с помощью менеджера пакетов *conda*).

macOS

На момент выхода этой книги операционная система macOS для компьютеров Apple поставлялась с предустановленной версией v2 (только текстовый режим). Однако мы рекомендуем установить самую последнюю версию Python и всех улучшений, руководствуясь инструкциями и ссылками, приведенными на странице Downloads сайта Python (`https://www.python.org/downloads/`). Учитывая периодичность выпуска версий продуктов Apple, версия Python, включенная в вашу версию macOS, может быть устаревшей или неполной. Самая последняя версия Python

устанавливается параллельно той, которая была поставлена компанией Apple. Для реализации некоторого программного обеспечения, распространяемого в качестве составной части macOS, компания Apple использует собственную версию Python и проприетарные расширения, поэтому любое вмешательство в исходную установку Python чревато нарушением ее нормальной работы.

Популярный менеджер пакетов с открытым исходным кодом Homebrew (`https://brew.sh/index_ru.html`) предлагает простую процедуру установки в macOS версий Python (`https://github.com/Homebrew/brew/blob/master/docs/Homebrew-and-Python.md`) обеих линеек, v2 и v3. Имейте, однако, в виду, что обычно для каждой из линеек доступен только один выпуск.

Установка Python из исходного кода

Для установки CPython из исходного кода необходима платформа с компилятором C, совместимым со стандартом ISO, и инструменты наподобие утилиты `make`. Обычно установка Python в Windows выполняется с помощью Visual Studio (идеальный вариант — VS2008 или VS2010 для v2 и VS 2015 для v3). Чтобы загрузить исходный код Python, посетите сайт Python (`www.python.org`), щелкните на кнопке Downloads и выберите нужный вариант: v2 или v3. Чтобы получить доступ к расширенному списку опций, наведите указатель мыши на кнопку Downloads и воспользуйтесь всплывающим меню. Расширение *.tgz* имени файла, появляющегося после щелчка на ссылке Gzipped source tarball, эквивалентно расширению *.tar.gz* (т.е. соответствует файловому tar-архиву, сжатому с помощью популярной мощной утилиты сжатия `gzip`). Если в вашем распоряжении имеются инструменты библиотеки XZ, можете воспользоваться ссылкой XZ compressed source tarball, обеспечивающей доступ к версии с расширением *.tar.xz* вместо *.tgz*, сжатой с помощью еще более мощного компрессора `xz`.

Microsoft Windows

В Windows, если вы недостаточно хорошо знакомы с Visual Studio и не привыкли работать в текстовом режиме командной строки, установка Python из исходного кода может доставить определенные хлопоты.

Если выполнение приведенных ниже инструкций представляется для вас затруднительным, следуйте процедуре, описанной в разделе "Установка Python из бинарных дистрибутивов". В любом случае наиболее оптимальный вариант — это установка из бинарных дистрибутивов, даже если одновременно вы хотите выполнить установку из исходного кода. Если, работая с версией, установленной из исходного кода, вы замечаете нечто необычное, дважды щелкните на файле запуска Python, установленном из бинарного дистрибутива. Если после этого странности в поведении Python исчезнут, то вы будете знать, на каком значке следует щелкать в дальнейшем.

В последующих разделах для определенности предполагается, что вы создали новую папку *%USERPROFILE%\py* (например, *c:\users\tim\py*). Перейдите в эту папку, введя,

например, `%USERPROFILE%` в адресной строке Проводника. Загрузите исходный *.tgz*-файл, например файл *Python-3.5.2.tgz* (или файл любой другой выбранной вами версии), в эту папку. Разумеется, имя и место расположения файла вы выбираете по своему усмотрению: имя, которое выбрали мы, приведено лишь в качестве примера.

Извлечение и распаковка исходного кода Python

Извлечение и распаковка *.tgz-* или *.tar.xz*-файла могут быть выполнены с помощью бесплатной программы 7-Zip. Загрузите и установите программу 7-Zip со страницы загрузки (`http://www.7-zip.org/download.html`) и запустите ее для *.tgz*-файла (например, файла *c:\users\tim\py\Python-3.5.2.tgz*), который вы перед этим загрузили. (Загруженный *.tgz*-файл мог оказаться в папке *%USERPROFILE%\downloads*; в этом случае вы должны переместить его в свою папку *\py*, прежде чем распаковывать его с помощью программы 7-Zip.) Теперь у вас имеется папка *%USERPROFILE%\py\Python-3.5.2* (или папка с другим именем, соответствующим загруженной версии), являющаяся корнем иерархической структуры папок, содержащих полный стандартный дистрибутив Python в виде исходного кода.

Сборка Python на базе исходного кода

Откройте текстовый файл *%USERPROFILE%\py\Python-3.5.2\PCBuild\readme.txt* (или файл с другим именем, соответствующим загруженной версии Python) в любом удобном для вас текстовом редакторе и следуйте приведенным в нем подробным инструкциям.

Unix-подобные платформы

На Unix-подобных платформах процесс установки Python из исходного кода обычно выглядит проще. В следующих разделах для определенности предполагается, что вы создали новый каталог *~/Py* и загрузили в него *.tgz*-файл с исходными кодами, например файл *Python-3.5.2.tgz* (или файл с другим именем, соответствующим загруженной версии Python). Разумеется, имя и расположение каталога вы выбираете по своему усмотрению: имя, которое выбрали мы, приведено лишь в качестве примера.

Распаковка архивированных файлов с исходным кодом Python

Файл *.tgz* или *.tar.xz* можно распаковать с помощью популярной утилиты GNU `tar`. Для этого достаточно ввести в командной строке оболочки следующие команды.

```
$ cd ~/Py
$ tar xzf Python-3.5.2.tgz
```

Теперь у вас имеется каталог *~/Py/Python-3.5.2*, который является корнем иерархической структуры каталогов, содержащих полный стандартный дистрибутив Python в виде исходного кода.

Конфигурирование, сборка и тестирование

В файле *~/Py/Python-3.5.2/README* в разделе "Build instructions" содержатся подробные инструкции, которые мы рекомендуем тщательно изучить. Однако в простейшем случае все, что от вас требуется, сводится к следующим командам.

```
$ cd ~/Py/Python-3.5.2
$ ./configure
    [процедура выводит большой объем информации - она здесь опущена]
$ make
    [выполнение команды make занимает некоторое время
     и сопровождается выводом большого объема информации]
```

Если выполнению команды `make` не предшествовало выполнение команды `./configure`, то команда `make` неявно выполнит ее. Когда `make` закончит свою работу, убедитесь в том, что установленная версия Python работает, как ожидается.

```
$ make test
    [выполнение команды занимает некоторое время и сопровождается
     выводом большого объема информации]
```

Как правило, команда `make test` подтверждает работоспособность сборки, но при этом выводит сообщения о том, что часть тестов была опущена ввиду отсутствия некоторых необязательных модулей.

Некоторые модули платформозависимы (например, могут выполняться только на платформах, работающих под управлением устаревшей операционной системы Irix компании SGI), но вам нечего беспокоиться по этому поводу. В то же время тестирование других модулей может быть пропущено по той причине, что они зависят от других пакетов с открытым исходным кодом, не установленных на вашей машине. Например, модуль `_tkinter`, который требуется для выполнения пакета Tkinter (кросс-платформенная библиотека для разработки графического интерфейса) и IDLE (интегрированная среда разработки, поставляемая вместе с Python), может быть установлен лишь в том случае, если команде `./configure` удается найти на вашем компьютере установку связки Tcl/Tk 8.0 или более поздней ее версии. Более подробную информацию относительно особенностей установки Python на различных Unix-подобных платформах можно найти в файле *~/Py/Python-3.5.2/README*.

Установка из исходного кода допускает несколько вариантов настройки конфигурации. Например, можно создать специальную сборку Python, позволяющую отслеживать утечку памяти при разработке расширений Python на языке C, о чем пойдет речь в разделе "Создание и установка C-расширений Python" главы 24. Информацию о доступных опциях настройки конфигурации можно получить, выполнив команду `./configure --help`.

Установка после сборки

По умолчанию команда `./configure` подготавливает Python к установке в каталогах */usr/local/bin* и */usr/local/lib*. Эти настройки можно изменить, выполнив команду

./configure с опцией --prefix до выполнения команды make. Например, если вы хотите установить Python в подкаталог *py35* своего домашнего каталога, то выполните следующие команды.

```
$ cd ~/Py/Python-3.5.2
$ ./configure --prefix=~/py35
```

После этого выполните команду make, как в предыдущем разделе. Создав сборку Python и протестировав ее, выполните фактическую установку всех файлов с помощью такой команды:

```
$ make install
```

Эту команду следует выполнять от имени пользователями, обладающего правом записи в целевые каталоги. В зависимости от выбранных вами целевых каталогов и условий доступа к ним, вам может понадобиться выполнить команду make install от имени пользователя *root*, *bin* или другого, воспользовавшись командой su. Распространенной идиомой для этих целей является команда sudo make install: если sudo затребует пароль, введите свой текущий пароль пользователя, а не пароль пользователя *root*. Суть альтернативного (и рекомендуемого) подхода заключается в установке Python в виртуальном окружении, о чем будет рассказано в разделе "Окружения Python" главы 6.

Установка Jython

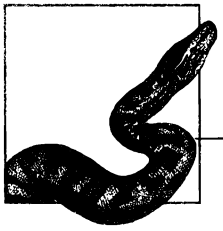
Чтобы загрузить Jython, посетите домашнюю страницу Jython (http://www.jython.org/) и перейдите по ссылке Download. На момент выхода книги самой свежей версией была Jython 2.7.0, поддерживающая Python 2.7 и поставляемая в виде двух исполняемых JAR-файлов, один из которых предназначен только для выполнения Jython, а второй — для его внедрения в Java-приложения. Выполните установку, следуя инструкциям, приведенным по адресу https://wiki.python.org/jython/InstallationInstructions.

Установка IronPython

Для установки IronPython требуется, чтобы на вашей машине была установлена текущая реализация среды Common Language Runtime (CLR, также известна как .NET). С IronPython хорошо работают как Mono (www.mono-project.com), так и Microsoft .NET Framework. Чтобы загрузить IronPython, посетите домашнюю страницу IronPython (ironpython.net) и щелкните на стрелке с надписью Download IronPython 2.7. Получив файл программы-установщика, запустите его для установки IronPython.

Установка PyPy

Чтобы установить PyPy из исходного кода, загрузите (`http://pypy.org/download.html`) исходные файлы выбранной вами версии PyPy (на момент выхода книги последней версией была PyPy 5.6.0, но пусть вас не смущает нумерация — она полностью поддерживает версию Python 2.7, обозначаемую в книге как v2) и следуйте инструкциям, приведенным по адресу `http://pypy.org/download.html#building-from-source`. Альтернативный вариант заключается в загрузке бинарного установщика для вашей платформы (если такая возможность поддерживается), его распаковке из сжатого файла формата *.zip* или *.tar.bz2* и запуске результирующего исполняемого файла.



2

Интерпретатор Python

В процессе разработки программных систем с помощью Python вы создаете текстовые файлы, содержащие исходный код и документацию. Для этого подойдет любой текстовый редактор, в том числе из тех, которые поставляются вместе с интегрированными средами разработки (Integrated Development Environment — IDE). Далее вы обрабатываете файлы с помощью компилятора и интерпретатора Python. Это можно делать либо непосредственно, либо в IDE, либо с помощью другой программы, в которую внедрен Python. Как и IDE, интерпретатор Python позволяет выполнять код, написанный на языке Python, в интерактивном режиме.

Программа `python`

Интерпретатор Python выполняется в виде программы `python` (в Windows она называется *python.exe*). Программа `python` включает как интерпретатор, так и компилятор Python, который неявно вызывается для импортируемых модулей по мере необходимости. В типичных случаях, в зависимости от вашей системы, эта программа должна находиться в папке, указанной в переменной среды `PATH`. Если это не так, может потребоваться определение полного пути доступа к программе при ее запуске с помощью системной командной строки, сценария оболочки или ярлыка[1].

PEP 397

После выхода документа **PEP 397** (`https://www.python.org/dev/peps/pep-0397/`) в Windows появилась возможность выполнять основную программу Python с помощью программы запуска сценариев *py.exe*, которая устанавливается в системной области Windows и обеспечивает гарантированный запуск `python`, не требуя никаких манипуляций с переменной среды `PATH` с вашей стороны.

[1] Если путь доступа содержит пробелы, то, возможно, его нужно будет заключить в кавычки, но, опять-таки, это зависит от конкретной операционной системы.

Если вы работаете в Windows, нажмите клавишу <Windows> и начните вводить python: выберите в открывшемся списке строку *"Python 3.5 (command-line)"* (предполагается, что на вашем компьютере установлена версия v3), наряду с которой будут отображаться и другие варианты выбора, такие как *"IDLE (Python GUI)"*. Альтернативным способом запуска Python является ввод в командной строке команды py, при условии, что в вашей системе установлена программа запуска сценариев *py.exe* (она устанавливается автоматически версией v3; для версии v2 это необходимо сделать самостоятельно: `https://bitbucket.org/vinay.sajip/pylauncher/downloads/`). При наличии нескольких установленных версий Python запускается версия v2, установленная последней, однако это поведение можно изменить путем настройки конфигурационного файла или задания параметра командной строки (например, использовав команду `py -3`).

Переменные среды

Помимо переменной `PATH` на выполнение программы `python` влияют другие переменные среды. Влияние некоторых из них аналогично влиянию параметров, передаваемых `python` в командной строке, о которых речь пойдет в следующем разделе. Некоторые переменные среды предлагают настройки, недоступные через параметры командной строки. Приведенный ниже список охватывает лишь те переменные, которые используются чаще всего. Для получения более полных сведений о переменных среды следует обратиться к онлайн-документации (`https://docs.python.org/3/using/cmdline.html#environment-variables`).

`PYTHONHOME`
: Содержит путь к каталогу установки Python. В этом каталоге должен существовать подкаталог *lib*, содержащий стандартную библиотеку Python. В Unix-подобных системах модули стандартной библиотеки должны находиться в каталоге *lib/python-2.7* для Python 2.7, каталоге *lib/python-3.5* для Python 3.5 и т.д. Если эта переменная не задана, интерпретатор будет искать каталог установки, используя набор эвристических правил.

`PYTHONPATH`
: Содержит список каталогов, в котором в качестве символа-разделителя используется двоеточие в Unix-подобных системах и точка с запятой в Windows. Python может импортировать модули из каталогов, указанных в этом списке. Данный список расширяет начальное значение переменной Python `sys.path`. Более подробно о модулях, операции импорта и переменной `sys.path` речь пойдет в главе 6.

`PYTHONSTARTUP`
: Имя файла сценария, который будет выполняться при каждом запуске интерпретатора Python в интерактивном режиме. Если эта переменная не установлена или системе не удается обнаружить файл по указанному в ней пути

доступа, то никакие подобного рода файлы не выполняются. При запуске сценариев Python файл, путь к которому указан в переменной PYTHONSTARTUP, не выполняется; он выполняется лишь при запуске интерактивных сеансов.

Как именно осуществляется установка или просмотр переменных среды, зависит от операционной системы. В Unix для этого можно использовать команды оболочки; нередко это делается с помощью постоянно хранимых сценариев автозапуска. Если вы работаете в Windows, нажмите клавишу <Windows> и начните вводить **переменные среды**. Один из появившихся значков откроет вам доступ к настройке переменных среды на уровне системы, другой — на уровне текущего пользователя. В случае компьютеров Mac можно действовать примерно так же, как в Unix-подобных системах, но имеются и другие возможности, включая специфическую для MacPython IDE. Для получения более подробной информации относительно использования Python на компьютерах Mac посетите следующие страницы: `https://docs.python.org/2/using/mac.html` (v2) или `https://docs.python.org/3.5/using/mac.html` (v3).

Синтаксис и параметры командной строки

Синтаксис командной строки интерпретатора можно кратко представить в следующем виде:

```
[путь]python {параметры} [-c команда | -m модуль | файл | -]
{аргументы}
```

Здесь квадратные скобки ([]) включают необязательные элементы, фигурные ({}) — элементы, количество которых может быть от нуля до нескольких, а вертикальная черта (|) разделяет альтернативные варианты. Как и в Unix, при указании путей доступа к файлам в Python используется косая черта (/).

В своей простейшей форме команда запуска сценария (скрипта) Python из командной строки выглядит примерно так.

```
$ python hello.py
Hello World
```

Если файл сценария находится не в текущем каталоге, то необходимо указать путь доступа к нему.

```
$ python ./hello/hello.py
Hello World
```

В качестве имени файла сценария допускается использование любого абсолютного или относительного пути, и включение в него какого-либо специфического расширения не является обязательным (хотя обычно принято использовать расширение *.py*). В каждой операционной системе существуют свои способы объявления сценариев Python исполняемыми файлами, но в данной книге эти детали не рассматриваются.

Параметры (опции) командной строки представляют собой короткие строки, начинающиеся с дефиса, которые позволяют изменять заданное по умолчанию поведение программы python. Команда python воспринимает лишь параметры,

начинающиеся с символа дефиса (-), а не с символа косой черты. Наиболее часто используемые параметры приведены в табл. 2.1. В описании каждого параметра указывается соответствующая ему переменная среды (если таковая имеется), установка которой определяет аналогичный вариант поведения. Для многих параметров предусмотрены более длинные версии, начинающиеся с двух символов дефиса, о чем можно прочитать в справке, доступ к которой предоставляет команда `python -h`. За более подробной информацией обратитесь к онлайн-документации (`https://docs.python.org/3/using/cmdline.html#command-line`).

Таблица 2.1. Наиболее часто используемые параметры командной строки интерпретатора Python

Параметр	Описание (также указывается эквивалентная переменная среды, если она существует)
`-B`	Если этот параметр задан, Python не будет пытаться записывать на диск скомпилированные файлы, содержащие байт-код
`-c`	Задает выполнение кода Python, указанного в качестве части командной строки
`-E`	Игнорировать значения всех переменных среды, которые, возможно, были установлены
`-h`	Вывести полный список доступных параметров и справочную информацию, после чего прекратить выполнение команды
`-i`	Запустить интерактивный сеанс по завершении выполнения сценария или команды (`PYTHONINSPECT`)
`-m`	Задает модуль или пакет Python, который должен быть выполнен в качестве основного сценария
`-O`	Включает режим оптимизации генерируемого байт-кода (обратите внимание на то, что здесь используется буква "O" в верхнем регистре, а не цифра "0")
`-OO`	Подобен параметру `-O`, но задает удаление строк документации из байт-кода
`-Q` *аргумент*	Управляет поведением оператора целочисленного деления `/` (только в версии v2)
`-S`	Отключить импорт модуля `site`, неявно заданный в сценарии автозапуска (раздел "Модули `site` и `sitecustomize`" в главе 13)
`-t`	Активизировать вывод предупреждающих сообщений о ненадлежащем использованием символов табуляции (`-tt` — то же самое, но в отношении сообщений об ошибках)
`-u`	Использовать небуферизованные двоичные файлы для стандартных потоков ввода-вывода и ошибок (`PYTHONUNBUFFERED`)
`-v`	Активизировать вывод отладочной информации всякий раз, когда импортируется модуль или освобождаются используемые им системные ресурсы (`PYTHONVERBOSE`)
`-V`	Вывести номер версии Python и выйти из программы интерпретатора

Параметр	Описание (также указывается эквивалентная переменная среды, если она существует)
`-W` *аргумент*	Добавить запись в фильтр сообщений (раздел "Фильтры" в главе 16)
`-x`	Исключить (пропустить) первую строку в файле с исходным кодом основного сценария

Используйте параметр `-i`, если хотите открыть интерактивный сеанс сразу же после запуска сценария с гарантией того, что вы не сможете изменить переменные верхнего уровня и они будут оставаться доступными для просмотра. Указывать данный параметр при обычном запуске интерактивного сеанса не следует, хотя никакого вреда от этого не будет. Параметры `-t` и `-tt` гарантируют, что символы пробела и табуляции в исходном коде Python используются в соответствии с рекомендациями (для получения более подробной информации относительно использования пробельных символов в Python см. раздел "Строки и отступы" главы 3).

Параметры `-O` и `-OO` обеспечивают определенную экономию времени и памяти за счет оптимизации байт-кода, генерируемого для импортируемых модулей, путем превращения инструкций `assert` в инструкции, которым соответствует отсутствие операции, о чем подробнее рассказано в разделе "Инструкция `assert`" главы 5. Кроме того, при установленном параметре `-OO` строки документирования игнорируются[2]. Параметр `-Q` (только в версии v2) определяет поведение оператора деления, если он применяется к двум целочисленным операндам (операция деления рассматривается в разделе "Арифметические операции" главы 3). Параметр `-W` задает добавление записей в фильтр предупреждающих сообщений (о чем подробнее рассказано в разделе "Фильтры" главы 16).

В версии v2 параметр `-u` принудительно вводит двоичный режим для стандартных потоков ввода-вывода и ошибок (но не для других файловых объектов). Некоторые платформы, в основном это касается Windows, различают двоичный и текстовый режимы. Двоичный режим нужен для отправки двоичных данных на стандартное устройство вывода. Кроме того, параметр `-u` гарантирует немедленный вывод данных без использования буферизации. Это может понадобиться в ситуациях, когда задержка, вызванная буферизацией, создает проблемы, как это бывает в случае некоторых каналов Unix. В версии v3 параметр `-u` принудительно задает небуферизованный вывод для двоичного слоя стандартных устройств вывода и ошибок (доступного из их атрибута `.buffer`), но не для их текстового слоя.

Вслед за параметрами, если таковые предоставлены, указывается сценарий Python, подлежащий выполнению. Если указывается путь к файлу, то он относится к файлу с исходным кодом или байт-кодом Python, имя которого дополняется расширением при условии, что таковое имеется. На любой платформе в качестве

[2] Это может иметь отрицательные последствия для кода, пытающегося просматривать строки документирования (docstrings) в целях семантического анализа. Мы полагаем, что вы будете избегать написания такого кода.

разделителя компонентов этого пути можно использовать косую черту (/). Только в системах, работающих под управлением Windows, в качестве альтернативного варианта допускается использование обратной косой черты (\). Вместо указания пути к файлу можно использовать флаг `-c`, вслед за которым указывается строка *команда*, содержащая код Python. Обычно строка *команда* содержит пробелы, поэтому лучше заключать ее в кавычки, чтобы удовлетворить требованиям оболочки операционной системы или интерпретатора командной строки. Некоторые оболочки (например, `bash`) разрешают ввод нескольких строк в виде одного аргумента, благодаря чему команда может содержать последовательность инструкций Python. Другие оболочки (например, в Windows) ограничивают вас одной строкой. В таком случае *команда* может содержать только одну или несколько коротких инструкций, разделенных точкой с запятой (;), о чем более подробно рассказано в разделе "Инструкции" главы 3.

Еще один способ указания сценария Python, подлежащего выполнению, обеспечивает опция `-m` *модуль*, которая сообщает Python о том, что необходимо загрузить и выполнить модуль с именем *модуль* (или же элемент *__main__.py* пакета или ZIP-файла с именем *модуль*) из каталога, указанного в переменной Python `sys.path`. Этот способ удобно использовать в отношении нескольких модулей, входящих в стандартную библиотеку Python. Например, использование опции `-m timeit` нередко является наилучшим способом тестирования быстродействия инструкций и выражений Python на микроуровне (раздел "Модуль `timeit`" в главе 16).

Наличие дефиса или отсутствие любой лексемы в этой позиции сообщает интерпретатору о том, что исходный код программы должен быть получен из стандартного ввода, т.е., как правило, из интерактивного сеанса. Явное указание дефиса требуется только в том случае, если за ним следуют аргументы. Здесь параметр *аргументы* представляет произвольные строки. Выполняющееся приложение Python может получить доступ к ним как к элементам списка `sys.argv`.

Например, в случае стандартной установки в Windows результатом выполнения в командной строке следующей команды будет вывод текущих значений даты и времени:

```
C:\> py -c "import time; print(time.asctime())"
```

В случае Cygwin, Linux, OpenBSD, macOS и других Unix-подобных систем, в которых Python установлен в каталог, предусмотренный по умолчанию в дистрибутиве, выполнение следующей команды запустит интерактивный сеанс с выводом сообщений о загрузке модулей и освобождении занимаемой ими памяти:

```
$ /usr/local/bin/python -v
```

Если исполняемый файл Python находится в каталоге, включенном в переменную среды `PATH`, то ввод команды можно начать со слова **python**, не указывая полный путь к исполняемому файлу. (Если на компьютере установлено несколько версий Python, можно указать конкретный номер нужной версии, например: `python2`, `python2.7`, `python3` или `python3.5`. Ввод команды `python` без указания номера версии в данном случае обычно означает использование версии, которая была установлена последней.)

Интерактивные сеансы

Если команда `python` введена без указания аргумента, определяющего сценарий, то интерпретатор Python запускается в интерактивном режиме, предлагая вам вручную вводить инструкции и выражения. Интерактивные сеансы очень удобны для изучения результатов выполнения команд, просмотра содержимого объектов и проведения вычислений, в которых Python используется в качестве мощного калькулятора с расширяемыми возможностями. (Реализация IPython, которая обсуждалась в разделе "IPython" главы 1, особенно удобна для интерактивного режима.)

Если вы введете завершенную инструкцию, то Python выполнит ее. Если вы введете завершенное выражение, Python вычислит его. Если выражение возвращает результат, то Python выведет строку, представляющую данный результат, и при этом присвоит данный результат специальной переменной _ (символ подчеркивания), чтобы впоследствии его можно было использовать в другом выражении. Если Python ожидает ввода инструкции или выражения, то строка приглашения к вводу имеет вид >>>; если же вы начали ввод инструкции или выражения, но не завершили его, то в качестве приглашения к продолжению ввода выводится многоточие (. . .). В частности, Python выводит приглашение в виде . . . в тех случаях, когда закрывающая скобка, соответствующая скобке, открытой в предыдущей строке, еще не была введена.

Чтобы завершить интерактивный сеанс работы с интерпретатором Python, введите символ конца файла (комбинация клавиш <Ctrl+Z> в Windows и <Ctrl+D> в Unix-подобных системах). Возбуждение исключения с помощью инструкции `raise SystemExit`, равно как и вызов функции `sys.exit()`, также завершает интерактивный сеанс, независимо от того, как это было сделано: в интерактивном режиме или из выполняющегося кода (исключения обсуждаются в главе 5).

Возможности обработки текста в окне командной строки и работы с историей команд частично зависят от того, каким образом была выполнена сборка Python: если в нее был включен модуль `readline`, то вам будут доступны средства *GNU readline* — свободной библиотеки для интерфейса командной строки и обработки строк. В Windows для интерактивных программ наподобие `python`, выполняющихся в текстовом режиме, предусмотрены простые, но удобные возможности для работы с историей команд. Также можно использовать другие аналогичные средства, установив пакет `pyreadline` для Windows (`https://pypi.python.org/pypi/pyreadline/2.0`) или `pyrepl` для Unix (`https://pypi.python.org/pypi/pyrepl/0.8.4`).

В дополнение к встроенной интерактивной среде Python и тем оболочкам, которые предлагаются в составе более совершенных сред разработки, рассмотренных в следующем разделе, вы можете бесплатно загрузить другие мощные IDE. Одной из наиболее популярных из них является среда IPython (`ipython.org`), упомянутая в разделе "IPython" главы 1 и предлагающая чрезвычайно широкий набор возможностей. Более простой, облегченной, но тем не менее весьма удобной и перспективной альтернативой является интерпретатор `bpython` (`bpython-interpreter.org`), расширяющий возможности стандартной оболочки Python.

Среды разработки Python

Встроенный интерактивный режим интерпретатора — простейшая среда разработки Python. Она немного примитивна, но зато легковесна, мало нагружает систему и быстро запускается. Вместе с подходящим текстовым редактором (раздел "Бесплатные текстовые редакторы с поддержкой Python") и дополнительными средствами для работы с командной строкой и историей команд интерактивный интерпретатор (или его альтернатива — гораздо более мощный интерпретатор команд IPython/Jupyter) представляет собой удобную в использовании и очень популярную среду разработки. Однако есть целый ряд других сред разработки, которые вы также можете использовать.

IDLE

Интегрированная среда разработки Python (IDLE) поставляется вместе со стандартным дистрибутивом Python для большинства платформ. Она представляет собой кросс-платформенное приложение, созданное исключительно средствами Python с использованием библиотеки *Tkinter*. Оболочка Python, предлагаемая IDLE, аналогична интерактивным сеансам Python, но обладает гораздо более широкой функциональностью. К тому же она включает текстовый редактор, оптимизированный для работы с исходным кодом Python, интегрированный интерактивный отладчик и несколько специализированных браузеров/программ просмотра.

Если вам нужно дополнительно расширить функциональность IDLE, загрузите и установите IdleX — внушительную коллекцию расширений от сторонних производителей (`http://idlex.sourceforge.net/`).

Чтобы установить и использовать IDLE в macOS, руководствуйтесь инструкциями, приведенными по следующему адресу:

`https://www.python.org/download/mac/tcltk/`

Другие IDE для Python

IDLE — это зрелое, стабильное, простое в использовании, наделенное богатыми функциональными возможностями и расширяемое приложение. Однако существует множество других IDE — кросс-платформенных и платформозависимых, бесплатных и коммерческих (в том числе коммерческие IDE с вариантами бесплатной поставки, особенно если вы являетесь разработчиком программ с открытым исходным кодом), как в виде отдельных приложений, так и в виде надстроек к другим IDE.

Некоторые из этих IDE отличаются такими возможностями, как статический анализ, средства создания графического интерфейса пользователя (GUI), отладчики и т.п. Список, содержащий свыше 30 IDE, вместе с десятком ссылок на внешние страницы с обзорами и сравнительными статьями приведен на вики-странице IDE Python (`https://wiki.python.org/moin/IntegratedDevelopmentEnvironments`). Если вы коллекционируете IDE — удачной охоты!

Мы не имеем возможности подробно рассказать даже о некоторых из них, но вам стоит обратить внимание на популярную модульную кросс-платформенную IDE *Eclipse* (`www.eclipse.org`), предназначенную для разработки программ на множестве языков: бесплатный плагин PyDev для *Eclipse* от сторонних разработчиков (`www.pydev.org`) обеспечивает великолепную поддержку Python. Высокую оценку разработчиков получила *Wing IDE* (`wingware.com`), которой вот уже длительное время пользуется автор книги Стив Холден. Возможно, наиболее популярной из всех IDE для Python на сегодняшний день является *PyCharm* (`https://www.jetbrains.com/pycharm/`).

Если вы пользуетесь Visual Studio, ознакомьтесь с *PTVS* — бесплатным расширением с открытым исходным кодом, которое особенно хорошо подходит для отладки смешанного кода на языках Python и C (`https://github.com/Microsoft/PTVS/wiki/PTVS-Installation`)[3].

Бесплатные текстовые редакторы с поддержкой Python

Для редактирования исходного кода Python вам подойдет любой текстовый редактор, даже самый простейший, включая Блокнот в Windows и *ed* в Linux. Мощные бесплатные редакторы поддерживают Python, предоставляя такие возможности, как подсветка синтаксиса и автоматическое создание отступов. Кросс-платформенные редакторы обеспечивают единообразный стиль работы на различных платформах. Хорошие текстовые редакторы также предоставляют командный режим, позволяющий выполнять другие программы с использованием редактируемого кода. С обновляемым списком редакторов для Python, в настоящее время включающим не один десяток позиций, можно ознакомиться на вики-странице Python по адресу `https://wiki.python.org/moin/PythonEditors`.

Вероятно, наиболее широким спектром возможностей редактирования исходных текстов Python по-прежнему может считаться классический редактор *Emacs* (`https://www.gnu.org/software/emacs/`). (Список множества других дополнений, специфических для Python, приведен на вики-странице по адресу `https://wiki.python.org/moin/EmacsEditor`.) Однако научиться работать с Emacs не так-то просто, и к тому же его нельзя назвать легковесным. Автор книги Алекс Мартелли отдает предпочтение другому классическому редактору — *vim* (`www.vim.org`), который представляет собой разработанную Брамом Моленаром современную улучшенную версию традиционно используемого в Unix редактора *vi*. Пожалуй, *vim* несколько уступает редактору Emacs, но может составить ему конкуренцию: его достоинствами являются быстродействие, легковесность, поддержка Python, способность выполняться как в чисто текстовом режиме, так и с использованием графического интерфейса. Специфические для Python рекомендации и список дополнений приведены на вики-странице Python (`https://wiki.python.org/moin/Vim`). Авторы книги Стив Холден и Анна Рейвенскрофт также используют *vim*. Дополнительно, когда

[3] См. также статью одного из разработчиков PTVS Павла Минаева, доступную по адресу `https://habrahabr.ru/company/microsoft/blog/185412/`. — *Примеч. ред.*

это возможно, Стив применяет коммерческий редактор *Sublime Text 2*, который предлагает цветовое выделение синтаксиса и достаточно развитые средства интеграции, позволяющие запускать программы непосредственно из редактора.

Средства проверки программ на Python

Проверка программ и модулей, осуществляемая компилятором Python, не отличается тщательностью: проверяется только синтаксис кода. Если требуется более тщательная проверка кода, можете загрузить и установить сторонние средства, предназначенные для этих целей. *Pyflakes* (`https://pypi.python.org/pypi/pyflakes`) — весьма быстрая облегченная программа проверки: она проверяет лишь отсутствие логических ошибок, но не импортирует проверяемые модули, что повышает безопасность ее использования. На другом конце спектра находится программа *PyLint* (`www.pylint.org`), очень быстрая и с богатейшими возможностями настройки конфигурации. PyLint не относится к числу облегченных средств, но это окупается очень гибкими возможностями детального управления стилями проверки за счет использования настраиваемых конфигурационных файлов.

Выполнение программ на Python

Независимо от того, с помощью каких инструментов было создано приложение на Python, вы можете просмотреть его в виде файлов с исходным кодом, которые представляют собой обычные текстовые файлы. Сценарий (скрипт) — это файл, который можно выполнить непосредственно. Модуль — это файл, который можно импортировать (подробнее о модулях — в главе 6) для предоставления некоторой функциональности другим файлам или интерактивному сеансу. Файл Python можно использовать как в качестве модуля (добавляющего функциональность при его импорте), так и в качестве сценария (выполняемого непосредственно). В соответствии с полезным общепринятым соглашением файлы Python, которые в основном предназначены для импортирования в качестве модулей, при их непосредственном запуске должны выполнять некоторые простые операции самотестирования, о чем пойдет речь в разделе "Тестирование" главы 16.

Интерпретатор Python автоматически компилирует файлы с исходным кодом по мере надобности. Как правило, файлы с исходным кодом на языке Python имеют расширение *.py*. В версии v2 Python сохраняет файл, содержащий скомпилированный байт-код модуля, в том же каталоге, в котором хранится соответствующий файл с исходным кодом модуля, используя то же имя файла, но с расширением *.pyc* (или *.pyo*, если Python был запущен с опцией `-O`). В версии v3 Python сохраняет скомпилированный код в подкаталоге *__pycache__* того каталога, в котором содержится исходный код модуля, причем используется расширение имени файла, зависящее от версии и указывающее на уровень оптимизации.

Запуск Python с опцией `-B` позволяет избежать сохранения скомпилированного байт-кода на диске, что может пригодиться, если модули импортируются с диска,

открытого только для чтения. Кроме того, Python не сохраняет скомпилированный байт-код непосредственно из запускаемого сценария; вместо этого сценарий заново компилируется при каждом запуске. Python сохраняет файлы с байт-кодом лишь для импортируемых модулей. Он автоматически заново создает файлы с байт-кодом каждого модуля во всех необходимых случаях, например, если его исходный код был изменен. Наконец, в целях развертывания приложения вы можете пакетировать модули Python, используя инструменты, описанные в главе 25.

Код Python можно выполнять интерактивно, используя интерпретатор или IDE. Однако обычно выполнение программы инициируется запуском сценария верхнего уровня. Чтобы запустить сценарий, передайте путь к нему команде `python` в качестве аргумента, о чем уже говорилось в разделе "Программа `python`". В зависимости от операционной системы команду `python` можно выполнять непосредственно из оболочки сценария или из командной строки. В Unix-подобных системах файл сценария можно сделать исполняемым, установив для него биты `x` и `r` в разрешениях доступа и поместив в начале файла так называемую *магическую строку* следующего вида:

```
#!/usr/bin/env python
```

Это может быть любая другая строка, начинающаяся символами `#!` (жарг. *shebang*, произн. *шебанг*), за которыми следует путь к программе интерпретатора *python*. В таком случае у вас имеется возможность указать также слово, содержащее параметры, например:

```
#!/usr/bin/python -O
```

В соответствии с документом **PEP 397**, теперь и в Windows разрешается использовать магическую строку в стиле "шебанг" (`#!`) для указания конкретной версии Python, что обеспечивает возможность свободной миграции сценариев между Unix-подобными и Windows-системами. Кроме того, вы сможете использовать для запуска сценариев Python обычные механизмы Windows, такие как двойной щелчок на значке сценария. В последнем случае окно консоли текстового режима, ассоциированное с данным сценарием, закрывается сразу же после того, как он завершит свою работу. Если немедленное закрытие окна консоли для вас нежелательно (например, вы хотите предоставить пользователю возможность прочитать результаты, выведенные сценарием), примите меры к тому, чтобы это не происходило так быстро. Например, в случае версии v3 можно поместить в конце сценария следующую строку:

```
input('Для завершения нажмите клавишу Enter')
```

(В версии v2 следует использовать функцию `raw_input()`.) Этого не требуется делать, если сценарий запускается из командной строки.

Кроме того, в Windows вместо файлов с расширением *.py* и интерпретатора *python.exe* можно использовать файлы с расширением *.pyw* и интерпретатор *pythonw.exe*. В этом случае Python выполняется без консоли текстового режима и стандартных устройств ввода-вывода. Такой вариант удобно использовать для сценариев, использующих графический интерфейс пользователя или выполняющихся в виде

фоновых задач. Прибегайте к данному варианту лишь в случае полностью отлаженных сценариев, чтобы оставить стандартные устройства вывода доступными для вывода информации, предупреждений и сообщений об ошибках в процессе разработки. На компьютерах Mac, если вы хотите выполнить сценарий, которому требуется доступ к графическому интерфейсу пользователя, а не просто интерактивное взаимодействие в текстовом режиме, используйте интерпретатор *pythonw* вместо *python*.

Приложения, написанные на других языках, могут внедрять Python и управлять его выполнением для собственных целей. Этот вопрос вкратце обсуждается в разделе "Внедрение Python" главы 24.

Интерпретатор Jython

Интерпретатор Jython, сборка которого осуществляется в процессе установки (см. раздел "Установка Jython" главы 1), запускается аналогично программе `python`:

`[`*путь*`]`**`jython`** `{`*параметры*`}` `[` **`-j`** *jar-файл* `|` **`-c`** *команда* `|` *файл* `|` `-` `]` `{`*аргументы*`}`

Опция `-j` *jar-файл* сообщает Jython о том, что в качестве основного сценария должен выполняться сценарий *__run__.py*, находящийся в *.jar*-файле. Параметры `-i`, `-S` и `-v` имеют тот же смысл, что и в случае `python`, тогда как параметр `--help` аналогичен параметру `-h`, а параметр `--version` — параметру `-V`. Вместо переменных среды Jython использует текстовый файл с именем *registry*, находящийся в каталоге установки и предназначенный для записи свойств, имеющих структурированные имена. Например, в Jython свойство `python.path` эквивалентно переменной среды `PYTHONPATH` в Python. Кроме того, можно устанавливать значения свойств в командной строке `jython` с помощью опций, следуя синтаксису `-D` *имя=значение*.

Более полную обновленную информацию вы найдете на главной странице сайта Jython (`www.jython.org`).

Интерпретатор IronPython

IronPython можно выполнить аналогично программе `python`:

`[`*путь*`]`**`ipy`** `{`*параметры*`}` `[`**`-c`** *команда* `|` *файл* `|` `-` `]` `{`*аргументы*`}`

Более полную обновленную информацию вы найдете на главной странице сайта IronPython (`ironpython.net`).

Интерпретатор PyPy

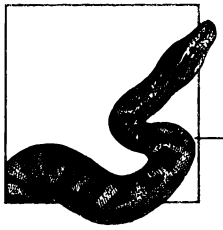
PyPy можно выполнить аналогично программе `python`:

`[`*путь*`]`**`pypy`** `{`*параметры*`}` `[`**`-c`** *команда* `|` *файл* `|` `-` `]` `{`*аргументы*`}`

Более полную обновленную информацию вы найдете на главной странице сайта PyPy (`pypy.org`).

II

Ядро и встроенные объекты Python



3

Язык программирования Python

Эта глава представляет собой путеводитель по языку Python. Тем читателям, которые нуждаются в изучении Python с самых азов, рекомендуем начать с посещения веб-страницы по адресу `https://www.python.org/about/gettingstarted/`, выбрав соответствующую ссылку в зависимости от того, являетесь ли вы начинающим программистом или успели приобрести некоторый опыт в программировании. Но если вы уже хорошо владеете другими языками программирования и вам просто нужно ознакомиться со спецификой Python, то эта глава — как раз то, что вам нужно. Мы не пытаемся обучить вас языку Python в одной главе: здесь лишь приведен достаточно большой объем базовых сведений в лаконичном изложении. Основное внимание уделено рассмотрению правил языка, а не обсуждению наилучших практик программирования и стиля написания программ на языке Python. В качестве основного руководства по стилевому оформлению кода руководствуйтесь документом **PEP 8** (`https://www.python.org/dev/peps/pep-0008/`), попутно ознакомившись с дополнительными рекомендациями опытных программистов (`http://docs.python-guide.org/en/latest/writing/style/`), а также с рекомендациями, приведенными на сайтах CKAN (`http://docs.ckan.org/en/latest/contributing/python.html`) и Google (`https://google.github.io/styleguide/pyguide.html`).

Лексическая структура

Лексическая структура языка программирования — это набор базовых правил, определяющих написание программ на данном языке. Ей соответствует самый низкий уровень синтаксиса языка, задающий, например, правила именования переменных или обозначение комментариев. Как и любой другой текстовый файл, файл, содержащий исходный код программы на языке Python (исходный файл), представляет собой последовательность символов. Такой файл также удобно рассматривать как

последовательность строк, лексем или инструкций (команд). Эти лексические аспекты являются взаимодополняющими. Python чрезвычайно придирчив в отношении компоновки программы, особенно во всем, что касается использования строк и отступов: обратите на это особое внимание, если приступаете к разработке на Python, имея опыт программирования на других языках.

Строки и отступы

Программа на языке Python представляет собой последовательность *логических строк*, каждая из которых может состоять из одной или нескольких *физических строк*. Каждая физическая строка может заканчиваться комментарием. Начало комментария обозначается символом "решетка" (#), не входящим в состав строкового литерала. Все символы, расположенные между символом # и символом конца строки (не включая его), являются комментарием: Python игнорирует эти символы. Строка, содержащая лишь пробелы и, возможно, комментарии, называется *пустой*: Python игнорирует такие строки[1].

В Python конец физической строки в большинстве случаев означает конец инструкции. В отличие от других языков программирования, завершать инструкции Python какими-либо символами-разделителями, например точкой с запятой (;), обычно не требуется. Длинную инструкцию, не умещающуюся в одной строке, можно продолжить на следующей, поместив в конце первой строки символ обратной косой черты, при условии, что данная строка не содержит комментария. В то же время Python автоматически объединяет смежные физические строки в одну логическую, если какая-либо открытая скобка — круглая (()), квадратная ([]) или фигурная ({}) — еще не была закрыта: используйте преимущества этого механизма, который обеспечивает получение более удобочитаемого кода, чем тот, который получается при использовании символа обратной косой черты в конце строки. Строковые литералы, заключенные в тройные кавычки, также можно располагать на нескольких физических строках. Физические строки, входящие в одну логическую строку, называются *строками продолжения*. Соображения, касающиеся отступов, относятся к первой физической строке каждой логической строки, а не к строкам продолжения.

В Python для организации блочной структуры программ используются *отступы*. В отличие от других языков программирования, именно отступы, а не, скажем, заключение блока кода в фигурные скобки или его ограничение начальным и конечным разделителями, позволяют судить о границах блоков. Отступы создаются путем смещения начала логической строки блока от левого края на определенное количество пробелов. *Блок* — это последовательность смежных логических строк, имеющих одинаковые отступы; логическая строка с меньшим отступом означает конец блока. Все инструкции блока, равно как и все предложения составной инструкции, должны иметь одинаковые отступы. Первая инструкция в файле исходного кода не должна

[1] При работе в интерактивном режиме многострочные инструкции должны завершаться вводом пустой физической строки (без каких-либо пробельных символов и комментариев).

выделяться отступом (т.е. не должна начинаться с пробельного символа). Инструкции, которые вы вводите в ответ на приглашение (подсказку) интерактивного интерпретатора >>> (см. раздел "Интерактивные сеансы" в главе 2), также не должны выделяться отступами.

В версии v2 каждый символ табуляции логически заменяется восемью пробелами, поэтому следующие за ними символы попадают в логические столбцы 9, 17, 25 и т.д. В соответствии с рекомендованным стандартным стилем оформления кода на Python применять табуляцию для создания отступов не следует; для этого необходимо использовать только пробелы, по четыре на каждый уровень отступов.

Создавая отступы, никогда не смешивайте символы табуляции и пробелы, поскольку различные инструменты (например, редакторы, системы обработки сообщений электронной почты, принтеры) обрабатывают табуляцию по-разному. В версии v2, чтобы воспрепятствовать непоследовательному использованию символов табуляции и пробелов в исходном коде, следует передать опцию `-t` или `-tt` интерпретатору Python при его вызове (см. раздел "Синтаксис и параметры командной строки" в главе 2). В версии v3 смешивание символов табуляции и пробелов для создания отступов не допускается.

Используйте пробелы вместо символов табуляции

Мы рекомендуем сконфигурировать используемый вами текстовый редактор таким образом, чтобы он расширял символы табуляции до четырех пробелов, тем самым обеспечивая наличие в вашем коде только пробелов, а не символов табуляции. Благодаря этому все инструменты, включая сам интерпретатор Python, будут обрабатывать отступы в исходном коде согласованным образом. Оптимальный стиль Python предполагает создание отступов, равных четырем пробелам, без использования табуляции.

Наборы символов

В версии v3 исходные файлы могут содержать любые символы Unicode в кодировке UTF-8. (Символы, числовые коды которых принадлежат диапазону чисел от 0 до 127, так называемые *ASCII-символы*, кодируются в UTF-8 одиночными байтами, поэтому в версии v3 исходные файлы в кодировке ASCII также считаются допустимыми.)

В версии v2 исходные файлы обычно содержат ASCII-символы (коды символов находятся в диапазоне 0-127).

Как в версии v2, так и в версии v3 можно сообщить Python о том, что исходный файл записан с использованием другой кодировки. В этом случае Python будет читать файл, применяя указанную кодировку (в версии v2 символы, не являющиеся символами ASCII, разрешается использовать только в комментариях и строковых литералах).

Чтобы сообщить Python о том, что исходный файл записан с использованием нестандартной кодировки, поместите в его начале строку комментария примерно следующего вида:

```
# coding: iso-8859-1
```

Вслед за словом `coding:` указывается имя кодека, известного Python и совместимого с ASCII, например `utf-8` или `iso-8859-1`. Обратите внимание на то, что подобный комментарий воспринимается как *директива кодировки* (также *объявление кодировки*) лишь в том случае, если он находится в начале файла с исходным кодом (возможно, после "магической" строки, о которой шла речь в разделе "Выполнение программ на Python" главы 2). В версии v2 *единственным* результатом применения директивы кодировки является то, что это позволяет использовать символы, не являющиеся символами ASCII, в строковых литералах и комментариях. Лучше всего использовать кодировку `utf-8` во всех текстовых файлах, включая файлы с исходным кодом на языке Python.

Лексемы

Python разбивает каждую логическую строку на ряд элементарных лексических компонентов, так называемых *лексем*. Каждая лексема соответствует подстроке логической строки. К числу обычных типов лексем относятся *идентификаторы*, *ключевые слова*, *операторы*, *разделители* и *литералы*, которые будут рассмотрены в следующих разделах. Для разделения лексем можно свободно использовать пробелы. В некоторых случаях, в частности, когда два идентификатора или ключевых слова располагаются рядом, разделение пробелами является обязательным, иначе Python воспримет их как одиночный идентификатор. Например, `ifx` — идентификатор; чтобы записать ключевое слово `if`, за которым следует идентификатор `x`, между ними необходимо вставить пробел (например, `if x`).

Идентификаторы

Идентификатор — это имя, используемое для задания переменной, функции, класса, модуля или другого объекта. Идентификатор должен начинаться с буквы (в версии v2 это буквы от `A` до `Z` и от `a` до `z`; в версии v3 допускаются также другие символы, классифицируемые в Unicode как буквы) или символа подчеркивания (`_`), за которыми могут следовать или не следовать одна или несколько букв, символы подчеркивания и цифры (в версии v2 это цифры от `0` до `9`; в версии v3 допускаются также другие символы, классифицируемые в Unicode как цифры или *модифицирующие символы*). С таблицей символов Unicode, с которых может начинаться и которые может содержать идентификатор в версии v3, можно ознакомиться по адресу `https://www.dcl.hpi.uni-potsdam.de/home/loewis/table-3131.html`. Регистр имеет значение: буквы нижнего и верхнего регистра различаются. Использование знаков пунктуации, таких как `@`, `$` и `!`, не допускается.

В соответствии с общепринятыми в Python соглашениями имена классов начинаются с прописной буквы, остальные идентификаторы — со строчной. Использование знака подчеркивания в качестве первого символа идентификатора указывает на то, что данный идентификатор является закрытым (частным, приватным). Использование двух следующих подряд символов подчеркивания в качестве первых символов идентификатора указывает на то, что данный идентификатор является *строго закрытым*; если при этом идентификатор заканчивается двумя символами подчеркивания, то это означает, что он является специальным символом, определенным в языке.

Использование символа подчеркивания (_) в интерактивном интерпретаторе

Идентификатор _ (одиночный символ подчеркивания) имеет специальное значение в интерактивных сеансах: интерпретатор связывает этот идентификатор с результатом вычисления последнего выражения.

Ключевые слова

В Python имеются *ключевые слова* (их количество составляет 31 в версии v2 и 33 в версии v3) — идентификаторы, зарезервированные для специального использования. Ключевые слова включают только строчные буквы. Их нельзя использовать в качестве обычных идентификаторов (именно поэтому их иногда называют "зарезервированными словами"). Одни ключевые слова используются в качестве начальной части простых инструкций или предложений составных инструкций, тогда как другие — в качестве операторов. Все ключевые слова будут подробно рассмотрены в этой главе, а также в главах 4–6. В версии v2 имеются следующие ключевые слова.

```
and       continue    except    global    lambda    raise     yield
as        def         exec      if        not       return
assert    del         finally   import    or        try
break     elif        for       in        pass      while
class     else        from      is        print     with
```

В версии v3 `exec` и `print` уже не являются ключевыми словами: они были инструкциями в v2, но теперь, в v3, они функции. (Чтобы использовать функцию `print()` в v2, поместите в начале исходного файла строку `from __future__ import print_function`, как отмечалось во введении.) Лексемы `False`, `None`, `True` и `nonlocal` — новые, дополнительные ключевые слова в v3 (из них `False`, `None` и `True` уже были в v2 в качестве встроенных констант, но технически они не являлись ключевыми словами). Специальные лексемы `async` и `await`, о которых пойдет речь в главе 18, в настоящее время не являются ключевыми словами, но это планируется сделать в версии Python 3.7.

Операторы

В качестве *операторов* в Python используются символы, не являющиеся алфавитно-цифровыми, а также сочетания символов. Список символов, которые Python распознает как операторы (раздел "Выражения и операторы"), приводится ниже.

```
+ - * / % ** // << >> &
| ^ ~ < <= > >= <> != ==
```

В версии v3, но не v2, оператором служит также символ `@` (используется в матричном умножении, которое обсуждается в главе 15), хотя с технической точки зрения он является разделителем.

Разделители

В Python в качестве *разделителей*, применяемых в литеральных выражениях, списках, словарях, множествах, а также в различных инструкциях и в других случаях, используются следующие символы и их сочетания.

```
(  )  [  ]  {  }
,  :  .  '  =  ;  @
+= -= *= /=  //= %=
&= |= ^= >>= <<= **=
```

Точка (`.`) может появляться также в числах с плавающей точкой (например, `2.3`) и мнимых литералах (например, `2.3j`). Последние две строки содержат составные операторы присваивания, которые являются разделителями, но одновременно выполняют операции. Мы продолжим обсуждение синтаксиса операторов, когда будем знакомиться с объектами или инструкциями, в которых используются объекты.

Приведенные ниже символы имеют специальный смысл, если являются частью других лексем.

```
' " # \
```

Символами `'` или `"` окружаются строковые литералы. Символ `#` является признаком начала комментария. Символ `\` в конце физической строки объединяет ее со следующей физической строкой в одну логическую; кроме того, он используется в строках в качестве экранирующего символа. Символы `$` и `?`, все специальные управляющие символы[2] за исключением пробельных, а также, в версии v2, все символы с кодами ISO больше 126 (т.е. символы, которые не являются ASCII-символами, такие, например, как буквы с диакритическими знаками) не могут быть частью текста в программах на языке Python, за исключением случаев, когда они используются в составе комментариев и строковых литералов. (Чтобы стало возможным использовать символы, не являющиеся символами ASCII, в комментариях или строковых

[2] Управляющие символы — это символы, не имеющие графического представления (непечатаемые символы), такие как `\t` (табуляция) и `\n` (перевод строки), которые являются пробельными символами; а также `\a` (звуковой сигнал) и `\b` (возврат на шаг), которые не являются пробельными символами.

литералах в версии v2, первая строка файла с исходным кодом на языке Python должна содержать *директиву кодировки*, рассмотренную в разделе "Наборы символов".)

Литералы

Литерал — это значение (число, строка или контейнер), указанное непосредственно в программе. Ниже приведены примеры того, как могут выглядеть числовые и строковые литералы в Python.

```
42                      # Целочисленный литерал
3.14                    # Литерал числа с плавающей точкой
1.0j                    # Мнимый литерал
'hello'                 # Строковый литерал
"world"                 # Другой строковый литерал
"""Good
night"""                # Строковый литерал, заключенный
                        # в тройные кавычки
```

Сочетая числовые и строковые литералы с соответствующими разделителями, можно создавать литералы, непосредственно определяющие значения контейнерных типов.

```
[42, 3.14, 'hello']     # Список
[]                      # Пустой список
100, 200, 300           # Кортеж
()                      # Пустой кортеж
{'x':42, 'y':3.14}      # Словарь
{}                      # Пустой словарь
{1, 2, 4, 8, 'string'}  # Множество
# Литерала, обозначающего пустое множество, не существует;
# вместо него используйте функцию set()
```

Синтаксис литералов будет подробно рассмотрен в разделе "Типы данных", где обсуждаются различные типы данных, поддерживаемые в Python.

Инструкции

Файлы с исходным кодом Python (исходные файлы) можно рассматривать как последовательность простых и составных инструкций. В отличие от некоторых других языков программирования, в Python не предусмотрены "объявления" или другие аналогичные высокоуровневые синтаксические элементы: в нем используются только инструкции.

Простые инструкции

Простыми называют инструкции, которые не содержат никаких других инструкций. Простая инструкция целиком располагается в пределах одной логической строки. Как и во многих других языках программирования, допускается размещение нескольких простых инструкций в одной логической строке с использованием точки

с запятой (;) в качестве разделителя. Однако рекомендуемый стиль Python предполагает размещение инструкций в отдельных строках, что облегчает чтение кода.

Любое *выражение* может выступать в качестве самостоятельной инструкции (выражения обсуждаются в разделе "Выражения и операторы"). При работе в интерактивном режиме интерпретатор отображает результат вычисления выражения-инструкции, введенного в ответ на приглашение (>>>), и связывает его с глобальной переменной _ (символ подчеркивания). Помимо этого, выражения-инструкции могут быть полезными только при вызове функций (и других *вызываемых объектов*), имеющих побочные эффекты (например, вывод данных, изменение глобальных переменных или возбуждение исключений).

Присваивание — это простая инструкция, назначающая значения переменным (раздел "Атрибуты и элементы объектов"). В Python операция присваивания является инструкцией и поэтому не может входить в выражения.

Составные инструкции

Составными называют инструкции, которые содержат одну или несколько других инструкций и управляют их выполнением. Составная инструкция состоит из одного или нескольких *предложений*, выровненных с использованием одинаковых отступов. Каждое предложение имеет *заголовок*, начинающийся с ключевого слова и заканчивающийся двоеточием (:), за которым следует тело предложения, представляющее собой одну или несколько инструкций. Если тело содержит несколько инструкций, так называемый *блок*, то эти инструкции располагаются на отдельных логических строках вслед за строкой заголовка с отступом вправо на четыре пробела. Лексическим признаком конца блока служит возврат к отступу, равному по величине отступу заголовка предложения (или с меньшим отступом, соответствующим какому-либо предложению составной инструкции, объемлющему данное предложение). Телом предложения также может служить одиночная простая инструкция, располагающаяся вслед за символом двоеточия (:) в той же логической строке, что и заголовок. Кроме того, тело может состоять из нескольких простых инструкций, расположенных в одной и той же логической строке и разделенных символами точки с запятой (;), но, как уже отмечалось, использовать такой стиль в Python не рекомендуется.

Типы данных

Операции, выполняемые в Python, тесно связаны с обрабатываемыми данными. В Python значения данных являются *объектами*; каждый объект, или *значение*, имеет *тип*. Тип объекта определяет, какие операции поддерживаются данным объектом (иными словами, в каких операциях может участвовать данное значение). Кроме того, тип определяет *атрибуты* и *элементы* (если таковые имеются), а также допускается ли изменение объекта. Объект, который может быть изменен, называют *изменяемым*; объект, который нельзя изменить, называют *неизменяемым*. Атрибуты и элементы объекта рассматриваются в разделе "Атрибуты и элементы объектов".

Встроенная функция `type(obj)` получает в качестве аргумента любой объект и возвращает объектный тип, являющийся типом объекта *obj*. Встроенная функция `isinstance(obj, type)` возвращает значение `True`, если объект *obj* имеет тип *type* (или тип любого его подкласса); в противном случае возвращается значение `False`.

В Python имеются встроенные типы для всех основных типов данных, таких как числа, строки, кортежи, списки, словари и множества, рассмотрению которых посвящены несколько следующих разделов. Кроме того, допускается создавать пользовательские типы, так называемые *классы*, которые обсуждаются в разделе "Классы и экземпляры" главы 4.

Числа

К встроенным числовым типам Python относятся целые числа (`int` и `long` в версии v2; в версии v3 никаких видовых различий между целыми числами не существует), числа с плавающей точкой и комплексные числа. Кроме того, стандартная библиотека предлагает десятичные дроби с фиксированной точностью (раздел "Модуль `decimal`" в главе 15) и рациональные дроби (раздел "Модуль `fractions`" в главе 15). В Python все числа являются неизменяемыми объектами, поэтому, выполняя операцию над числовым объектом, вы получаете новый числовой объект. Операции над числами, известные как арифметические операции, рассматриваются в разделе "Операции над числами".

Числовые литералы не включают знак: ведущий знак + или –, если таковой имеется, является отдельным оператором (раздел "Арифметические операции").

Целые числа

Существуют десятичные, двоичные, восьмеричные и шестнадцатеричные целочисленные литералы. Десятичный литерал — это последовательность цифр, начинающаяся с цифры, отличной от нуля. Двоичный литерал начинается с символов `0b`, за которыми следуют двоичные цифры `0` или `1`. Восьмеричный литерал может (но только в версии v2) начинаться с `0`, за которым следуют восьмеричные цифры от `0` до `7`. Такой синтаксис может сбить с толку того, кто читает код, и потому не рекомендован. Вместо этого лучше использовать другой вариант синтаксиса, в соответствии с которым восьмеричный литерал начинается с символов `0o`, за которыми следуют восьмеричные цифры. Этот вариант работает в обеих версиях, v2 и v3, и не может ввести читателя в заблуждение. Шестнадцатеричный литерал начинается с символов `0x`, за которыми следуют шестнадцатеричные цифры (от `0` до `9` и от `A` до `F` в верхнем или нижнем регистре).

```
1, 23, 3493          # Десятичные целочисленные литералы
0b010101, 0b110010   # Двоичные целочисленные литералы
0o1, 0o27, 0o6645    # Восьмеричные целочисленные литералы
0x1, 0x17, 0xDA5     # Шестнадцатеричные целочисленные литералы
```

Верхняя граница для целочисленных литералов не определена, и они могут принимать сколь угодно большие значения (в версии v2 целые значения, превышающие значение `sys.maxint`, являются экземплярами встроенного типа `long`; в версии v3 это различие не проводится, и тип `int` используется для всех целых чисел).

Числа с плавающей точкой

Литерал числа с плавающей точкой — это последовательность десятичных цифр, которая включает десятичную точку (`.`), суффикс экспоненциального (научного) представления (`e` или `E`, за которым следует необязательный знак `+` или `-` и одна или несколько цифр) или и то и другое. Такой литерал не может начинаться с символа `e` или `E`; первым символом может быть любая цифра или точка (`.`).

```
0., 0.0, .0, 1., 1.0, 1e0, 1.e0, 1.0e0  # Литерал числа с плавающей
                                        # точкой
```

Числа с плавающей точкой в Python соответствуют числам с двойной точностью в C (тип `double`) как в отношении диапазона допустимых значений, так и точности, и на большинстве современных платформ для представления их мантиссы (вместе со знаком) отводятся 53 бита. (Информацию относительно диапазона допустимых значений и точности представления чисел с плавающей точкой на данной платформе предоставляет кортеж `sys.float_info`. Для получения более подробной информации по этому вопросу можете воспользоваться онлайн-документацией: `https://docs.python.org/3.5/library/sys.html#sys.float_info`.)

Комплексные числа

Комплексное число состоит из двух чисел с плавающей точкой, представляющих соответственно его действительную и мнимую части. Доступ к обеим частям комплексного объекта `z` обеспечивают атрибуты `z.real` и `z.imag`, доступные только для чтения. Мнимый литерал можно задать с помощью числа с плавающей точкой или десятичного дробного числа с фиксированной точностью с последующей буквой `j` или `J`:

```
0j, 0.j, 0.0j, .0j, 1j, 1.j, 1.0j, 1e0j, 1.e0j, 1.0e0j
```

Буква `j` в конце литерала обозначает квадратный корень из `-1`, как это принято в электротехнике (в некоторых других дисциплинах для этого используется буква `i`, но разработчики Python остановили свой выбор на букве `j`). Никаких других комплексных литералов не существует. Для записи любого постоянного комплексного числа достаточно сложить (или вычесть) литерал числа с плавающей точкой (или целочисленный литерал) с мнимой частью. Например, комплексное число, равное единице, можно представить как `1+0j` или `1.0+0.0j`. Python выполнит операцию сложения или вычитания во время компиляции.

Новое в версии 3.6: использование символа подчеркивания в числовых литералах

Начиная с версии 3.6, чтобы облегчить визуальную оценку величины чисел, между цифрами и после любого спецификатора системы счисления разрешается вставлять одиночные символы подчеркивания. Как вы, наверное, и сами догадались, этот прием применим не только в отношении десятичных числовых констант:

```
>>> 100_000.000_0001, 0x_FF_FF, 0o7_777, 0b_1010_1010
(100000.0000001, 65535, 4095, 170)
```

Последовательности

Последовательность — это упорядоченная коллекция элементов, индексируемых с помощью целых чисел. В Python имеются встроенные типы последовательностей: строки (байтовые последовательности и последовательности символов Unicode), кортежи и списки. Стандартная библиотека и модули расширения предоставляют другие типы последовательностей, также можно создавать собственные типы последовательностей. Последовательностями можно манипулировать множеством способов (раздел "Операции над последовательностями").

Итерируемые объекты

В языке Python понятие *итерируемый объект*, которое подробно обсуждается в разделах "Инструкция `for`" и "Итераторы", является обобщением идеи последовательности. Все последовательности являются итерируемыми объектами: всякий раз, когда говорится о возможности использования итерируемого объекта, допускается, в частности, использовать последовательность (например, список).

Кроме того, когда мы говорим о возможности использования итерируемого объекта, то, как правило, имеем в виду *ограниченный* итерируемый объект, т.е. объект, элементы которого в конечном счете исчерпываются. Все последовательности являются ограниченными. Вообще говоря, возможны и неограниченные итерируемые объекты, но если вы попытаетесь использовать такой объект, не приняв специальных мер предосторожности, то получите программу, которая никогда не сможет закончиться, разве что это произойдет в результате исчерпания всей доступной памяти.

Строки

Встроенный строковый объект (в виде байтовой строки или строки символов Unicode) — это последовательность символов, используемая для хранения и представления текстовой информации (байтовые строки, или *байтовые объекты*, хранят и представляют последовательности двоичных байтов). Строки в Python — *неизменяемые объекты*: выполняя операцию над строкой, вы всегда получаете новый строковый объект, а не изменяете существующий. Строковые объекты предоставляют множество методов, которые подробно обсуждаются в разделе "Методы строковых и байтовых объектов" главы 8.

Различие между строковыми типами в версиях v2 и v3

В версии v2 недекорированные строковые литералы означают байтовые строки; в версии v3 такие литералы означают строки символов Unicode (текстовые строки).

Строковый литерал может заключаться в кавычки или тройные кавычки. Строка, заключенная в кавычки, — это последовательность, содержащая произвольное количество символов, в том числе ни одного, и заключенная между парой кавычек одного вида: одинарных (') или двойных (").

```
'Это литеральная строка'
"Это еще одна строка"
```

Оба вида кавычек функционируют одинаковым образом. Наличие двух разновидностей кавычек позволяет включать кавычки одного вида в строку, определенную с использованием кавычек другого вида, не прибегая к экранированию кавычек символами обратной косой черты (\).

```
'I\'m a Python fanatic' # кавычку можно экранировать
"I'm a Python fanatic"  # этот вариант строки более удобочитаемый
```

При прочих равных условиях использование одинарных кавычек (апострофов) в большей степени соответствует духу Python. Строковый литерал можно продолжить на нескольких физических строках, завершая текущую строку символом обратной косой черты, указывающей на то, что следующая строка является продолжением литерала.

```
'Не очень длинная строка, которая располагается \
на двух строках'          # использование комментария в предыдущей
                          # строке запрещено
```

Для создания строки, располагающейся на двух строках, можно использовать символ перевода строки.

```
'Не очень длинная строка, которая выводится\n\
на двух строках'          # использование комментария в предыдущей
                          # строке запрещено
```

Лучший подход заключается в использовании строки, заключенной в тройные кавычки (''' или, чаще, """).

```
"""Более длинная строка,
которая располагается на
трех физических строках"""    # использование комментария в
                              # предыдущей строке запрещено
```

Разрывы строк в строковом литерале с тройными кавычками появляются в результирующем строковом объекте в виде символов перевода строки. Чтобы избежать смещения содержимого первой строки такого литерала относительно остальных

строк, начинайте ввод содержимого с символа обратной косой черты, за которым следует перевод строки.

```
the_text = """\
Первая строка
Вторая строка
"""    # то же самое, что 'Первая строка\nВторая строка\n',
       # но в более удобочитаемой форме
```

Единственный символ, который не может быть частью строки, заключенной в тройные кавычки, — это неэкранированный символ обратной косой черты, тогда как строка, заключенная в одинарные или двойные кавычки, не может содержать ни неэкранированных символов обратной косой черты, ни символов перевода строки, ни символа кавычек, совпадающего с ограничивающими ее кавычками. Символ обратной косой черты указывает на начало *управляющей последовательности* (escape sequence), позволяющей ввести любой символ, независимо от разновидности строки. Управляющие последовательности Python приведены в табл. 3.1.

Таблица 3.1. Управляющие последовательности

Последовательность	Назначение	Код ASCII/ISO
\<новая строка>	Игнорировать конец строки	Отсутствует
\\	Обратная косая черта	0x5c
\'	Одиночная кавычка	0x27
\"	Двойная кавычка	0x22
\a	Звуковой сигнал	0x07
\b	Возврат на шаг (забой)	0x08
\f	Прогон страницы	0x0c
\n	Перевод строки	0x0a
\r	Возврат каретки	0x0d
\t	Горизонтальная табуляция	0x09
\v	Вертикальная табуляция	0x0b
DDD	Восьмеричное значение *DDD*	Предоставленное значение
\x *XX*	Шестнадцатеричное значение *XX*	Предоставленное значение
другой_символ	Любой другой символ: двухсимвольная строка	0x5c + предоставленное значение

Разновидностью строковых литералов являются так называемые *сырые* (состоящие из двоичных байтовых значений) строки с отключенным механизмом экранирования символов. Они имеют тот же синтаксис, что и обычные строки, заключенные в кавычки любого вида, но в случае "сырых" строк перед открывающей кавычкой ставится символ `r` или `R`. В "сырых" строках экранированные последовательности,

включая символы обратной косой черты и новой строки, не интерпретируются в соответствии с табл. 3.1, а буквально копируются в строку. Синтаксис "сырых" строк удобно использовать в строках, содержащих большое количество символов обратной косой черты, особенно в строках шаблонов регулярных выражений (раздел "Синтаксис строковых шаблонов" в главы 9). "Сырая" строка не может заканчиваться нечетным количеством символов обратной косой черты: последняя косая черта будет восприниматься так, будто она экранирует закрывающую кавычку.

В строковых литералах Unicode для обозначения символов Unicode можно использовать последовательность `\u`, за которой следуют четыре шестнадцатеричные цифры, и последовательность `\U`, за которой следуют восемь шестнадцатеричных цифр. Такие литералы могут включать экранированные последовательности, приведенные в табл. 3.1. Кроме того, строковые литералы Unicode могут включать экранированные последовательности вида `\N{`*имя*`}`, где *имя* — одно из стандартных имен Unicode, перечень которых доступен по адресу `http://www.unicode.org/charts/`. Например, последовательность `\N{Copyright Sign}` — это обозначение символа Unicode, соответствующего знаку охраны авторского права (©).

В версии v2 "сырые" строковые литералы Unicode начинаются с символов `ur`, а не `ru`; "сырые " байтовые литеральные строки начинаются в версии v2 с символов `br`, а не `rb` (в версии v3 можно использовать как `br`, так и `rb`).

"Сырые" строки — это не другой тип строк

"Сырые" строки не отличаются по своему типу от обычных строк: это всего лишь альтернативный синтаксис для литералов двух обычных строковых типов — байтовых строк и строк Unicode.

Форматируемые строковые литералы, впервые появившиеся ***в версии 3.6***, позволяют внедрять форматирующие выражения в строки, которые после этого перестают быть константами и должны вычисляться во время выполнения. Эти новые литералы подробно обсуждаются в разделе "Новое в версии 3.6: форматированные строковые литералы" главы 8. С точки зрения синтаксиса они могут рассматриваться всего лишь как еще одна разновидность строковых литералов.

Многострочные строковые литералы любого вида — заключенные в одинарные, двойные или тройные кавычки, "сырые", байтовые, форматируемые, литералы Unicode — могут примыкать друг к другу с необязательным пробелом между ними (но в версии v3 таким способом нельзя смешивать байты и символы Unicode). Компилятор конкатенирует соседние строковые литералы в один строковый объект. В версии v2, если хотя бы один из конкатенируемых литералов является литералом Unicode, весь результат преобразуется в Unicode. Запись строкового литерала таким способом позволяет представить его в более удобочитаемом виде на нескольких строках, одновременно предоставляя возможность вставлять комментарии между частями строки.

```
marypop = ('super'              # Скобка не закрыта - логическая
                                # строка продолжается
           'califragilistic')   # При продолжении строки
                                # отступ игнорируется
```

Строковое значение, присвоенное переменной `marypop`, представляет собой одно слово, состоящее из 20 символов.

Кортежи

Кортеж — это неизменяемая упорядоченная последовательность произвольных объектов, типы которых могут различаться. В качестве элементов кортежа можно использовать изменяемые объекты (например, списки), однако в соответствии с установившейся практикой этого лучше избегать.

Кортеж записывается в виде последовательности выражений (*элементов* кортежа), разделенных запятыми (`,`); если каждый из элементов является литералом, то вся их совокупность является *литеральным кортежем*. Вслед за последним элементом может стоять необязательная запятая. Элементы кортежа можно группировать с помощью круглых скобок, однако скобки обязательны только для обозначения пустых или вложенных кортежей, а также в тех случаях, когда их отсутствие могло бы привести к неправильной интерпретации запятых (например, когда кортеж передается функции в качестве аргумента). Кортеж, содержащий ровно два элемента, называется *парой*. Чтобы создать кортеж, состоящий из одного элемента, добавьте запятую в конце выражения. Пустой кортеж обозначается парой круглых скобок. Ниже приведены примеры того, как могут выглядеть литеральные кортежи (круглые скобки обязательны лишь в последнем примере).

```
(100, 200, 300)  # Кортеж, состоящий из трех элементов
(3.14,)          # В кортеже, состоящем из одного элемента,
                 # требуется замыкающая запятая
()               # Пустой кортеж (скобки ОБЯЗАТЕЛЬНЫ)
```

Также существует возможность создания кортежей с помощью встроенной функции `tuple()`.

```
tuple('wow')
```

Этот вызов создает кортеж, который в литеральной записи выглядит так:

```
('w', 'o', 'w')
```

В результате вызова функции `tuple()` без аргументов создается и возвращается пустой кортеж, `()`. Если *x* — итерируемый объект, то функция `tuple(x)` возвращает новый кортеж, который имеет те же элементы, что и *x*.

Списки

Список — это изменяемая упорядоченная последовательность произвольных объектов, типы которых могут различаться. Список записывается в виде последовательности выражений (*элементов* списка), разделенных запятыми (`,`), причем вся

последовательность заключается в квадратные скобки ([]). Если каждый из элементов является литералом, то вся их совокупность является *литеральным списком*. Вслед за последним элементом списка может помещаться необязательная запятая. Пустой список обозначается парой квадратных скобок. Ниже приведены примеры того, как могут выглядеть литеральные списки.

```
[42, 3.14, 'hello']  # Список, состоящий из трех элементов
[100]                # Список, состоящий из одного элемента
[]                   # Пустой список
```

Списки можно также создавать с помощью встроенной функции `list()`.

```
list('wow')
```

Этот вызов создает литеральный список, который в литеральной записи выглядит так:

```
['w', 'o', 'w']
```

В результате вызова функции `list()` без аргументов создается и возвращается пустой список, `[]`. Если *x* — итерируемый объект, то функция `list(x)` создает и возвращает новый список, который имеет те же элементы, что и *x*. Списки можно также создавать с помощью генераторов списков, которые рассматриваются в разделе "Генераторы списков".

Множества

В Python предусмотрены два встроенных типа *множеств*, `set` и `frozenset`, которые предназначены для представления неупорядоченных коллекций уникальных (неповторяющихся) элементов. Элементы, входящие в состав множества, могут иметь разные типы, но они должны быть *хешируемыми* (см. описание функции `hash()` в табл. 7.2). Экземпляры типа `set` — изменяемые и поэтому не являются хешируемыми; экземпляры типа `frozenset` — неизменяемые, а значит, хешируемые. Не может быть изменяемых множеств, элементами которых являются изменяемые множества, но как изменяемые, так и неизменяемые множества могут содержать неизменяемые множества в качестве элементов. Оба указанных типа множеств *не* являются упорядоченными.

Изменяемое множество можно создать, вызвав встроенную функцию `set()` без аргумента (при этом создается пустое множество) или с одним аргументом в виде итерируемого объекта (при этом создается множество с теми же элементами, которые содержатся в итерируемом объекте). Аналогичным образом можно создать неизменяемое множество, вызвав функцию `frozenset()`.

Множество (изменяемое, непустое) записывается в виде последовательности выражений (элементов множества), разделенных запятыми (`,`), и заключается в фигурные скобки (`{}`); если каждый из элементов является литералом, то вся их совокупность является *литеральным множеством*. Вслед за последним элементом множества

может помещаться необязательная запятая. Ниже приведены примеры того, как могут выглядеть множества (два литеральных и одно, не являющееся таковым).

```
{42, 3.14, 'hello'}  # Литерал множества с тремя элементами
{100}                # Литерал множества с одним элементом
set()                # Пустое множество; для его создания нельзя
                     # использовать выражение {}
```

Изменяемые множества можно создавать с помощью аналогов генераторов списков — генераторов множеств (раздел "Генераторы списков").

Словари

Отображение — это произвольная коллекция объектов, индексируемых с помощью почти[3] произвольных значений — *ключей*. Как и множества, отображения являются изменяемыми и, в отличие от последовательностей, *не* обязательно упорядоченными.

Python предоставляет один встроенный тип отображений: *словарь*. Библиотеки и модули расширений предоставляют другие типы отображений; их также можно создавать самостоятельно (раздел "Отображения" в главе 4). Ключи словаря могут иметь различные типы, но они должны быть *хешируемыми* (см. описание функции `hash()` в табл. 7.2). Значениями словаря могут быть произвольные объекты любого типа. *Элементами* словаря являются пары "ключ–значение". Словари аналогичны ассоциативным массивам (в других языках для них используются термины "неупорядоченная карта", "хеш-таблица", "хеш" и др.).

Словарь записывается в виде последовательности пар выражений, разделенных двоеточиями (эти пары являются элементами словаря), которые, в свою очередь, разделяются запятыми (`,`), причем вся их совокупность заключается в фигурные скобки (`{}`); если каждое из выражений является литералом, то вся совокупность является *литеральным словарем*. Вслед за последним элементом может помещаться необязательная запятая. Каждый элемент словаря записывается в виде *ключ*`:` *значение*, где *ключ* — выражение, представляющее ключ элемента, а *значение* — выражение, представляющее значение элемента. Если значение ключа встречается в выражениях более одного раза, то в результирующий объект попадает лишь один из элементов с данным значением ключа, выбираемый произвольно, — дублирование ключей в словарях не допускается. Пустой словарь записывается в виде пустой пары фигурных скобок.

Вот примеры того, как могут выглядеть литералы словарей.

```
{'x':42, 'y':3.14, 'z':7}  # Словарь, состоящий из трех элементов,
                           # тип ключей - str
{1:2, 3:4}                 # Словарь, состоящий из двух элементов,
                           # тип ключей - int
```

[3] Каждый конкретный тип отображения может налагать ограничения на тип допустимых ключей: в частности, словари допускают лишь хешируемые ключи.

```
{1:'za', 'br':23}          # Словарь с ключами различных типов
{}                         # Пустой словарь
```

Также существует возможность создания словарей с помощью встроенной функции `dict()`. И хотя этот способ уступает в лаконичности предыдущему, иногда он улучшает удобочитаемость кода. Приведенные выше примеры можно переписать в следующей эквивалентной форме.

```
dict(x=42, y=3.14, z=7)    # Словарь, состоящий из трех элементов,
                           # тип ключей - str
dict([(1, 2), (3, 4)])     # Словарь, состоящий из двух элементов,
                           # тип ключей - int
dict([(1,'za'), ('br',23)])# Словарь с ключами различных типов
dict()                     # Пустой словарь
```

В результате вызова функции `dict()` без аргументов создается и возвращается пустой словарь, `{}`. Если аргумент *x*, передаваемый функции `dict()`, является отображением, то она возвращает новый объект словаря с теми же ключами и значениями, что и в *x*. Если *x* — итерируемый объект, то его элементами должны быть пары, и в этом случае функция `dict(x)` возвращает словарь с теми же элементами (пары "ключ–значение"), что и в *x*. Если значение ключа встречается в *x* более одного раза, то в результирующий словарь попадает лишь *последний* из элементов *x* с данным значением ключа.

Вызывая функцию `dict()`, можно указать дополнительно к позиционному аргументу *x* или вместо него *именованные аргументы* вида *имя=значение*, где *имя* — идентификатор, используемый в качестве ключа элемента, а *значение* — выражение, представляющее значение элемента. Если при вызове функции `dict()` с передачей ей как позиционного, так и одного или нескольких именованных аргументов какой-либо ключ встречается одновременно в виде позиционного и именованного аргументов, то Python ассоциирует этот ключ со значением, предоставляемым именованным аргументом (т.е. именованный аргумент имеет более высокий приоритет).

Также возможно создание словаря путем вызова функции `dict.fromkeys()`. Ее первым аргументом является итерируемый объект, элементы которого становятся ключами словаря; второй аргумент — это значение, которое соответствует всем без исключения ключам (первоначально все ключи отображаются в одно и то же значение). Опущенный второй аргумент по умолчанию имеет значение `None`.

```
dict.fromkeys('hello', 2) # то же самое, что
                          # {'h':2, 'e':2, 'l':2, 'o':2}
dict.fromkeys([1, 2, 3])  # то же самое, что
                          # {1:None, 2:None, 3:None}
```

Кроме того, словари можно создавать с помощью генераторов словарей (раздел "Генераторы списков").

Объект None

Встроенный объект `None` обозначает нулевой объект. Он не имеет методов и других атрибутов. Объект `None` можно использовать в качестве заместителя в тех случаях, когда требуется ссылка, но вам безразлично, на какой объект она указывает, или когда требуется указать на отсутствие какого-либо объекта. Если функция не содержит инструкций `return`, возвращающих другие значения, то она возвращает объект `None`.

Вызываемые типы

В Python *вызываемые типы* — это типы, экземпляры которых поддерживают операцию вызова функции (раздел "Вызов функций" в главе 4). Функции — вызываемый тип. Python предоставляет ряд встроенных функций (раздел "Встроенные функции" в главе 7) и поддерживает функции, определяемые пользователем (раздел "Инструкция `def`"). Вызываемым типом являются также генераторы (раздел "Генераторы").

Типы также являются вызываемыми объектами, в чем вы имели возможность убедиться на примере встроенных типов `dict`, `list`, `set` и `tuple`. (Полный перечень встроенных типов приведен в разделе "Встроенные типы" главы 7.) В соответствии с приведенным в разделе "Классы Python" главы 4 рассмотрением объекты `class` (типы, определяемые пользователем) также являются вызываемыми. Обычно в результате вызова типа создается и возвращается новый экземпляр этого типа.

К вызываемым объектам относятся также *методы*, которые представляют собой функции, связанные с атрибутами класса, и экземпляры классов, предоставляющие специальный метод `__call__`.

Булевы значения

В Python любое значение может служить показателем логической истинности или ложности: `True` или `False`. Любое ненулевое значение или непустой контейнер (например, строка, кортеж, список, множество или словарь) является истинным значением (эквивалентными значению `True`). `0` (любого числового типа), `None` и пустые контейнеры являются ложными значениями (эквивалентными значению `False`).

Остерегайтесь использовать числа с плавающей точкой в качестве эквивалентов логических значений

Будьте предельно внимательны, когда используете числа с плавающей точкой в качестве истинных значений: это все равно что проверять число на точное равенство нулю, в то время как числа с плавающей точкой почти никогда нельзя использовать для точного сравнения.

Встроенный тип `bool` — это подкласс типа `int`. Он может принимать только два значения — `True` или `False`, которые имеют строковые представления `'True'` и `'False'`, но в действительности являются значениями `1` и `0` соответственно. Булевы значения возвращаются операторами сравнения, а также рядом встроенных функций.

Функция `bool(`*x*`)` может принимать любой аргумент *x*. Она возвращает значение `True`, если `x` представляет истинное значение, и `False`, если *x* представляет ложное значение. Хороший стиль программирования на Python требует избегать таких вызовов, если в этом нет особой необходимости, как чаще всего и бывает: *всегда* используйте только конструкцию `if` *x*`:` и *никогда* не используйте конструкции `if bool(`*x*`):`, `if` *x* `is True`, `if` *x*`==True:`, `if bool(`*x*`)==True`. В то же время можно использовать вызовы `bool(`*x*`)` для подсчета количества истинных элементов в последовательности, например:

```
def count_trues(seq): return sum(bool(x) for x in seq)
```

В этом примере вызов функции `bool` гарантирует, что каждый элемент последовательности `seq` будет подсчитываться со значением `0` (если он ложный) или со значением `1` (если он истинный), поэтому функция `count_trues(seq)` более универсальна по сравнению с функцией `sum(seq)`.

Когда мы говорим, что "выражение является истинным", это означает, что вызов `bool(`*выражение*`)` возвращает значение `True`.

Переменные и другие ссылки

Программа на языке Python получает доступ к значениям данных посредством ссылок. *Ссылка* — это "имя", которое позволяет обращаться к значению (объекту). Ссылки принимают форму переменных, атрибутов и элементов. В Python переменные и другие ссылки не имеют внутреннего типа. Объект, с которым в данный момент связана ссылка, всегда имеет тип, но любая ссылка в процессе выполнения программы может быть связана с объектами разных типов.

Переменные

В Python отсутствуют "объявления" переменных. Существование переменной начинается с того момента, когда впервые встречается инструкция, которая *связывает* имя переменной с некоторым объектом (иными словами, когда переменная используется для сохранения ссылки на объект). Вы также можете *освободить* переменную, т.е. разорвать ее связь с объектом таким образом, чтобы в ней больше не хранилась никакая ссылка. Самым распространенным способом связывания переменных и других ссылок является инструкция присваивания. Инструкция `del` освобождает переменную, т.е. открепляет ее от объекта.

Связывание ссылки, которая до этого уже была связана, называется *повторным связыванием*. Всякий раз, когда речь идет о связывании переменных, мы неявно

подразумеваем также возможность повторного связывания (за исключением тех случаев, когда явно оговаривается иное). Повторное связывание переменной с другим объектом или разрыв ее связи с объектом не оказывают никакого влияния на сам объект, за исключением того, что объект, ссылки на который отсутствуют, удаляется. Освобождение ресурсов, используемых объектом, на который не указывает ни одна ссылка, называют *сборкой мусора*.

В качестве имени переменной можно использовать любой идентификатор, кроме тех (их свыше тридцати), которые зарезервированы в Python в качестве ключевых слов (см. раздел "Ключевые слова"). Существуют глобальные и локальные переменные. *Глобальная переменная* — это атрибут объекта модуля (глава 6). *Локальная переменная* существует в локальном пространстве имен функции (см. раздел "Пространства имен").

Атрибуты и элементы объектов

Различие между атрибутами и элементами в основном обусловлено синтаксисом доступа к ним. Для обозначения *атрибута* объекта используется ссылка на объект, за которой через точку (`.`) указывается идентификатор — *имя атрибута*. Например, выражение `x.y` ссылается на один из атрибутов объекта, связанных с именем `x`, а именно — на атрибут `'y'`.

Для обозначения *элемента* объекта используется ссылка на объект, за которой следует выражение, заключенное в квадратные скобки (`[]`). Выражение в скобках — это *индекс* или *ключ* элемента, а объект — это *контейнер*. Например, выражение `x[y]` ссылается на элемент, ключ или индекс которого связан с именем `y` и который содержится в объекте-контейнере, связанном с именем `x`.

Атрибуты, являющиеся вызываемыми объектами, называются *методами*. Python не проводит различий между вызываемыми и невызываемыми атрибутами, как это делается в некоторых других языках программирования. Все правила, касающиеся атрибутов, применимы также к вызываемым атрибутам (методам).

Попытки доступа по несуществующим ссылкам

Распространенной ошибкой в программировании является попытка использования несуществующей ссылки для доступа к объекту. Причиной возникновения ошибки может стать переменная, не связанная ни с каким объектом, или же отсутствие у объекта атрибута с указанным именем или элемента с указанным индексом. В процессе компиляции и анализа исходного кода компилятор Python проверяет лишь синтаксис. Семантические ошибки, такие как попытка доступа к атрибуту, элементу или переменной, не имеющих связывания, в процессе компиляции не диагностируются. Python может обнаружить семантические ошибки лишь *во время выполнения*. Если выполняемая операция содержит семантическую ошибку, Python генерирует исключение (см. главу 5). При попытке доступа к несуществующей переменной, атрибуту или элементу, как и при любой другой семантической ошибке, генерируется исключение.

Операции присваивания

В Python операции присваивания бывают двух видов: простые и составные. Простое присваивание значения переменной (например, *имя=значение*) — это именно тот способ, с помощью которого вы создаете новую переменную или повторно связываете существующую переменную с другим значением. Простое присваивание значения атрибуту объекта (например, *x.атрибут=значение*) — это запрос к объекту *x* создать или повторно связать атрибут *'атрибут'*. Простое присваивание значения элементу контейнера (например, *x[k]=значение*) — это запрос к контейнеру *x* создать или повторно связать элемент с индексом или ключом *k*.

Составное присваивание (например, *имя+=значение*) не может само по себе создать новую ссылку. Такое присваивание может лишь повторно связать переменную, а также попросить объект повторно связать один из его существующих атрибутов или элементов или изменить самого себя. Когда вы обращаетесь к объекту, он самостоятельно решает, можно ли удовлетворить ваш запрос и как именно или же следует сгенерировать исключение.

Простое присваивание

В своей наиболее элементарной форме операция простого присваивания имеет следующий синтаксис:

```
цель = выражение
```

Целевая ссылка (*цель*), указанная слева от знака равенства, называется левой частью (lefthand side, LHS) инструкции присваивания, тогда как *выражение* справа — ее правой частью (righthand side, RHS). Когда выполняется операция присваивания, Python вычисляет выражение RHS, а затем связывает полученное значение с целевой ссылкой LHS. Это связывание не зависит от типа значения. В частности, Python не проводит строгого различия между вызываемыми и невызываемыми объектами, как это принято делать в некоторых языках программирования, поэтому допускается связывать с переменными функции, методы, типы и другие вызываемые объекты точно так же, как числа, строки, списки и т.п. В частности, это обусловлено тем, что функции и им подобные объекты являются *объектами первого класса*.

Однако детали связывания зависят от вида целевой ссылки. В качестве целевой ссылки в операции присвоения может выступать идентификатор, ссылка на атрибут, а также ссылка на индексированный элемент или срез.

Идентификатор
: Это имя переменной. Присваивание значения идентификатору связывает переменную с этим именем.

Ссылка на атрибут
: Записывается с использованием точечной нотации вида *объект.имя*, где *объект* — произвольное выражение, а *имя* — идентификатор, который служит

именем атрибута объекта. Присваивание значения ссылке на атрибут запрашивает у объекта связывание атрибута, указанного с помощью имени `'имя'`.

Индексированный элемент

Записывается с использованием скобочной нотации в виде `объект[выражение]`, где `объект` и `выражение` — произвольные выражения. Присваивание значения по индексу запрашивает у контейнера `объект` связывание его элемента, указанного с помощью значения `выражения`, которое называют *индексом* или *ключом* элемента контейнера.

Срез

Записывается в виде `объект[начало:конец]` или `объект[начало:конец:шаг]`, где `объект`, `начало`, `конец` и `шаг` — произвольные выражения. Каждое из выражений `начало`, `конец` и `шаг` является необязательным (например, выражения `объект[:конец:]` и `объект[:конец]` синтаксически корректны и эквивалентны выражению `объект[None:конец:None]`). Присваивание значения срезу запрашивает у контейнера `объект` связывание или открепление некоторых из его элементов. Присваивание значения срезу вида `объект[начало:конец:шаг]` эквивалентно присваиванию значения с помощью индексирования вида `объект[slice(начало, конец, шаг)]`. Информация о встроенном типе Python `slice`, экземпляры которого представляют срезы, приведена в табл. 7.1.

К обсуждению индексирования и извлечения срезов мы еще вернемся, когда будем рассматривать операции, выполняемые над списками (раздел "Изменение списка") и словарями (раздел "Индексирование словаря").

Если целевой ссылкой операции присваивания является идентификатор, то оператор присваивания просто связывает значение с именем переменной. Эта операция *никогда* не отклоняется: если вы ее запрашиваете, она выполняется. Во всех остальных случаях инструкция присваивания определяет запрос к объекту на связывание одного или нескольких атрибутов или элементов объекта. Объект может отказать в связывании или повторном связывании некоторых (или всех) атрибутов или элементов, генерируя исключение, если данная операция запрещена (см. также описания функции `__setattr__` в табл. 4.1 и функции `__setitem__` в разделе "Методы контейнеров" главы 4).

Простое присваивание допускает использование цепочек, образованных целевыми ссылками и знаками равенства (=). Например, следующая инструкция выполняет *групповое присваивание*, связывая переменные `a`, `b` и `c` с одним и тем же значением `0`:

```
a = b = c = 0
```

При выполнении группового присваивания выражение, находящееся в правой части (RHS), вычисляется только один раз, независимо от количества целевых ссылок, указанных в инструкции. Каждая из целевых ссылок, в порядке следования слева направо, связывается с единственным объектом, возвращенным выражением RHS, как если бы несколько операций присваивания выполнялись поочередно.

Целевой ссылкой в операции простого присваивания может служить список, состоящий из двух и более ссылок, разделенных запятыми, которые могут заключаться в необязательные круглые или квадратные скобки.

```
a, b, c = x
```

Эта инструкция, в которой требуется, чтобы объект `x` был итерируемым и содержал ровно три элемента, связывает `a` с первым элементом, `b` — со вторым и `c` — с третьим. Операции такого рода носят название *присваивание с распаковкой*. Выражение в правой части должно быть итерируемым и содержать ровно столько элементов, сколько указано ссылок в левой части; в противном случае Python генерирует исключение. Каждая из ссылок, указанных в левой части, связывается с соответствующим элементом из правой части. Присваивание с распаковкой можно использовать для обмена ссылками:

```
a, b = b, a
```

Данная инструкция присваивания повторно связывает имя `a` со значением, с которым было связано имя `b`, и наоборот. В версии v3, если имеется несколько целевых ссылок, разрешается помечать одну из них символом "звездочка" (`*`). Целевая ссылка, которой предшествует этот символ, связывается со списком всех элементов (если таковые имеются), оставшихся не связанными с другими целевыми ссылками. Например, в версии v3 инструкция

```
first, *middle, last = x
```

в которой *x* является списком, эквивалентна (но более лаконична, понятна, обладает большей общностью и выполняется быстрее) следующей инструкции:

```
first, middle, last = x[0], x[1:-1], x[-1]
```

В обеих инструкциях присваивания требуется, чтобы объект `x` имел по крайней мере два элемента. Вторая форма, совместимая с версией v2, требует, чтобы объект `x` был последовательностью, допускающей обращение к ее элементам по индексу; первая форма, допустимая только в версии v3, годится для любого итерируемого объекта `x`, содержащего по крайней мере два элемента. Эту форму операции присваивания, использовать которую можно только в версии v3, называют *расширенной распаковкой*.

Составное присваивание

Операция *составного присваивания* (другое название — *присваивание на месте*) отличается от простого присваивания тем, что вместо знака равенства (`=`), находящегося между целевой ссылкой и выражением, в ней используется составной оператор, представляющий собой комбинацию бинарного оператора и оператора `=`. Составными являются операторы `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `|=`, `>>=`, `<<=`, `&=` и `^=` (а также, только в версии v3, `@=`). В левой части составного присваивания допускается только одна целевая ссылка; множественные целевые ссылки не поддерживаются.

Как и в случае простого присваивания, при выполнении составного присваивания сначала вычисляется RHS-выражение. Затем, если LHS ссылается на объект, в котором предусмотрен специальный метод для выполнения соответствующей бинарной операции на месте, Python вызывает этот метод, передавая ему RHS в качестве аргумента. Именно этот метод отвечает за изменение объекта LHS и возврат измененного объекта (специальные методы рассматриваются в разделе "Специальные методы" главы 4). Если же объект LHS не имеет специального метода для выполнения операции на месте, то Python применяет соответствующий бинарный оператор по отношению к объектам LHS и RHS, а затем связывает целевую ссылку с результатом, возвращаемым этим оператором. Например, инструкция `x+=y` выполняется как инструкция `x=x.__iadd__(y)`, если для `x` предусмотрен специальный метод `__iadd__`, предназначенный для выполнения операции сложения на месте. Если же такой метод не предусмотрен, то инструкция `x+=y` выполняется как инструкция `x=x+y`.

Операция составного присваивания никогда не создает свою целевую ссылку; к моменту ее выполнения целевая ссылка должна существовать. Составное присваивание может повторно связать целевую ссылку с другим объектом или изменить тот объект, с которым она уже была связана. В отличие от этого, простое присваивание может создавать целевую ссылку LHS или повторно связывать ее, но никогда не изменяет объект, с которым она до этого была связана, если таковой имеется. В данном случае различие между объектами и ссылками на объекты играет ключевую роль. Например, операция `x=x+y` не изменяет объект, с которым первоначально было связано имя `x`. Вместо этого она изменяет связывание имени `x` таким образом, чтобы оно ссылалось на новый объект. В то же время операция `x+=y` изменяет объект, с которым связано имя `x`, если этот объект имеет специальный метод `__iadd__`; в противном случае операция `x+=y` выполняется как операция `x=x+y`, повторно связывая имя `x` с новым объектом.

Инструкции `del`

Несмотря на свое название, инструкция `del`[4] *не* удаляет объекты в буквальном смысле, она лишь *открепляет ссылки*, т.е. освобождает их, разрывая связь между именем и объектом, с которым ассоциировано данное имя. Удаление объекта может происходить лишь как следствие этого и автоматически выполняется *сборщиком мусора* в отношении объектов, на которые больше не ссылается ни один другой объект.

Инструкция `del` состоит из ключевого слова `del`, за которым следует одна или несколько целевых ссылок, разделенных запятыми (`,`). Каждая целевая ссылка может быть переменной, ссылкой на атрибут, индексированным элементом или срезом, как в инструкциях присваивания, и к моменту выполнения инструкции `del` должна быть связана с каким-либо объектом. Если целевая ссылка является идентификатором, выполнение инструкции `del` означает разрыв связи между объектом и закрепленной

[4] От англ. *delete* (удалить). — *Примеч. ред.*

за ним переменной. Если к моменту выполнения инструкции `del` идентификатор был связан с каким-либо объектом, то откреплению идентификатора от объекта ничто не может помешать: если эта операция запрашивается, она всегда выполняется.

Во всех остальных случаях инструкция `del` определяет запрос к объекту на открепление одного или нескольких его атрибутов или элементов. Объект может отказать в выполнении этой операции по отношению к некоторым (или всем) атрибутам или элементам, генерируя исключение при попытке отмены связывания в тех случаях, когда это запрещено (см. также описание функций `__delattr__` в разделе "Универсальные специальные методы" главы 4 и `__delitem__` в разделе "Методы контейнеров" главы 4). Обычно отмена связывания среза приводит к тому же результату, что и присваивание пустой последовательности данному срезу, но ответственность за реализацию подобной эквивалентности возлагается на объект-контейнер.

Контейнерам разрешено сопровождать выполнение инструкции `del` побочными эффектами. Например, если *C* — словарь и инструкция `del C[2]` успешно выполнена, то это означает, что последующие обращения к элементу `C[2]` будут недопустимыми (приведут к возбуждению исключения `KeyError`) до тех пор, пока вы вновь не свяжете элемент `C[2]` (не присвоите ему некоторое значение). Но если *C* — список, то в результате выполнения инструкции `del C[2]` каждый последующий элемент *C* сместится на одну позицию влево, и если список *C* достаточно длинный, то последующие обращения к `C[2]` будут по-прежнему действительными, хотя и могут давать другое значение, отличное от того, которое возвращалось до выполнения инструкции `del`.

Выражения и операторы

Выражение — это "фраза" кода, которую Python вычисляет для получения значения. Простейшими выражениями являются литералы и идентификаторы. Другие выражения строятся посредством объединения подвыражений с помощью операторов, выполняющих соответствующие операции, и/или разделителей, перечисленных в табл. 3.2. В этой таблице операции перечислены в порядке уменьшения приоритета: операции с более высоким приоритетом выполняются до выполнения операций с более низким приоритетом. Операции, указанные в одной строке таблицы, имеют одинаковый приоритет. В третьем столбце указана *ассоциативность* операторов: L (слева направо), R (справа налево) или NA (неассоциативный).

Таблица 3.2. Приоритет операций в выражениях

Операция	Описание	Ассоциативность
`{` *ключ* `:` *выражение* `,...}`	Создание словаря	NA
`{` *выражение* `,...}`	Создание множества	NA
`[` *выражение* `,...]`	Создание списка	NA
`(` *выражение* `,...)`	Создание кортежа или просто скобки	NA

Операция	Описание	Ассоциативность
`f(` *выражение* `,...)`	Вызов функции	L
`x[` *индекс*`:`*индекс* `]`	Выделение среза	L
`x[`*индекс*`]`	Взятие элемента по индексу	L
`x.`*атрибут*	Ссылка на атрибут	L
`x**y`	Возведение в степень (*x* в степени *y*)	R
`~x`	Побитовое НЕ	NA
`+x, -x`	Унарные "плюс" и "минус"	NA
`x*y, x/y, x//y, x%y`	Умножение, деление, деление с усечением до целого, остаток от деления	L
`x+y, x-y`	Сложение, вычитание	L
`x<<y, x>>y`	Сдвиг влево, сдвиг вправо	L
`x&y`	Побитовое И	L
`x^y`	Побитовое исключающее ИЛИ	L
`x|y`	Побитовое ИЛИ	L
`x<y, x<=y, x>y, x>=y, x<>y` (только в версии v2), `x!=y, x==y`	Сравнения (меньше чем, меньше или равно, больше чем, больше или равно, неравенство, равенство)*	NA
`x is y, x is not y`	Проверки тождественности	NA
`x in y, x not in y`	Проверки принадлежности	NA
`not x`	Логическое НЕ	NA
`x and y`	Логическое И	L
`x or y`	Логическое ИЛИ	L
`x if` *выражение* `else y`	Тернарный оператор	NA
`lambda` *аргумент*`,...:`*выражение*	Создание анонимной простой функции	NA

* В версии v2 операторы <> и != — это две альтернативные формы одного и того же оператора. Вариант != является более предпочтительным; вариант <> выходит из употребления и не поддерживается в версии v3.

В табл. 3.2 *выражение*, *ключ*, *f*, *индекс*, *x* и *y* обозначают любое выражение, тогда как *атрибут* и *аргумент* обозначают любой идентификатор. Символы `,...` означают запятые, присоединяющие нуль или более повторений предыдущего компонента. Во всех подобных случаях замыкающая запятая является необязательной, и ее наличие ни на что не влияет.

Цепочки сравнений

Сравнения можно выстраивать в цепочки, в которых для соединения сравнений используются неявные логические операторы `and`. Например, выражение

```
a < b <= c < d
```

означает то же самое, что и выражение

```
a < b and b <= c and c < d
```

В такой форме сравнения легче читаются, и каждое подвыражение вычисляется по крайней мере один раз[5].

Закорачивание операторов

Операторы `and` и `or` *закорачивают* вычисление своих операндов: правый операнд вычисляется лишь в том случае, если его значение необходимо для получения истинного значения операции `and` или `or`.

Другими словами, в выражении *x* `and` *y* сначала вычисляется *x*. Если *x* имеет ложное значение , то результат равен *x*; в противном случае результат равен *y*. Аналогичным образом в выражении *x* `or` *y* сначала вычисляется *x*. Если *x* имеет истинное значение, результат равен *x*; в противном случае результат равен *y*.

Операторы `and` и `or` не приводят свои результаты принудительно к значениям `True` или `False`, а возвращают один из своих операндов. Такой подход позволяет использовать эти операторы в более общих, а не только булевых контекстах. Если другие операторы, прежде чем выполнить операцию, вычисляют все свои операнды, то в случае операторов `and` и `or` с их семантикой закорачивания необходимость вычисления правого операнда определяется результатом вычисления левого. Операторы `and` и `or` позволяют левому операнду выполнять функции своеобразного *стража* по отношению к правому операнду.

Тернарный оператор

Еще одним оператором, способным закорачивать вычисления операндов, является тернарный оператор `if/else`:

если_истинно `if` *условие* `else` *если_ложно*

Здесь *если_истинно*, *если_ложно* и *условие* — произвольные выражения. Выражение *условие* вычисляется первым. Если условие истинно, результат равен значению *если_истинно*; в противном случае результат равен значению *если_ложно*. Из двух выражений *если_истинно* и *если_ложно* вычисляется только одно, в зависимости от истинности условия.

---

[5] Обратите внимание на тот не совсем очевидный факт, что истинность выражения *a!=b!=c*, равно как и истинность эквивалентного ему более длинного выражения *a!=b and b!=c*, не есть неявным свидетельством того, что выражение *a!=c* также является истинным.

Порядок следования подвыражений в тернарном операторе может несколько сбивать с толку. Чтобы этого избежать, рекомендуется всегда заключать все выражение в круглые скобки.

Операции над числами

Python поддерживает все обычные операции, выполняемые над числами. Числа — неизменяемые объекты: выполняя операции над числовыми объектами, вы всегда получаете новые числовые объекты, а не изменяете существующие. Разрешается обращаться к мнимой и действительной частям комплексного объекта `z` как к атрибутам `z.real` и `z.imag`, доступным только для чтения. Попытка присвоить этим атрибутам новые значения приведет к возбуждению исключения.

Необязательные знаки + и - перед числами, а также знак +, присоединяющий литерал с плавающей точкой к мнимому для образования комплексного числа, не являются частью синтаксиса литералов. Они являются обычными операторами, которые подчиняются обычным правилам, регламентирующим приоритет операторов (см. табл. 3.2). Например, вычисление выражения `-2**2` дает результат `-4`: возведение в степень имеет более высокий приоритет, чем унарный "минус", поэтому анализатор воспринимает данное выражение как `-(2**2)`, а не как `(-2)**2`.

Преобразование числовых типов

В арифметических операциях и операциях сравнения могут принимать участие любые два числа, относящиеся к встроенным типам Python. Если типы операндов различаются, выполняется автоматическое *повышение типов*: Python преобразует операнд "меньшего" размера к операнду "большего" размера. Числовые типы имеют следующий порядок следования "размеров" от наименьшего к наибольшему: целые, с плавающей точкой и комплексные. Можно затребовать принудительное преобразование типов, передав аргумент, не являющийся комплексным числом, любому из встроенных числовых типов: `int`, `float` и `complex`. Тип `int` отбрасывает дробную часть своего аргумента, если таковая имеется (например, `int(9.8)` равно `9`). Кроме того, можно вызвать тип `complex` с двумя числовыми аргументами, представляющими действительную и мнимую части. Преобразовать таким способом комплексное число в другой числовой тип невозможно, поскольку однозначных способов преобразования комплексного числа, скажем, в число с плавающей точкой просто не существует.

Кроме того, любой встроенный числовой тип можно вызвать с передачей ему строкового аргумента, в котором используется синтаксис соответствующего числового литерала с дополнительными элементами: строка аргумента может содержать ведущий и/или замыкающий пробелы, начинаться со знака "плюс" или "минус" либо — в случае комплексных чисел — выполнять сложение или вычитание действительной и мнимой частей. Тип `int` также может быть вызван с двумя аргументами:

первый из них — это преобразуемая строка, второй — основание системы счисления, используемой в преобразуемой строке, в виде целого числа от 2 до 36 (например, `int('101', 2)` возвращает 5, т.е. значение `'101'` в системе счисления с основанием 2). Для оснований систем счисления, значение которых превышает 10, недостающие дополнительные "цифры" берутся из подмножества букв начальной части алфавита (как в нижнем, так и в верхнем регистре).

Арифметические операции

Поведение арифметических операций в Python довольно очевидно, если не считать некоторых особых случаев, касающихся деления и возведения в степень.

Деление

Если правый операнд операторов `/`, `//` или `%` равен `0`, то Python генерирует исключение времени выполнения. Оператор `//` выполняет деление с усечением до целого и возвращает результат в виде целого числа (преобразованного к типу более широкого операнда), игнорируя дробный остаток, если таковой имеется.

Деление с усечением до целого, обеспечиваемое оператором `//`, выполняется одинаково в обеих версиях, v2 и v3, возвращая результат в виде числа с плавающей точкой, если хотя бы один из операндов является числом с плавающей точкой, и в виде целого числа, если оба операнда целые числа.

В то же время в тех случаях, когда оба операнда оператора `/` являются целыми числами, он ведет себя по-разному в версиях v3 и v2. В версии v3 он выполняет истинное деление, возвращая результат в виде числа с плавающей точкой (или комплексного, если один из операндов комплексный). Однако в версии v2 он работает аналогично оператору `//`, отбрасывая дробную часть. Если вы хотите, чтобы поведение оператора `/` в версии v2 было таким же, как в версии v3, используйте параметр `-Qnew` в командной строке Python (не рекомендуется) или, что лучше, поместите в начале файла с исходным кодом следующую инструкцию (или начните с нее интерактивный сеанс):

```
from __future__ import division
```

Тем самым будет гарантировано, что в пределах модуля, начинающегося с этой инструкции, оператор `/` будет работать без усечения до целого при операндах любого типа.

Чтобы операция деления с усечением до целого выполнялась одинаково, независимо от используемой версии Python, всегда используйте оператор `//`. Для выполнения истинного деления (без усечения до целого) используйте оператор `/`, но при этом убеждайтесь в том, что хотя бы один из операндов не относится к целочисленному типу. Например, вместо выражения *a*`/`*b* используйте выражение `1.0*`*a*`/`*b* (вариант `float(`*a*`)/`*b* работает несколько медленнее и неприменим, если *a* — комплексное число), что избавит вас от необходимости делать предположения относительно типов *a* и *b*. Чтобы проверить наличие версионных зависимостей операций деления, выполняемых

в вашей программе на основе v2, используйте параметр `-Qwarn` в командной строке Python, который обеспечит вывод предупреждающих сообщений, касающихся использования оператора / с целочисленными операндами, во время выполнения.

Встроенная функция `divmod` принимает два числовых аргумента и возвращает пару, элементами которой являются частное от деления и остаток, тем самым избавляя вас от необходимости использовать операторы `//` и `%` по отдельности для получения того же результата.

Возведение в степень

В версии v2 операция возведения в степень, *a*`**`*b*, генерирует исключение, если *a* меньше нуля, а *b* — число с плавающей точкой и ненулевой дробной частью, однако в версии v3 она возвращает в подобных случаях соответствующее комплексное число.

Встроенная функция `pow(`*a*`, `*b*`)` возвращает тот же результат, что и выражение *a*`**`*b*. Вызов этой функции с тремя аргументами, `pow(`*a*`, `*b*`, `*c*`)`, возвращает тот же результат, что и выражение `(`*a*`**`*b*`)%`*c*, но работает быстрее.

Сравнения

Все объекты, включая числа, можно сравнивать между собой для проверки их равенства (==) или неравенства (!=). В сравнениях, требующих соблюдения порядка следования операндов (<, <=, >, >=), могут участвовать два числа, но в тех случаях, когда одно из них комплексное, генерируется исключение времени выполнения. Каждый из перечисленных операторов возвращает булево значение (`True` или `False`). Остерегайтесь проверки на равенство при сравнении двух чисел с плавающей точкой; о причинах этого можно прочитать в онлайновом руководстве (`https://docs.python.org/3.5/tutorial/floatingpoint.html`).

Битовые операции над целыми числами

Целые числа можно интерпретировать как битовые строки и использовать в битовых операциях (см. табл. 3.2). Битовые операторы имеют более низкий приоритет по сравнению с арифметическими. Положительные целые числа концептуально расширяются влево неограниченной строкой битов 0. Отрицательные числа, для представления которых используется дополнительный код (`https://ru.wikipedia.org/wiki/Дополнительный_код`), концептуально расширяются влево неограниченной строкой битов 1.

Операции над последовательностями

Python поддерживает ряд операций, применимых ко всем последовательностям, включая строки, списки и кортежи. Одни из этих операций применимы ко всем контейнерам (включая множества и словари, которые не являются последовательностями); другие применимы ко всем итерируемым типам. Термин "итерируемый

объект" означает любой объект, допускающий циклический перебор элементов (см. раздел "Итерируемые объекты"). Все контейнеры, независимо от того, являются ли они последовательностями или не являются, — итерируемые объекты, как и ряд других объектов, не являющихся контейнерами, таких как файлы (раздел "Модуль `io`" в главе 10) и генераторы (раздел "Генераторы"). Далее в тех случаях, когда будет требоваться уточнение вида объектов, к которым применимы операции определенной категории, мы будем использовать конкретные термины: *последовательность*, *контейнер* и *итерируемый объект*.

Последовательности в целом

Последовательность — это упорядоченный контейнер с элементами, доступ к которым осуществляется по индексу или срезу. Встроенная функция `len` принимает в качестве аргумента любой контейнер и возвращает количество содержащихся в нем элементов. Встроенные функции `min` и `max` принимают один аргумент в виде непустого итерируемого объекта, элементы которого можно сравнивать между собой, и возвращают соответственно наименьший и наибольший из элементов. Функции `min` и `max` также можно вызывать с несколькими аргументами; в этом случае они возвращают соответственно наименьший и наибольший из аргументов. Встроенная функция `sum` принимает один аргумент в виде итерируемого объекта, элементы которого являются числами, и возвращает сумму этих чисел.

Преобразование последовательностей

За исключением преобразования байтовых строк в строки Unicode, возможного в версии v2, неявные преобразования между различными типами последовательностей в Python не выполняются. (Преобразования строк подробно рассматриваются в разделе "Unicode" главы 8.) Вызовы встроенных функций `tuple` и `list` с одним аргументом (в качестве которого можно использовать любой итерируемый объект) позволяют получать новые экземпляры вызываемого типа с теми же элементами и с тем же порядком их следования, что и в аргументе.

Конкатенация и повторение

Последовательности одного и того же типа можно *конкатенировать* (объединять) с помощью оператора +. Последовательность *S* можно умножить на целое число *n*, используя оператор *. Выражение *S***n* или *n***S* конкатенирует *n* экземпляров *S*. Если *n*<=0, то выражение *S***n* возвращает пустую последовательность того же типа, что и *S*.

Проверка принадлежности

Оператор *x* `in` *S* выполняет проверку на равенство объекта *x* одному из элементов последовательности (или другого контейнера или итерируемого объекта) *S*. Он возвращает значение `True`, если такой элемент существует, и `False` — в противном случае. Оператор *x* `not in` *S* эквивалентен выражению `not (`*x* `in` *S*`)`. В случае

словарей оператор `x in S` проверяет наличие `x` среди ключей словаря. Однако чаще всего оператор `x in S` применяется к строкам, при этом он выполняет проверку на равенство `x` какой-либо подстроке `S`, а не одиночному символу.

Индексирование последовательности

Для обозначения `n`-го элемента последовательности `S` используется индексирование: `S[n]`. Индексы отсчитываются от нуля (первым элементом последовательности `S` является `S[0]`). Для последовательности `S`, состоящей из `L` элементов, индекс `n` может принимать значения `0`, `1`... вплоть до `L-1`, но не бо́льшие. Кроме того, `n` может принимать значения `-1`, `-2`... вплоть до `-L`, но не меньшие. Отрицательному индексу `n` (например, `-1`) соответствует тот же элемент последовательности `S`, что и индексу `L+n` (например, `L + -1`). Другими словами, `S[-1]`, как и `S[L-1]`, — это последний элемент `S`, `S[-2]` — предпоследний и т.д.

```
x = [1, 2, 3, 4]
x[1]                    # 2
x[-1]                   # 4
```

В случае использования индекса `>=L` или `<-L` генерируется исключение. При попытке присваивания значения элементу с недопустимым индексом также генерируется исключение. В список можно добавлять элементы, но это делается посредством присваивания значений не элементам, а срезам, к рассмотрению которых мы сейчас переходим.

Взятие среза последовательности

Для обращения к частям последовательности `S` можно использовать *срезы*. Срезы имеют следующий синтаксис: `S[i:j]`, где `i` и `j` — целые числа. `S[i:j]` — это подпоследовательность `S`, состоящая из элементов от `i`-го до `j`-го, причем `i`-й элемент включается в срез, а `j`-й не включается. В Python диапазоны всегда включают нижнюю границу, но не включают верхнюю. В тех случаях, когда `j` меньше или равно `i` или `i` больше или равно `L`, длины `S`, срез представляет собой пустую подпоследовательность. Если индекс `i` равен `0`, т.е. если начало среза совпадает с началом последовательности `S`, он может быть опущен. Если `j` больше или равно `L`, то срез простирается до конца последовательности `S`. Вы даже можете опустить оба индекса, и тогда срез будет обозначать поверхностную (мелкую) копию всей последовательности: `S[:]`. Любой из индексов или оба одновременно могут быть меньше `0`. Вот несколько примеров.

```
x = [1, 2, 3, 4]
x[1:3]                  # [2, 3]
x[1:]                   # [2, 3, 4]
x[:2]                   # [1, 2]
```

Отрицательное значение индекса `n` при выделении среза указывает ту же позицию в `S`, что и `L+n`, как и при обычном индексировании. Индексы, значения которых

равны или превышают L, означают конец последовательности S, тогда как отрицательные индексы, значения которых меньше или равны $-L$, означают начало последовательности. При выделении срезов допускается использование расширенного синтаксиса `S[i:j:k]`, где k — *шаг* среза, определяющий расстояние между последовательными значениями индекса. Срез `S[i:j]` эквивалентен срезу `S[i:j:1]`, срез `S[::2]` — это подпоследовательность S, включающая все элементы с четным индексом, а срез `S[::-1]`[6] — та же подпоследовательность S, но с обратным порядком следования элементов. Чтобы срез был непустым при отрицательном шаге, второй индекс должен быть *меньше* первого — это условие обратно тому, которое должно соблюдаться при положительном шаге. Если шаг равен `0`, генерируется исключение.

```
y = list(range(10))
y[-5:]                  # последние пять элементов
[5, 6, 7, 8, 9]
y[::2]                  # через один элемент
[0, 2, 4, 6, 8]
y[10:0:-2]              # то же в обратном порядке
[9, 7, 5, 3, 1]
y[:0:-2]                # то же (более простой способ)
[9, 7, 5, 3, 1]
y[::-2]                 # то же (наилучший способ)
[9, 7, 5, 3, 1]
```

Строки

Строковые объекты (байтовые строки, а также текстовые, известные как строки Unicode) — неизменяемые, поэтому при попытке присвоить значение элементу или срезу строки или удалить его генерируется исключение. Элементы строкового объекта (соответствующие каждому из символов строки) сами являются своего рода строками, длина каждой из которых равна `1`, — в Python не предусмотрен специальный тип данных для "одиночных символов" (за исключением элементов объекта `bytes` в версии v3: в этом случае индексирование приводит к значениям типа `int`). Все срезы строки являются строками того же вида. Строковые объекты имеют множество методов, которые рассматриваются в разделе "Методы строковых и байтовых объектов" главы 8.

Кортежи

Кортежи — неизменяемые объекты, поэтому при попытке присвоить значение элементу или срезу кортежа или удалить его генерируется исключение. Элементами кортежа могут быть произвольные объекты, причем не обязательно одного и того же типа. Элементы кортежа могут быть изменяемыми, но мы не рекомендуем использовать такую практику, поскольку это может приводить к непредвиденным

[6] Срез, получивший шутливое прозвище "марсианский смайлик".

результатам. Срезы кортежа также являются кортежами. Кортежи не имеют обычных (не являющихся специальными) методов, за исключением методов `count` и `index`, аналогичных одноименным методам списков, однако они поддерживают некоторые из специальных методов, рассмотренных в разделе "Специальные методы" главы 4.

Списки

Списки — изменяемые объекты: элементам и срезам списков можно присваивать значения или удалять их. Элементами списка могут быть произвольные объекты, которые не обязательно должны быть одного типа. Срезы списков являются списками.

Изменение списка

Любому элементу списка можно присвоить другое значение, указав его индекс.

```
x = [1, 2, 3, 4]
x[1] = 42                # Теперь x равен [1, 42, 3, 4]
```

Объект *L* списка также можно изменить, используя срез *L* в качестве целевой ссылки в левой части (LHS) оператора присваивания. Правая часть (RHS) оператора присваивания должна быть итерируемым объектом. Если в LHS срез указан в расширенной форме (т.е. с шагом, не равным 1), то количество элементов, указанных RHS, должно совпадать с количеством элементов среза в LHS. Если для среза в LHS шаг не задан или для него явно определено значение 1, то срез в LHS и объект в RHS должны иметь равную длину. При таком присваивании список может удлиняться или укорачиваться.

```
x = [1, 2, 3, 4]
x[1:3] = [22, 33, 44]    # Теперь x равен [1, 22, 33, 44, 4]
x[1:4] = [8, 9]          # Теперь x равен [1, 8, 9, 4]
```

Ниже отмечен ряд важных особых случаев присваивания значений срезам.

- Использование пустого списка `[]` в качестве выражения в RHS приводит к удалению целевого среза из *L*. Другими словами, присваивание *L*`[`*i*`:`*j*`]=[]` равносильно инструкции `del` *L*`[`*i*`:`*j*`]` (или довольно специфической инструкции *L*`[`*i*`:`*j*`]*=0`).
- Использование пустого среза *L* в качестве целевой ссылки в LHS приводит к вставке элементов объекта, указанного в RHS, в соответствующем месте списка *L*. Например, инструкция `L[i:i]=['a','b']` вставляет элементы `'a'` и `'b'` перед элементом списка *L*, которому до присваивания соответствовал индекс *i*.
- Использование среза *L*`[:]`, охватывающего весь объект списка, в качестве целевой ссылки в LHS полностью заменяет содержимое *L*.

Для удаления элемента или среза списка можно использовать инструкцию `del`.

```
x = [1, 2, 3, 4, 5]
del x[1]                # Теперь x равен [1, 3, 4, 5]
del x[::2]              # Теперь x равен [3, 5]
```

Операции над списками, выполняемые на месте

Для списков определены версии операторов + и *, обеспечивающие выполнение соответствующих операций на месте и пригодные для использования в инструкциях составного присваивания. Инструкция составного присваивания `L+=L1` добавляет элементы итерируемого объекта `L1` в конец списка `L` и равносильна инструкции `L.extend(L1)`. Инструкция `L*=n` добавляет `n-1` копию списка `L` в конец `L`; если `n<=0`, то инструкция `L*=n` делает список пустым аналогично инструкции `L[:]=[]`.

Методы списков

Списки поддерживают ряд методов, которые представлены в табл. 3.3. Одни методы возвращают результат без изменения объекта, к которому они применяются (так называемые *немутирующие* методы), другие изменяют объект (так называемые *мутирующие* методы). Поведение многих мутирующих методов напоминает присваивание значений элементам соответствующего среза списка. В табл. 3.3 `L` обозначает любой объект списка, `i` — любой допустимый индекс элемента в списке, а `x` — произвольный объект.

Табл. 3.3. Методы списков

Метод	Описание
Немутирующие методы	
`L.count(x)`	Возвращает количество элементов списка `L`, равных `x`
`L.index(x)`	Возвращает индекс первого встретившегося элемента списка `L`, равного `x`, или, если такой элемент не найден, генерирует исключение
Мутирующие методы	
`L.append(x)`	Добавляет элемент `x` в конец списка `L`; действует подобно инструкции `L[len(L):]=[x]`
`L.extend(s)`	Добавляет все элементы итерируемого объекта `s` в конец списка `L`; действует подобно инструкциям `L[len(L):]=s` и `L+=s`
`L.insert(i,x)`	Вставляет элемент `x` в список `L` перед элементом с индексом `i`, сдвигая последующие элементы `L` (если таковые имеются) "вправо" для создания свободного места (увеличивает `len(L)` на 1, не замещает никаких элементов и не возбуждает исключений; действует подобно инструкции `L[i:i]=[x]`)
`L.remove(x)`	Удаляет из списка `L` первый встретившийся элемент, равный `x`, или возбуждает исключение, если такой элемент не найден

Метод	Описание
`L.pop(i=-1)`	Возвращает значение элемента с индексом `i` и удаляет этот элемент из `L`. Если индекс `i` опущен, удаляет и возвращает последний элемент. Генерирует исключение в случае пустого списка `L` или недопустимого значения индекса `i`
`L.reverse(x)`	Обращает на месте порядок следования элементов `L`
`L.sort(cmp=cmp, key=None, reverse=False)`	Сортирует на месте элементы `L`, попарно сравнивая их (только в версии v2) посредством функции, передаваемой в виде аргумента *cmp* (по умолчанию используется встроенная функция `cmp`). Если аргумент `key` не равен `None`, то в сравнении участвуют не сами элементы `x`, а значения `key(x)`. Более подробно об этом см. в разделе "Сортировка списка". Аргумент `cmp` признан устаревшим в версии v2 (мы рекомендуем *никогда* не использовать его) и вообще отсутствует в версии v3

Все мутирующие методы объектов списка, за исключением метода `pop`, возвращают значение `None`.

Сортировка списка

Метод `sort` сортирует элементы списка на месте (располагая их в возрастающем порядке) гарантированно устойчивым способом (элементы, оказавшиеся равными в результате сравнения, не меняются местами). Что касается практической стороны дела, то метод `sort` работает чрезвычайно быстро, зачастую демонстрируя *сверхъестественную* скорость, поскольку может учитывать наличие уже упорядоченных (в том числе и в обратном порядке) подсписков. (Используемый методом `sort` высокоэффективный алгоритм сортировки *Timsort*, названный так в честь его изобретателя, великого питониста Тима Петерса, представляет собой "нерекурсивный, адаптивный, устойчивый, гибридный алгоритм сортировки, сочетающий сортировку вставками и сортировку слиянием", — вот такой ворох определений!)

В версии v2 метод `sort` принимает три необязательных аргумента, которые могут передаваться с использованием синтаксиса позиционных или именованных аргументов. Аргумент *cmp* (признан устаревшим), если он задан, должен быть функцией, которая, будучи вызванной с двумя элементами списка в качестве аргументов, возвращает значение `-1`, `0` или `1`, в зависимости от того, каким должен считаться первый элемент по отношению ко второму в целях сортировки: бо́льшим, равным или меньшим (если он не задан, то по умолчанию используется встроенная функция `cmp`, имеющая в точности такую же семантику). Аргумент *key*, если он не равен `None`, должен быть функцией, которую можно вызывать с любым элементом списка в качестве единственного аргумента. В этом случае для сравнения любых двух элементов `x` и `y` Python использует вызов `cmp(key(x), key(y))`, а не `cmp(x, y)` (внутренняя реализация этого в Python аналогична реализации идиомы "декорирование, сортировка,

отмена декорирования", представленной в разделе "Поиск и сортировка" главы 16, но работает значительно быстрее). Если аргумент *reverse* равен True, то результат каждого сравнения обращается. Это не то же самое, что обращение списка *L* после сортировки, поскольку метод sort — устойчивый (элементы, оказавшиеся равными в результате сравнения, не меняются местами), независимо от того, равен аргумент *reverse* True или False.

В версии v3 метод sort работает точно так же, как и в версии v2, но объявленного устаревшим аргумента *cmp* теперь просто нет (при попытке его передачи Python генерирует исключение).

```
mylist = ['alpha', 'Beta', 'GAMMA']
mylist.sort()                   # ['Beta', 'GAMMA', 'alpha']
mylist.sort(key=str.lower)      # ['alpha', 'Beta', 'GAMMA']
```

Кроме того, Python предоставляет встроенную функцию sorted (описана в табл. 7.2), позволяющую получить отсортированный список из любого итерируемого объекта. Вслед за первым аргументом (который является итерируемым объектом, предоставляющим элементы) функция sorted принимает те же аргументы, что и метод sort списка.

Модуль operator стандартной библиотеки (раздел "Модуль operator" в главе 15) предоставляет функции высшего порядка attrgetter и itemgetter, которые возвращают функции, пригодные, в частности, для использования в качестве необязательного аргумента key метода sort списков и встроенной функции sorted. Такой же необязательный аргумент key имеют встроенные функции min и max, а также функции nsmallest и nlargest модуля heapq стандартной библиотеки (раздел "Модуль heapq" главы 7).

Операции над множествами

Python поддерживает разнообразные операции, применимые к множествам (как изменяемым, так и неизменяемым). Поскольку множества — это контейнеры, то встроенная функция len может принять множество в качестве своего единственного аргумента и вернуть количество содержащихся в нем элементов. Множество — итерируемый объект, поэтому его можно передать любой функции или методу, принимающим итерируемый аргумент. В данном случае итерации возвращают элементы множества в произвольном порядке. Например, для любого множества *S* функция min(*S*) возвращает наименьший из элементов *S*, поскольку функция min, вызванная с одним аргументом, выполняет итерации по этому аргументу (порядок следования элементов не имеет значения ввиду транзитивности операций сравнения).

Принадлежность к множеству

Выражение *k* in *S* проверяет, является ли объект *k* одним из элементов множества *S*. Если это действительно так, то оно возвращает значение True, в противном

случае возвращается значение `False`. Выражение *k* `not in` *S* эквивалентно выражению `not (`*k* `in` *S*`)`.

Методы множеств

Множества поддерживают ряд методов, которые представлены в табл. 3.4. Немутирующие методы возвращают результат, не изменяя объект, к которому они применяются, и могут вызываться также для экземпляров неизменяемых множеств (тип `frozenset`). Мутирующие методы способны изменять объекты, к которым они применяются, и могут вызываться только для экземпляров изменяемых множеств (тип `set`). В табл. 3.4 *S* обозначает любой объект множества, *S1* — любой итерируемый объект с хешируемыми элементами (часто, но не обязательно, изменяемое или неизменяемое множество), а *x* — любой хешируемый объект.

Табл. 3.4. Методы множеств

Метод	Описание
Немутирующие методы	
S`.copy()`	Возвращает поверхностную копию *S* (копия, элементами которой являются те же самые объекты, что и в *S*, а не их копии) подобно вызову функции `set(`*S*`)`
S`.difference(`*S1*`)`	Возвращает множество всех элементов *S*, отсутствующих в *S1*
S`.intersection(`*S1*`)`	Возвращает множество всех элементов *S*; которые содержатся также в *S1*
S`.issubset(`*S1*`)`	Возвращает значение `True`, если все элементы *S* содержатся также в *S1*; в противном случае возвращает значение `False`
S`.issuperset(`*S1*`)`	Возвращает значение `True`, если все элементы *S1* содержатся также в *S*; в противном случае возвращает значение `False` (равносильно вызову *S1*`.issubset(`*S*`)`)
S`.symmetric_difference(`*S1*`)`	Возвращает набор всех элементов, которые содержатся либо в *S*, либо в *S1*, но не в обоих множествах одновременно
S`.union(`*S1*`)`	Возвращает множество всех элементов, которые содержатся в *S*, *S1* или в обоих множествах одновременно
Мутирующие методы	
S`.add(`*x*`)`	Добавляет *x* в *S* в качестве элемента; не выполняет никаких действий, если *x* уже является элементом *S*
S`.clear()`	Удаляет все элементы из *S*, оставляя *S* пустым
S`.discard(`*x*`)`	Удаляет элемент *x* из *S*; не выполняет никаких действий, если *x* не является элементом *S*
S`.pop()`	Удаляет и возвращает произвольный элемент *S*
S`.remove(`*x*`)`	Удаляет элемент *x* из *S*; генерирует исключение `KeyError`, если *x* не является элементом *S*

Все мутирующие методы множеств, за исключением метода pop, возвращают значение None.

Метод pop можно использовать для деструктивного итерирования по элементам множества, требующего использования лишь небольших объемов дополнительной памяти. Эта экономия памяти делает метод pop весьма удобным для циклического обхода элементов крупных множеств в тех случаях, когда "расходование" множества в ходе выполнения цикла входит в ваши намерения. По сравнению с недеструктивными циклами вида

```
for элемент in S:
    ...обработка элемента...
```

деструктивные циклы вида

```
while S:
    элемент = S.pop()
    ...обработка элемента...
```

обладают тем потенциальным преимуществом, что позволяют изменять множество *S* (например, добавлять и/или удалять его элементы), что запрещено в недеструктивных циклах.

Кроме того, множества поддерживают ряд дополнительных мутирующих методов: difference_update, intersection_update, symmetric_difference_update и update (соответствует немутирующему методу union). Каждый из этих мутирующих методов выполняет ту же операцию, что и соответствующий немутирующий метод, но делает это на месте, изменяя множество, для которого он вызван, и возвращая значение None. Доступ к четырем аналогичным немутирующим методам также возможен с использованием операторного синтаксиса. Если *S2* — изменяемое или неизменяемое множество, то вызовам указанных методов соответствуют следующие выражения: *S-S2*, *S&S2*, *S^S2* и *S|S2*. Доступ к соответствующим мутирующим методам обеспечивают составные операторы присваивания: *S-=S2*, *S&=S2*, *S^=S2* и *S|=S2*. Обратите внимание на то, что в случае использования синтаксиса простого или составного оператора присваивания оба операнда должны быть изменяемыми или неизменяемыми множествами. Но если вы вызываете соответствующие методы по именам, то аргумент *S1* может быть любым итерируемым объектом с хешируемыми элементами, который будет обрабатываться так, как если бы аргумент был передан методу в виде set(*S1*).

Операции над словарями

Python предоставляет разнообразные методы, применимые к словарям. Поскольку словари — это контейнеры, встроенная функция len может принять словарь в качестве аргумента и вернуть количество содержащихся в нем элементов (пар "ключ–значение"). Словарь — итерируемый объект, поэтому его можно

передавать любой функции, принимающей итерируемый объект в качестве аргумента. В этом случае итерации возвращают ключи словаря в произвольном порядке[7]. Например, вызов функции `min(D)`, где `D` — произвольный словарь, возвращает наименьший из ключей `D`.

Принадлежность к словарю

Инструкция `k in D` проверяет, является ли объект `k` ключом в словаре `D`. Если это так, то она возвращает значение `True`, в противном случае — значение `False`. Инструкция `k not in D` аналогична инструкции `not (k in D)`.

Индексирование словаря

Для обращения к значению словаря `D`, ассоциированному с ключом `k`, используется индексирование: `D[k]`. При попытке обращения к значению по отсутствующему ключу генерируется исключение.

```
d = {'x':42, 'y':3.14, 'z':7}
d['x']                          # 42
d['z']                          # 7
d['a']                          # генерируется исключение KeyError
```

Присваивание значения элементу словаря с использованием индексирования по отсутствующему ключу (например, `D[новый_ключ]=значение`) — законная операция, в результате которой ключ и значение добавляются в словарь в качестве его элемента.

```
d = {'x':42, 'y':3.14}
d['a'] = 16             # состояние d: {'x':42, 'y':3.14, 'a':16}
```

Инструкция `del`, выполняемая в форме `del D[k]`, удаляет из словаря элемент с ключом `k`. Если ключ `k` отсутствует в словаре `D`, то инструкция `del D[k]` генерирует исключение `KeyError`.

Методы словарей

Словари поддерживают ряд методов, представленных в табл. 3.5. Немутирующие методы возвращают результат без изменения объекта, к которому они применяются, тогда как мутирующие методы могут изменять объекты. В табл. 3.5 `D` и `D1` обозначают любой объект словаря, `k` — любой хешируемый объект, а `x` — любой объект.

[7] **Новое в версии 3.6:** внутренняя реализация словарей изменена. Теперь ключи итерируются в порядке их вставки, но лишь до тех пор, пока к словарю не будет применена мутирующая функция, осуществляющая операцию, отличную от операции вставки элементов. В целях обеспечения устойчивости кода не следует рассчитывать на это негарантированное поведение, пока оно не будет подтверждено в официальной документации Python.

Табл. 3.5. Методы словарей

Метод	Описание
Немутирующие методы	
`D.copy()`	Возвращает поверхностную (мелкую) копию словаря (копия, элементами которой являются такие же объекты, как в *D*, а не их копии), подобно вызову функции `dict(D)`
`D.get(k[,x])`	Возвращает значение `D[k]`, если *k* является ключом в *D*; в противном случае возвращает *x* (или значение `None`, если *x* не задан)
`D.items()`	В версии v2 возвращает новый список, включающий все элементы (пары "ключ–значение") *D*; в версии v3 возвращает не список, а экземпляр итерируемого объекта `dict_items`
`D.iteritems()`	Возвращает итератор по всем элементам (парам "ключ–значение") (только в версии v2)
`D.iterkeys()`	Возвращает итератор по всем ключам *D* (только в версии v2)
`D.itervalues()`	Возвращает итератор по всем значениям *D* (только в версии v2)
`D.keys()`	В версии v2 возвращает новый список, содержащий все ключи *D*; в версии v3 возвращает не список, а итерируемый экземпляр `dict_keys`
`D.values()`	В версии v2 возвращает новый список, содержащий все значения *D*; в версии v3 возвращает не список, а итерируемый экземпляр `dict_values`
Мутирующие методы	
`D.clear()`	Удаляет из *D* все элементы, оставляя *D* пустым
`D.pop(k[,x])`	Удаляет и возвращает значение `D[k]`, если *k* является ключом в *D*; в противном случае возвращает *x* (генерирует исключение `KeyError`, если *x* не задан)
`D.popitem()`	Удаляет и возвращает произвольный элемент (пару "ключ–значение")
`D.setdefault(k[,x])`	Возвращает значение `D[k]`, если *k* является ключом в *D*; в противном случае устанавливает `D[k]` равным *x* (или значению `None`, если *x* не задан) и возвращает *x*
`D.update(D1)`	Для каждого *k* в отображении *D1* устанавливает `D[k]` равным `D1[k]`

Методы `items`, `keys` и `values` возвращают свои результаты (в v2 — списки, в v3 — итерируемые экземпляры `dict_...`) с произвольным порядком следования элементов в них[8]. Если вы вызываете несколько этих методов без изменения состоя-

[8] В версии 3.6, о чем уже было сказано в предыдущем примечании, — в порядке, соответствующем порядку вставки элементов, но лишь до первого применения к словарю метода, выполняющего действие, отличное от вставки элемента.

ния словаря, то для всех вызовов порядок элементов в возвращаемых списках будет одним и тем же. Методы iteritems, iterkeys и itervalues (только в версии v2) возвращают итераторы, эквивалентные этим спискам (итераторы рассматриваются в разделе "Итераторы").

Никогда не изменяйте набор ключей словаря в процессе итерирования по нему

Итератор или экземпляр dict_... требует меньшего объема памяти, чем список, но в процессе выполнения итераций по любому из итераторов данного словаря вы ни в коем случае не должны изменять набор ключей в словаре (т.е. не должны добавлять или удалять их).

В версии v2 это ограничение на итерирование по спискам, возвращаемым методами items, keys или values, не действует (в версии v3 для тех же целей можно явно создавать списки посредством вызовов наподобие list(*D*.keys())). Непосредственное итерирование по словарю *D* в точности эквивалентно итерированию по итераторам *D*.iterkeys() в версии v2 и *D*.keys() в версии v3.

В версии v3 все типы dict_... — итерируемые. Кроме того, типы dict_items и dict_keys реализуют немутирующие методы set и ведут себя подобно неизменяемым множествам. Это не относится к типу dict_values, поскольку, в отличие от других типов (и множеств), он может содержать повторяющиеся элементы.

Метод popitem можно использовать для деструктивного итерирования по словарю. Оба метода, items и popitem, возвращают элементы словаря в виде пар "ключ–значение". Метод popitem удобно использовать для циклического обхода элементов крупных словарей в тех случаях, когда "потребление" элементов словаря в ходе выполнения цикла входит в ваши намерения.

Метод *D*.setdefault(*k*, *x*) возвращает тот же результат, что и метод *D*.get(*k*, *x*), но если ключ *k* отсутствует в словаре *D*, то метод setdefault инициализирует элемент словаря указанным значением (*D*[k]=*x*). (В современных версиях Python метод setdefault используется редко, поскольку тип collections.defaultdict, рассмотренный в разделе "Класс defaultdict" главы 7, реализует эту функциональность более элегантным и быстрым способом.)

Метод pop возвращает тот же результат, что и метод get, но если *k* является ключом словаря *D*, то метод pop удаляет из словаря соответствующий элемент (если *x* не задан, а *k* не является ключом в *D*, то метод get возвращает значение None, тогда как метод pop генерирует исключение).

Метод update может принимать итерируемый объект с парами "ключ–значение" в качестве альтернативы аргументу в виде объекта отображения, а также именованные аргументы вместо позиционного аргумента или в дополнение к нему. В обоих случаях семантика остается той же, что и при передаче подобных аргументов в случае вызова встроенного типа dict (см. раздел "Словари").

Управляющие инструкции

Поток управления — это очередность выполнения кода программы. Поток управления программы на языке Python зависит от условных инструкций, инструкций циклов и вызовов функций. (В этом разделе мы рассмотрим условную инструкцию `if`, а также циклы `for` и `while`; функции рассматриваются в разделе "Функции".) Возбуждение и обработка исключений также влияют на поток управления; исключения рассматриваются в главе 5.

Инструкция `if`

Часто возникает необходимость в том, чтобы некоторые инструкции выполнялись лишь при соблюдении определенного условия. Также нередки ситуации, когда инструкции, подлежащие выполнению, должны выбираться исходя из выполнения одного из нескольких взаимоисключающих условий. Составная инструкция `if`, включающая предложения `if`, `elif` и `else`, обеспечивает условное выполнение блоков инструкций. Она имеет следующий синтаксис.

```
if выражение:
    инструкции
elif выражение:
    инструкции
elif выражение:
    инструкции
...
else:
    инструкции
```

Предложения `elif` и `else` могут отсутствовать. Обратите внимание на то, что, в отличие от некоторых других языков программирования, в Python отсутствует инструкция `switch`. Все виды условной обработки осуществляются с помощью инструкций `if`, `elif` и `else`.

Вот пример использования инструкции `if`, включающей все три вида предложений (используемый стиль не является оптимальным).

```
if x < 0: print('x - отрицательное число')
elif x % 2: print('x - нечетное положительное число')
else: print('x - четное неотрицательное число')
```

Если в предложении содержится несколько инструкций (т.е. если предложение управляет блоком инструкций), то инструкции располагаются на отдельных логических строках после строки, содержащей ключевое слово предложения (*строка заголовка* предложения), с отступом вправо относительно нее. Признаком конца блока служит возврат выделения отступом к уровню, который соответствует уровню отступа заголовка предложения (или уровню, смещенному влево от него). Простые одиночные инструкции, как в данном примере, могут располагаться после знака

двоеточия (:) в той же логической строке, что и заголовок, или на отдельной логической строке, следующей непосредственно за логической строкой заголовка с отступом вправо относительно нее. Большинство программистов на Python предпочитают использовать для *охраняемых* инструкций стиль отдельных строк с отступами величиной четыре пробела. Такой стиль более распространен, улучшает удобочитаемость кода и рекомендован документом **PEP 8** (`https://www.python.org/dev/peps/pep-0008/#indentation`). Поэтому общепринятым является следующий стиль.

```
if x < 0:
    print('x - отрицательное число')
elif x % 2:
    print('x - нечетное положительное число')
else:
    print('x - четное неотрицательное число')
```

В предложениях `if` и `elif` в качестве условия можно использовать любое выражение Python. В подобных случаях говорят, что условие используется в логическом (булевом) контексте. В логическом контексте любое выражение рассматривается как истинное или ложное. Как уже отмечалось, любое ненулевое число или любой непустой контейнер (строка, кортеж, список, словарь, множество) расценивается как истинное значение. Нуль (любого числового типа), `None` и пустой контейнер расцениваются как ложные значения. Для тестирования значения `x` в логическом контексте используйте следующий стиль написания кода:

```
if x:
```

Это наиболее ясная и соответствующая духу Python форма кода. Избегайте использования любой из следующих форм кода.

```
if x is True:
if x == True:
if bool(x):
```

Фразы "выражение *возвращает* значение `True`" (имеется в виду возврат значения `1` булевого типа) и "*вычисление* выражения дает истинное значение" (имеется в виду возврат выражением любого результата, являющегося истинным в логическом контексте) имеют различный смысл. Например, при тестировании выражения в предложении `if` вас интересует лишь истинность вычисленного результата, а не его точное значение.

Если вычисление условия предложения `if` дает истинный результат, то инструкции данного предложения выполняются, и на этом выполнение всей инструкции `if` заканчивается. В противном случае Python поочередно вычисляет условия каждого из предложений `elif` в порядке их следования. Инструкции первого из предложений `elif`, для которого вычисление условия дает истинный результат (если таковое встретится), выполняются, и на этом выполнение полной инструкции `if` заканчивается. В противном случае выполняются инструкции предложения `else`, если таковое имеется, после чего выполняются инструкции, следующие за полным оператором `if`.

Инструкция while

Инструкция while обеспечивает многократное выполнение инструкции или блока инструкций, управляемых условным выражением. Она имеет следующий синтаксис.

```
while выражение:
    инструкции
```

Инструкция while может включать предложение else (раздел "Предложение else в инструкциях циклов", а также инструкции break и continue (разделы "Инструкция break" и "Инструкция continue").

Вот пример типичной инструкции while.

```
count = 0
while x > 0:
    x //= 2     # целочисленное деление
    count += 1
print('The approximate log2 is', count)
```

Прежде всего Python вычисляет *выражение* (так называемое *условие продолжения цикла*). Если результат вычисления выражения представляет собой ложное значение, то выполнение цикла while прекращается. Если же результат представляет собой истинное значение, то выполняются инструкции, образующие *тело цикла*. Сразу по завершении выполнения тела цикла Python вновь проверяет условие, чтобы определить, должна ли выполняться следующая итерация. Этот процесс продолжается до тех пор, пока не будет получено ложное значение условия, после чего выполнение цикла while прекращается.

Код, образующий тело цикла, должен быть таким, чтобы в конечном счете условие могло приобрести ложное значение, иначе цикл никогда не сможет завершиться (если только в теле цикла не будет сгенерировано исключение или его выполнение не будет прервано инструкцией break). Циклы, выполняющиеся в теле функций, также могут завершаться при выполнении инструкции return в теле цикла, поскольку это будет означать завершение выполнения функции в целом.

Инструкция for

Инструкция for обеспечивает многократное выполнение инструкции или блока инструкций, управляемых итерируемым выражением. Она имеет следующий синтаксис.

```
for цель in итерируемое_выражение:
    инструкции
```

Ключевое слово in является частью синтаксиса инструкции for. В данном случае оно имеет иное назначение, нежели оператор in, с помощью которого проверяется принадлежность объекта коллекции объектов.

Вот пример типичной инструкции for.

```
for letter in 'ciao':
    print('дай мне', letter, '...')
```

Инструкция for может включать предложение else (раздел "Предложение else в инструкциях циклов"), а также инструкции break и continue (разделы "Инструкция break" и "Инструкция continue"). *Итерируемое_выражение* может быть любым выражением Python, пригодным для передачи в качестве аргумента встроенной функции iter, возвращающей объект итератора (подробно обсуждается в следующем разделе). В частности, любая последовательность является итерируемым объектом. Как правило, *цель* — это идентификатор, который представляет *управляющую переменную* цикла; в цикле for эта переменная последовательно получает значения в виде поочередно выбираемых элементов итератора. Инструкция или инструкции, образующие *тело цикла*, выполняются однократно для каждого элемента итерируемого объекта (если только цикл не будет преждевременно завершен в результате возникновения исключения или выполнения инструкции break или return). Обратите внимание на то, что наличие в теле цикла инструкции break, способной прекратить выполнение цикла, является тем единственным случаем, когда в цикле допускается использовать *неограниченный* итерируемый объект, т.е. объект, который сам по себе способен сколь угодно долго возвращать элементы.

Кроме того, в качестве цели можно указать несколько идентификаторов, как при присваивании с распаковкой. В этом случае элементы итератора должны быть итерируемыми объектами, каждый из которых содержит ровно столько элементов, сколько идентификаторов указано в качестве цели (только в версии v3: одному и только одному из идентификаторов может предшествовать символ "звездочка" (*); отмеченному звездочкой идентификатору присваивается список всех элементов, которые не были присвоены другим целям, если таковые имеются). Например, если *d* — словарь, то типичный цикл по всем его элементам (парам "ключ–значение") может быть организован следующим образом.

```
for key, value in d.items():
    if key and value:          # выводит только ключи и значения,
        print(key, value)      # являющиеся логически истинными
```

Метод items возвращает список (в версии v2; в версии v3 — итерируемый объект другого типа) пар "ключ–значение", поэтому для распаковки каждого элемента в ключ и значение мы используем цикл for с двумя идентификаторами в качестве цели. И хотя в данном случае компонентами цели являются два обычных идентификатора, значения могут присваиваться любому допустимому LHS-выражению (см. раздел "Операции присваивания").

```
prototype = [1, 'заместитель', 3]
for prototype[1] in 'xyz': print(prototype)
# вывод: [1, 'x', 3], затем [1, 'y', 3], затем [1, 'z', 3]
```

Не изменяйте изменяемые объекты при выполнении цикла по их элементам

Если в состав итерируемого объекта входит изменяемый объект, не пытайтесь изменить его при выполнении цикла по его элементам. Так, в предыдущем примере с выводом пар "ключ–значение" нельзя изменить словарь *d* в версии v3 (а также в версии v2 в случае использования функции `iteritems` вместо функции `items`). Метод `iteritems` в версии v2 (или метод `items` в версии v3) возвращает итерируемые объекты, базовым объектом которых является словарь *d*, поэтому тело цикла не может изменить набор ключей *d* (например, посредством выполнения инструкции `del` *d*`[`*ключ*`]`). Вызовы *d*`.items()` в версии v2 (или `list(`*d*`.items())` в версии v3) возвращают новые независимые списки, поэтому *d* не является базовым объектом итерируемого объекта; следовательно, тело цикла может изменить *d*. В частности, можно отметить следующее.

- При обходе элементов списка в цикле не следует вставлять, присоединять и удалять элементы (присваивание значения элементу по существующему индексу разрешается).
- При обходе элементов словаря в цикле не следует добавлять или удалять элементы (присваивание значения элементу по существующему ключу разрешается).
- При обходе элементов множества в цикле не следует добавлять или удалять элементы (не разрешены никакие операции, изменяющие множество).

Управляющим целевым переменным можно присваивать новые значения в теле цикла, однако перед началом выполнения следующей итерации цикла им будут заново присваиваться значения в виде очередного элемента итератора. Если итератор не возвращает ни одного значения, то тело цикла не выполняется. В этом случае инструкция `for` вообще не присваивает какие-либо значения управляющим переменным. Однако, если итератор возвращает хотя бы одно значение, то по завершении цикла `for` управляющие переменные сохраняют те значения, которые они получили при выполнении последней итерации. Поэтому следующий код корректен лишь в случае непустой последовательности `someseq`.

```
for x in someseq:
    process(x)
print('Последним был обработан элемент ', x)
```

Итераторы

Итератор — это любой объект *i*, обеспечивающий возможность выполнения вызова `next(`*i*`)`. Метод `next(`*i*`)` возвращает следующий элемент итератора *i* или, в случае исчерпания элементов итератора, генерирует исключение `StopIteration`.

Также возможен вариант вызова этого метода в форме `next(i, default)`; в этом случае при исчерпании элементов итератора возвращается значение *default*.

При написании кода класса (раздел "Классы и экземпляры" в главе 4) можно позволить его экземплярам быть итераторами, определив специальный метод `__next__` (в версии v2 этот метод должен называться `next`, что в конечном счете было признано ошибкой проектирования), который не принимает никаких других аргументов, кроме `self`, и возвращает следующий элемент или генерирует исключение `StopIteration`. Большинство итераторов создается посредством явных или неявных вызовов встроенной функции `iter`, описанной в табл. 7.2. Вызов генератора также возвращает генератор, о чем пойдет речь в разделе "Генераторы".

Функция `iter` неявно вызывается инструкцией `for` для получения итератора. Инструкция

```
for x in c:
    инструкции
```

в точности эквивалентна следующему коду:

```
_temporary_iterator = iter(c)
while True:
    try: x = next(_temporary_iterator)
    except StopIteration: break
    инструкции
```

Здесь *_temporary_iterator_* — произвольное имя, не используемое в других местах кода в текущей области видимости.

Таким образом, если вызов `iter(c)` возвращает итератор *i*, для которого вызов `next(i)` никогда не генерирует исключение `StopIteration` (*неограниченный итератор*), то цикл `for x in c` никогда не завершится (если только он не будет преждевременно завершен в результате либо выполнения инструкции `break` или `return`, либо возникновения или распространения исключения). Метод `iter(c)`, в свою очередь, вызывает специальный метод `c.__iter__()`, возвращающий итератор для объекта *c*. Метод `__iter__` более подробно обсуждается в разделе "Методы контейнеров" главы 4.

Стандартная библиотека предоставляет модуль `itertools` (раздел "Модуль `itertools`" в главе 7), который содержит многочисленные средства, предназначенные для создания итераторов и манипулирования ими.

Функции `range` и `xrange`

Обход последовательности целых чисел — весьма распространенная задача в программировании, поэтому Python предоставляет встроенную функцию `range` (в версии v2 — также функцию `xrange`), которая генерирует целочисленные последовательности. Простейшей формой организации цикла, выполняющегося *n* раз, является следующая.

```
for i in range(n):
    инструкции
```

В версии v2 функция `range(x)` возвращает список последовательных целых чисел, значения которых заключены в диапазоне от 0 (включительно) до *x* (исключая это значение). Функция `range(x, y)` возвращает список последовательных целых чисел, значения которых заключены в диапазоне от *x* (включительно) до *y* (исключая это значение). Если *x* больше или равно *y*, возвращается пустой список. Функция `range(x, y, шаг)` возвращает список целых чисел, которые заключены в диапазоне от *x* (включительно) до *y* (исключая это значение) и принимают значения, отсчитываемые от начального значения с заданным шагом. Если величина шага меньше *0*, то отсчет значений в направлении их убывания ведется от *x* до *y*. Функция `range(x, y, шаг)` возвращает пустой список, если *x* больше или равно *y* и *шаг* больше 0 или *x* меньше или равно *y* и *шаг* меньше *0*. Если *шаг* равен 0, то вызов `range(x, y, 0)` генерирует исключение.

В то время как в версии v2 функция `range` возвращает обычный объект списка, пригодный для использования в любых целях, функция `xrange` (определена только в версии v2) возвращает специальный объект, предназначенный исключительно для использования в итерациях, как в приведенной перед этим инструкции `for` (сама по себе функция `xrange` не возвращает итератор, однако в случае необходимости вы легко сможете получить его, вызвав функцию `iter(xrange(...))`). Специальный объект, возвращаемый функцией `xrange`, требует меньшего объема памяти по сравнению с объектом списка (особенно в случае больших диапазонов). Не забывая об этом, функцию `range` можно использовать везде, где можно использовать функцию `xrange`, но не наоборот. Вот пример в версии v2.

```
>>> print(range(1, 5))
[1, 2, 3, 4]
>>> print(xrange(1, 5))
xrange(1, 5)
```

Здесь функция `range` возвращает обычный список, отображаемый обычным способом, тогда как функция `range` возвращает специальный объект, который отображается специфическим для него способом.

В версии v3 функция `xrange` отсутствует, а функция `range` работает аналогично функции `xrange`, предоставляемой в версии v2. Если в версии v3 вам потребуется список, представляющий собой арифметическую прогрессию целых чисел, используйте вызов функции `list(range(...))`.

Генераторы списков

Типичный способ применения цикла `for` — просмотр элементов итерируемого объекта и создание нового списка путем объединения результатов вычислений, выполняемых над некоторыми или всеми элементами данного объекта. Python обеспечивает компактный способ записи кода этой идиомы с помощью специального выражения — так называемого *генератора списка* (list comprehension). Поскольку генератор списка является выражением (а не блоком инструкций), его можно использовать везде, где разрешено использование выражения (например, в качестве аргумента при вызове функции, в инструкции `return` или в качестве подвыражения другого выражения).

Генератор списка имеет следующий синтаксис:

```
[ выражение for цель in итерируемый_объект предложения_гс ]
```

Здесь обозначения *цель* и *итерируемый_объект* имеют тот же смысл, что и в обычной инструкции `for`. Если *выражение* обозначает кортеж, то его необходимо заключить в круглые скобки.

Обозначение *предложения_гс* представляет серию из нуля или нескольких предложений, каждое из которых может иметь одну из следующих двух форм.

```
for цель in итерируемый_объект
if выражение
```

Здесь обозначения *цель* и *итерируемый_объект* в каждом предложении `for` генератора списка имеют те же синтаксис и смысл, что и в обычной инструкции `for`, а обозначение *выражение* в каждом предложении `if` генератора списка имеет те же синтаксис и смысл, что и в обычной инструкции `if`.

Генератор списка эквивалентен циклу `for`, создающему тот же список посредством повторных вызовов метода `append` результирующего списка. Например, код (ниже для ясности результат работы генератора списка присваивается переменной)

```
result1 = [x+1 for x in some_sequence]
```

эквивалентен (не считая различий в именах переменных) следующему циклу `for`:

```
result2 = []
for x in some_sequence:
    result2.append(x+1)
```

Приведем пример генератора списка, в котором используется предложение `if`.

```
result3 = [x+1 for x in some_sequence if x>23]
```

Этот код эквивалентен следующему циклу `for`, содержащему инструкцию `if`.

```
result4 = []
for x in some_sequence:
    if x>23:
        result4.append(x+1)
```

А вот пример генератора списка, в котором вложенное предложение `for` используется для представления "списка списков" в виде плоского, т.е. лишенного внутренней иерархии, списка всех его элементов:

```
result5 = [x for sublist in listoflists for x in sublist]
```

Этот код эквивалентен следующему циклу `for`, в теле которого содержится вложенный цикл `for`.

```
result6 = []
for sublist in listoflists:
    for x in sublist:
        result6.append(x)
```

Как демонстрируют эти примеры, порядок следования предложений `for` и `if` в генераторе списка совпадает с их порядком следования в эквивалентном цикле, но в генераторе списка это вложение неявное. Если вы запомните фразу "порядок следования предложений как во вложенном цикле", то это поможет вам располагать предложения генератора списка в правильном порядке.

Не создавайте списки, если без этого можно обойтись

Если вам нужен простой перебор значений, а не получение фактического индексируемого списка, рассмотрите возможность использования выражения-генератора (раздел "Выражения-генераторы" в главе 4). Тем самым вы избежите создания списка и сократите потребление памяти.

Генераторы списков и области видимости переменных

В версии v2 целевые переменные предложений `for` в генераторе списка остаются связанными с их последним значением, которое они имели в тот момент, когда генератор списка завершил работу, точно так же, как это было бы в том случае, если бы вы использовали инструкции `for`. Поэтому всегда следите за тем, чтобы имена целевых переменных не совпадали с именами других локальных переменных. В версии v3 об этом не стоит беспокоиться, поскольку в ней генераторы списков имеют собственные области видимости. Кроме того, в обеих версиях, v2 и v3, это замечание относится только к генераторам списков, но не к выражениям-генераторам, а также к генераторам множеств и словарей.

Генераторы множеств

Генераторы множеств (set comprehension) имеют точно такие же синтаксис и семантику, что и генераторы списков, но заключаются в фигурные скобки (`{}`), а не квадратные (`[]`). Результат представляет собой множество, поэтому порядок следования элементов не имеет значения.

```
s = {n//2 for n in range(10)}
print(sorted(s))                # вывод: [0, 1, 2, 3, 4]
```

В результате, возвращенном аналогичным генератором списка, каждый элемент был бы повторен дважды, но, разумеется, множества содержат только уникальные элементы.

Генераторы словарей

Генераторы словарей (dict comprehension) имеют точно такой же синтаксис, что и генераторы множеств, за исключением того, что перед предложением `for` вместо одиночного выражения используются два выражения, разделенные двоеточием: *ключ*:*значение*. Результат представляет собой словарь, поэтому, как и в случае

множеств, порядок элементов (в словаре это пары "ключ–значение") не имеет значения.

```
d = {n:n//2 for n in range(5)}
print(d)                          # вывод: {0:0, 1:0, 2:1, 3:1, 4:2]
                                  # или в другом порядке
```

Инструкция `break`

Инструкцию `break` разрешается использовать только в теле цикла. Выполнение инструкции `break` приводит к преждевременному выходу из цикла. Если цикл вложен в другие циклы, то данная инструкция прерывает только ближайший вложенный цикл. На практике инструкция `break` включается, как правило, в предложение `if` тела цикла и поэтому выполняется по условию.

Инструкция `break` часто применяется для реализации циклов, в которых решение относительно необходимости продолжения цикла принимается посреди процесса выполнения каждой итерации. (То, что Дональд Кнут назвал "нецелыми циклами" в своей чрезвычайно конструктивной статье, написанной в 1974 году: `http://www.kohala.com/start/papers.others/knuth.dec74.html`. В той статье им также было предложено использовать для структурирования программы "такие средства, как отступы, вместо разделителей", — т.е. именно то, что и сделано в Python). Проиллюстрируем сказанное соответствующим примером.

```
while True:         # Этот цикл сам по себе никогда не завершится
    x = get_next()
    y = preprocess(x)
    if not keep_looping(x, y): break
    process(x, y)
```

Инструкция `continue`

Инструкция `continue` может встречаться лишь в теле цикла. Выполнение инструкции `continue` прерывает выполнение текущей итерации тела цикла и передает управление в начало цикла. На практике инструкция `continue` обычно используется в предложениях `if` тела цикла и поэтому выполняется по условию.

В некоторых случаях инструкция `continue` может заменять вложенные инструкции `if`.

```
for x in some_container:
    if not seems_ok(x): continue
    lowbound, highbound = bounds_to_test()
    if x<lowbound or x>=highbound: continue
    pre_process(x)
    if final_check(x):
        do_processing(x)
```

Этот код эквивалентен следующему циклу без инструкции `continue`, включающему условную обработку.

```
for x in some_container:
    if seems_ok(x):
        lowbound, highbound = bounds_to_test()
        if lowbound <= x < highbound:
            pre_process(x)
            if final_check(x):
                do_processing(x)
```

Плоское лучше, чем вложенное

Обе приведенные выше версии кода работают одинаково, поэтому выбор одной из них — дело вкуса каждого. Один из принципов дзен-философии Python (с полным перечнем которых вы сможете ознакомиться, выполнив в интерактивной оболочке Python команду `import this`) гласит: "Плоское лучше, чем вложенное". Инструкция `continue` — это лишь один из предлагаемых в Python способов, способствующих уменьшению глубины вложения структур в циклах, если таков будет ваш выбор.

Предложение `else` в инструкциях циклов

Инструкции `while` и `for` могут включать необязательное завершающее предложение `else`. Инструкция или блок, следующие за ключевым словом `else`, выполняются, если цикл завершился естественным образом (вследствие исчерпания итератора цикла `for` или когда условие цикла `while` приобрело ложное значение), но не тогда, когда цикл завершается преждевременно (вследствие выполнения инструкции `break` либо `return` или возникновения исключения). В тех случаях, когда цикл содержит одну или несколько инструкций `break`, нередко требуется проверять, закончился ли цикл естественным образом или был преждевременно прерван. Для этого можно использовать предложение `else` цикла.

```
for x in some_container:
    if is_ok(x): break # элемент x удовлетворяет условию,
                       # прервать цикл
else:
    print('Внимание: подходящий элемент в контейнере не найден!')
    x = None
```

Инструкция `pass`

Тело составной инструкции Python не может быть пустым; оно должно содержать хотя бы одну инструкцию. В подобных случаях, когда синтаксис требует наличия инструкции, можно использовать инструкцию-заполнитель `pass`, которая не выполняет никаких действий. Ниже приведен пример использования инструкции `pass` в условной инструкции с несколько запутанной логикой проверки взаимоисключающих условий.

```
if condition1(x):
    process1(x)
elif x>23 or condition2(x) and x<5:
    pass      # в данном случае никакие действия не выполняются
elif condition3(x):
    process3(x)
else:
    process_default(x)
```

Пустые инструкции def и class: используйте строки документирования, а не инструкцию pass

В качестве тела пустой во всех остальных отношениях инструкции `def` или `class` лучше использовать строки документирования (раздел "Строки документирования"). Если вы воспользуетесь этой возможностью, то добавлять инструкцию `pass` не потребуется (вы можете добавить и ее, но это не будет соответствовать стилю Python).

Инструкции try и raise

Python поддерживает обработку исключений с помощью инструкции `try`, включающей предложения `try`, `except`, `finally` и `else`. Ваш код может явно генерировать исключение с помощью инструкции `raise`. Когда генерируется исключение, обычный поток управления программы прерывается, и Python выполняет поиск подходящего обработчика исключений, о чем подробно говорится в разделе "Распространение исключений" главы 5.

Инструкция with

Иногда можно улучшить читаемость кода за счет использования инструкции `with` вместо инструкции `try/finally`. Этот вопрос подробно обсуждается в разделе "Инструкция `with` и менеджеры контекста" главы 5.

Функции

В типичной программе на языке Python большинство инструкций является частью функций. Код тела функции может работать быстрее кода модуля верхнего уровня, о чем говорится в разделе "Избегайте использования инструкций `exec` и `from ... import *`" главы 16, поэтому для размещения большей части кода в функциях есть веские практические причины, тем более что это делает код более понятным и улучшает его читаемость. К тому же организация кода в виде функций, а не в виде кода на уровне модуля, не имеет никаких неприятных последствий.

Функция — это группа инструкций, выполняемых по запросу. Python предоставляет множество встроенных функций и позволяет программистам определять собственные функции. Запрос выполнения функции называют *вызовом функции*.

Вызываемой функции можно передавать аргументы, определяющие данные, которые она будет использовать при выполнении вычислений. В Python функция всегда возвращает результат, которым может быть либо значение `None`, либо значение, представляющее результат выполненных функцией вычислений. Функции, определенные в инструкциях `class`, называются *методами*. Специфика методов обсуждается в разделе "Связанные и несвязанные методы" главы 4; вместе с тем общее рассмотрение функций, приведенное в этом разделе, относится также и к методам.

В Python функции — это объекты (значения), которые обрабатываются наравне с другими объектами. Таким образом, одну функцию можно передавать в качестве аргумента другой функции во время ее вызова. Точно так же функция может вернуть другую функцию в качестве результате вызова. Функция, как и любой другой объект, может быть связана с переменной, может содержаться в контейнере в качестве элемента и может служить атрибутом объекта. Кроме того, функции могут использоваться в качестве ключей в словарях. Например, если вам нужно организовать быстрый поиск функции, обратной заданной, можно определить словарь, ключами и значениями которого являются функции, а затем сделать словарь двунаправленным. Эта идея проиллюстрирована ниже на примере кода, в котором используются функции из модуля `math`, описанного в разделе "Модули `math` и `cmath`" главы 15.

```
def make_inverse(inverse_dict):
    for f in list(inverse_dict):
        inverse_dict[inverse_dict[f]] = f
    return inverse_dict
inverse = make_inverse({sin:asin, cos:acos, tan:atan, log:exp})
```

В силу описанного выше поведения функций Python как обычных объектов они относятся к категории *объектов первого класса*.

Инструкция def

Чаще всего функции определяются с помощью инструкции `def`. Инструкция `def` — это составная инструкция, включающая одно предложение и имеющая следующий синтаксис.

```
def имя_функции(параметры):
    инструкции
```

Здесь *имя_функции* — это идентификатор, т.е. переменная, которая при выполнении инструкции `def` связывается со значением (ей присваивается значение) в виде объекта функции.

Параметры — это необязательный список идентификаторов, которые связываются со значениями, предоставляемыми при вызове функции; в отношении последних мы используем термин "аргументы", чтобы провести различие между определениями (параметры) и вызовами (аргументы) функций. В простейшем случае функция может вообще не иметь параметров; это означает, что при ее вызове она не получает

никаких аргументов. В определении такой функций круглые скобки после ее имени остаются пустыми, и такими же они должны оставаться при ее вызове.

Список параметров функции, имеющей аргументы, содержит один или несколько идентификаторов, разделенных запятыми (`,`). В этом случае при каждом вызове функции ей предоставляются значения (*аргументы*), соответствующие параметрам, которые были указаны в определении функции. Параметры — локальные переменные функции (о чем говорится далее), при каждом вызове которой они связываются в ее пространстве имен с соответствующими значениями, предоставляемыми вызывающим кодом в качестве аргументов.

При выполнении инструкции `def` непустой блок инструкций, так называемое *тело функции*, не выполняется. Он выполняется впоследствии при каждом вызове данной функции. Инструкция `return`, о которой вскоре пойдет речь, может встречаться в теле функции несколько раз или вообще отсутствовать.

Вот пример простой функции, которая возвращает значение, в два раза превышающее то, которое передается ей при вызове.

```
def twice(x):
    return x*2
```

Обратите внимание на то, что в качестве аргумента здесь может выступать любой объект, допускающий выполнение операции умножения на 2, поэтому при вызове данной функции ей можно передать число, строку, список или кортеж, и в каждом из этих случаев она вернет объект того же типа, что и аргумент.

Параметры

Параметры (более строгий термин — *формальные параметры*) определяют имена значений, передаваемых функции при ее вызове, а иногда задают значения, используемые по умолчанию. Каждый раз, когда вызывается функция, каждое из этих имен связывается со значениями в новом локальном пространстве имен; когда функция возвращает значение или завершает выполнение иным способом, это пространство имен уничтожается (значения, возвращаемые функцией, могут сохраняться вызывающим кодом, связывающим возвращаемый функцией результат). Параметры, указанные с помощью простых идентификаторов, — это так называемые *позиционные параметры* (другое название — *обязательные параметры*). Соответствующие значения (аргументы) для каждого обязательного (позиционного) параметра должны предоставляться при каждом вызове функции.

В списке параметров, разделенных запятыми, вслед за позиционными параметрами (если они предусмотрены для данной функции) можно указать нуль или несколько *именованных параметров* (другое название — *необязательные параметры*), каждый из которых подчиняется следующему синтаксису:

идентификатор=выражение

Инструкция `def` вычисляет каждое такое выражение и сохраняет ссылку на полученное значение, которое носит название *значение по умолчанию* для данного параметра, наряду с другими атрибутами объекта функции. Если при вызове функции значение аргумента, соответствующего именованному параметру, не было задано, то при выполнении данного вызова функции идентификатор параметра связывается с его значением по умолчанию.

Обратите внимание на то, что Python вычисляет каждое значение по умолчанию только один раз, когда выполняется инструкция `def`, а не при каждом вызове функции. В частности, это означает, что всякий раз, когда вызывающий код не предоставляет значение соответствующего аргумента, с именованным параметром всегда связывается один и тот же объект: значение по умолчанию.

Остерегайтесь использования изменяемых значений по умолчанию

Такие изменяемые объекты, как списки, в случае задания их в качестве значения по умолчанию могут изменяться всякий раз, когда функция вызывается без явного задания аргумента для соответствующего параметра. Обычно это не соответствует тому поведению функции, которое вам нужно (см. пример ниже).

Вы можете столкнуться с некоторыми сюрпризами, если значением по умолчанию является изменяемый объект и тело функции изменяет параметр.

```
def f(x, y=[]):
    y.append(x)
    return y, id(y)
print(f(23))                # вывод: ([23], 4302354376)
print(f(42))                # вывод: ([23, 42], 4302354376)
```

Второй вызов выводит значение `[23, 42]`, поскольку первый вызов `f` изменяет значение по умолчанию `y`, первоначально заданное в виде пустого списка `[]`, присоединяя к нему значение `23`. Значения `id` подтверждают, что оба вызова возвращают один и тот же объект. Если вы хотите, чтобы идентификатор `y` связывался с новым пустым списком каждый раз, когда функция `f` вызывается с одним аргументом, необходимо использовать следующую идиому.

```
def f(x, y=None):
    if y is None: y = []
    y.append(x)
    return y, id(y)
print(f(23))                # вывод: ([23], 4302354376)
prinf(f(42))                # вывод: ([42], 4302180040)
```

Разумеется, возможны случаи, когда вам действительно нужно, чтобы заданное по умолчанию значение параметра изменилось. Чаще всего этим пользуются для кеширования данных, чтобы по возможности избежать повторного выполнения трудоемких вычислений, что продемонстрировано в следующем примере.

```
def cached_compute(x, _cache={}):
    if x not in _cache:
        _cache[x] = costly_computation(x)
    return _cache[x]
```

Однако подобное кеширование (также называемое *мемоизацией*) обычно лучше реализовывать посредством декорирования функции с помощью функции `functools.lru_cache`, описанной в табл. 7.4.

В конце списка параметров можно использовать любую из двух специальных форм: `*args` и `**kwds`[9]. В этих именах нет ничего особенного — можете использовать вместо них любой идентификатор. Имена `args` и `kwds` (или `kwargs`) — это всего лишь популярные метки, используемые для данных целей, и они ничем не лучше, чем, скажем, имена `a` и `k`. Если присутствуют обе формы, та из них, которая отмечена двумя звездочками, должна быть указана второй.

Форма `*args` определяет, что любые дополнительные позиционные аргументы, заданные при вызове функции (т.е. не указанные явно в сигнатуре функции, определение которой приведено в разделе "Сигнатура функции"), объединяются в (возможно, пустой) кортеж и связываются с именами аргументов в пространстве имен вызываемой функции. Точно так же форма `**kwds` определяет, что любые дополнительные именованные аргументы (т.е. именованные аргументы, не указанные явно в сигнатуре функции, определение которой приведено в разделе "Сигнатура функции"), объединяются в (возможно, пустой) словарь[10], элементами которого являются имена и значения этих аргументов, и связываются с именем `kwds` в пространстве имен вызываемой функции (позиционные и именованные аргументы обсуждаются в разделе "Вызов функций").

Ниже приведен пример функции, которая принимает произвольное количество позиционных аргументов и возвращает их сумму (попутно демонстрируется использование идентификатора, отличного от `*args`).

```
def sum_args(*numbers):
    return sum(numbers)
print(sum_args(23, 42))      # вывод: 65
```

Сигнатура функции

Количество параметров функции вместе с их именами, количество обязательных параметров, а также информация о наличии (в конце списка параметров) любой из специальных форм (или обеих форм одновременно), отмеченных одной или двумя звездочками, совокупно составляют спецификацию, известную под названием *сигнатура функции*. Сигнатура функции определяет допустимые способы ее вызова.

---

[9] Также записывается в виде `**kwargs`.

[10] **Новое в версии 3.6:** теперь отображение, с которым связываются дополнительные именованные аргументы `kwds`, сохраняет тот порядок их следования, в котором они были заданы при вызове функции.

Параметры, указываемые только как именованные (версия v3)

Только в версии v3: инструкция `def` предоставляет дополнительную возможность указывать параметры, которым при вызове функции *должны* соответствовать именованные параметры вида *идентификатор=выражение* (раздел "Виды аргументов"), если они вообще имеются. Эти параметры называются *только именованными* (keyword-only). Если эти параметры имеются, то они должны указываться вслед за параметрами `*args` (при их наличии) и перед параметрами `**kwargs` (при их наличии).

Каждый из параметров, определенных как только именованные, может указываться в списке параметров либо с помощью простого идентификатора (в этом случае его задание при вызове функции является обязательным, чтобы обеспечить передачу соответствующего именованного аргумента), либо в форме *идентификатор=значение_по_умолчанию* (в этом случае передача соответствующего аргумента при вызове функции не является обязательной, но если вы его задаете, то *должны* передавать его в форме именованного аргумента, а не позиционного). Для каждого из различных параметров, указанных как только именованные, можно использовать любую из этих форм. Чаще всего в начале спецификации таких параметров, если таковые имеются, для облегчения чтения кода используют простые идентификаторы, вслед за которыми, если в этом есть необходимость, указывают параметры в форме *идентификатор=значение_по_умолчанию*, однако это не является требованием языка.

Все определения параметров, указываемых как только именованные, должны указываться *после* специальной формы `*args`, если она имеется. Если же такая форма отсутствует, то начало последовательности определений параметров, указываемых как только именованные, обозначается нулевым параметром в виде одиночного символа `*` (звездочка), которому не соответствуют никакие аргументы.

```
def f(a, *, b, c=56): # b и c - только именованные
    return a, b, c
f(12,b=34)            # возвращает (12, 34, 56); c не является
                      # обязательным, имеет значение по умолчанию
f(12)                 # генерирует исключение TypeError
# сообщение об ошибке: missing 1 required keyword-only
# argument: 'b'
```

Если вы также определяете специальную форму `**kwds`, то она должна располагаться в конце списка параметров (после определений параметров, указываемых как только именованные, если таковые имеются).

```
def g(x, *a, b=23, **k): # b - только именованный
    return x, a, b, k
g(1, 2, 3, c=99)         # возвращает (1, (2, 3), 23, {'c': 99})
```

Атрибуты объектов функций

Инструкция `def` устанавливает некоторые атрибуты объекта функции. Атрибут `__name__` ссылается на идентификатор строки, предоставленной в качестве имени функции в инструкции `def`. Вы можете повторно связать этот атрибут с любым строковым значением, но попытка освободить его (открепить) приведет к возбуждению исключения. Атрибут `__defaults__`, который можно повторно связывать или освобождать, ссылается на кортеж значений по умолчанию для необязательных параметров (пустой кортеж, если функция не имеет необязательных параметров).

Строки документирования

Другим атрибутом функции является *строка документирования*, также известная под названием *docstring* (от *documentation string*). Вы можете использовать или повторно связывать атрибут строки документирования в виде `__doc__`. Если первой инструкцией в теле функции является строковый литерал, то компилятор связывает этот литерал в качестве атрибута строки документирования. Аналогичное правило действует в отношении классов (раздел "Строка документирования класса" в главе 4) и модулей (раздел "Строки документирования модулей" в главе 6). Как правило, строки документирования занимают несколько физических строк, поэтому обычно их определяют посредством литеральной формы с тройными кавычками.

```
def sum_args(*numbers):
    """Возвращает сумму нескольких числовых аргументов.

    Список аргументов может либо быть пустым, либо содержать
    несколько чисел.
    Результат является суммой чисел.
    """
    return sum(numbers)
```

Строки документирования должны быть неотъемлемой частью любого написанного вами кода. Они играют роль, аналогичную роли комментариев, но имеют более широкую сферу применения, поскольку остаются доступными во время выполнения программы (если только не были отключены с помощью соответствующей опции командной строки при вызове *python*: `python -OO`; см. раздел "Синтаксис и параметры командной строки" в главе 2). Среды разработки и утилиты могут использовать строки документирования объектов функций, классов и модулей для того, чтобы напомнить программисту порядок использования этих объектов. Модуль `doctest` (раздел "Модуль `doctest`" в главе 16) упрощает проверку того, что представленный в строке документирования образец кода является точным и корректным и остается таким же по мере редактирования кода и документации и их сопровождения.

Чтобы ваши строки документирования были максимально полезными, следуйте нескольким простым правилам. Первая строка должна содержать краткое описание назначения функции, должна начинаться с прописной буквы и заканчиваться точкой.

Имя функции не должно в ней упоминаться, если только оно не является словом обычного языка, которое естественным образом удачно включается в краткое описание назначения функции. Если строка документирования включает несколько строк, то вторая строка должна оставаться пустой, а следующие строки должны быть сформированы в один или несколько разделенных пустыми строками абзацев, содержащих описание параметров функции, предусловий, возвращаемого значения и побочных эффектов (если таковые имеются). В конце строки документирования могут размещаться необязательные дополнительные пояснения, библиографические ссылки и примеры использования (которые должны проверяться вами с помощью модуля `doctest`).

Другие атрибуты объектов функций

Кроме своих предопределенных атрибутов, объект функции может иметь другие произвольные атрибуты. Чтобы создать атрибут объекта функции, свяжите значение с соответствующей ссылкой на атрибут с помощью инструкции присваивания после выполнения инструкции `def`. Например, функция может подсчитать, сколько раз она вызывалась.

```
def counter():
    counter.count += 1
    return counter.count
counter.count = 0
```

Имейте в виду, что такой способ использования атрибутов не является общепринятым. В большинстве случаев, если вы хотите сгруппировать состояние (данные) и поведение (код), вам следует задействовать объектно-ориентированные механизмы, рассмотрению которых посвящена глава 4. Однако возможность связывания атрибутов с функцией иногда может быть очень кстати.

Аннотирование функций и подсказки типов (только в версии v3)

Только в версии v3: каждый из параметров в предложении `def` можно *аннотировать* произвольным выражением, т.е. везде в списке параметров `def`, где можно использовать идентификатор, вы можете, если хотите, использовать форму *идентификатор*`:`*выражение*, и значение выражения становится *аннотацией* для этого параметра.

Для аннотирования возвращаемого значения функции можно использовать форму `->`*выражение* между закрывающей круглой скобкой `)` предложения `def` и двоеточием `:`, которым оно заканчивается; значение выражения становится аннотацией для имени `'return'`.

```
>>> def f(a:'foo', b)->'bar': pass
...
>>> f.__annotations__
{'a': 'foo', 'return': 'bar'}
```

Как показано в этом примере, атрибут `__annotations__` объекта функции является словарем, отображающим каждый аннотированный идентификатор в соответствующую аннотацию.

Вы можете свободно использовать аннотации любым удобным для вас способом. Сам по себе Python никак не использует соответствующую информацию и лишь позволяет создавать атрибут `__annotations__`.

Аннотации ориентированы на будущее использование в качестве средства, с помощью которого гипотетические инструменты сторонних производителей смогут обеспечивать более тщательную статическую проверку аннотированных функций. (Если у вас возникнет желание поэкспериментировать с такими статическими проверками, рекомендуем посетить страницу проекта *mypy*: `mypy-lang.org`).

Подсказки типов (только в Python 3.5)

В версии Python 3.5 с целью обеспечения поддержки гипотетических инструментов сторонних производителей в будущем и стандартизации способов аннотирования подсказок типов был введен сложный набор соглашений об использовании комментариев, а в стандартную библиотеку был включен новый экспериментальный модуль `typing` (вновь подчеркнем, что в настоящее время Python никак не использует эту информацию, и ее стандартизация рассчитана лишь на будущие разработки сторонних производителей).

Экспериментальным (provisional) называют модуль, который может быть подвергнут существенным изменениям или вообще исчезнуть со сменой младшего номера версии (поэтому существует вероятность того, что в версии Python 3.7 модуль `typing` будет исключен из стандартной библиотеки, а если и останется, то его содержимое будет изменено). Этот модуль все еще предназначен лишь для использования в экспериментальных, а не производственных целях.

Имея в виду ограниченную применимость модуля `typing` в настоящее время, мы не будем рассматривать его в этой книге. Для получения более подробной информации об этом модуле обратитесь к документу **PEP 484** (`https://www.python.org/dev/peps/pep-0484/`), где вы найдете множество дополнительных ссылок. Независимый проект *mypy* (`mypy-lang.org`) обеспечивает полную совместимость кода Python v3 с требованиями документа **PEP 484**.

Новое в версии 3.6: аннотирование типов

Несмотря на то что соглашения, определенные в документе **PEP 484**, могут быть полезными для статического анализа кода на Python с использованием соответствующих инструментов от независимых производителей, основное внимание в этом документе уделено аннотированию функций, и он не включает поддержку синтаксиса, обеспечивающего явное маркирование принадлежности переменных определенному типу. Этот тип аннотирования реализован в нем посредством комментариев. Теперь синтаксис для аннотирования типов переменных определен в документе **PEP 526**.

Гвидо ван Россум привел пример того, насколько полезным может быть аннотирование типов, скажем, при переносе унаследованных кодовых баз большого объема на другие платформы (`https://mail.python.org/pipermail/python-dev/2016-September/146282.html`). Аннотации типов могут быть сложными для новичков, но это необязательное средство, и при необходимости обращению с аннотациями можно научить тех, кто уже имеет некоторый опыт работы с Python. Вы по-прежнему можете использовать аннотирование типов посредством комментариев в стиле **PEP 484**, однако вполне вероятно, что этот способ в ближайшем будущем будет вытеснен аннотациями в стиле PEP 526. (На момент написания книги независимый проект `mypy` лишь частично поддерживал код на Python 3.6, совместимый с **PEP 526**.)

Инструкция `return`

В Python инструкция `return` может встречаться лишь в теле функции, и за ней может следовать необязательное выражение. Выполнение инструкции `return` приводит к прекращению работы функции, а значение выражения, если оно имеется, возвращается в качестве результата. Если работа функции завершается достижением конца ее тела или посредством выполнения инструкции `return`, не содержащей выражения, то она возвращает значение `None` (разумеется, функция может вернуть это значение посредством инструкции `return None`).

Рекомендации по стилю написания инструкций `return`

В соответствии с установившейся практикой программирования *избегайте* завершать тело функции инструкцией `return`, не содержащей выражения. Если некоторые из инструкций `return` в теле функции содержат выражения, то выражения должны содержаться в каждой инструкции `return` данной функции. Использование инструкции `return None` оправданно лишь в тех случаях, когда это делается в интересах соблюдения данных рекомендаций по стилевому оформлению кода. Python не возводит эти соглашения в ранг обязательных требований, но, соблюдая их, вы сделаете свой код более понятным и удобочитаемым.

Вызов функций

Вызов функции осуществляется с помощью выражения, имеющего следующий синтаксис:

`объект_функции``(`*`аргументы`*`)`

Здесь *`объект_функции`* означает любую ссылку на объект функции (или любой другой вызываемый объект); чаще всего в качестве такой ссылки используют имя функции. Круглые скобки обозначают саму операцию вызова. В простейшем случае обозначение *`аргументы`* представляет последовательность из нуля или нескольких

разделенных запятыми (,) выражений, предоставляющих значения для соответствующих параметров функции. При вызове функции параметры связываются со значениями аргументов и выполняется тело функции, причем значением выражения вызова функции является результат, возвращаемый функцией.

Вызывая функцию, не забывайте указывать пару круглых скобок после ее имени

Одно лишь *указание* имени функции (или другого вызываемого объекта) еще *не* равнозначно ее вызову. Чтобы *вызвать* функцию (или другой объект) без аргументов, *необходимо* указать пару круглых скобок `()` после ее имени (или любой другой ссылки на вызываемый объект).

Семантика передачи аргументов

В рамках традиционной терминологии все аргументы в Python передаются *по значению* (хотя в соответствии с современной терминологией точнее было бы говорить о передаче аргументов *по ссылке на объект*; некоторым авторам этой книги также нравится синонимичный термин *вызов по разделяемой ссылке*). Например, если вы передаете переменную в качестве аргумента, то Python передает функции объект (значение), на который в данный момент ссылается переменная, а не саму переменную, и это значение связывается с именем параметра в пространстве имен вызова функции. Таким образом, функция не может изменить связывание переменных вызывающего кода. Если в качестве аргумента передается изменяемый объект, то функция может запросить изменение этого объекта, поскольку Python передает сам объект, а не его копию. Изменение связывания переменной и изменение объекта — это совершенно разные вещи.

```
def f(x, y):
    x = 23
    y.append(42)
a = 77
b = [99]
f(a, b)
print(a, b)                    # вывод: 77 [99, 42]
```

Как следует из этих результатов, переменная `a` осталась связанной со значением `77`. Тот факт, что функция `f` привязала свой параметр `x` к значению `23`, никак не сказался на вызывающем коде и, в частности, на связывании его переменной, которая использовалась для передачи значения `77` в качестве параметра. В то же время мы видим, что переменная `b` теперь связана со списком `[99, 42]`. Она по-прежнему связана с тем же объектом, с которым была связана до вызова функции, но сам объект изменился, поскольку к объекту этого списка было присоединено значение `42`. В обоих случаях функция `f` не изменила связывания, существующие в вызывающем коде, как не изменила она и число `77`, поскольку числа — неизменяемые объекты. Однако `f` может изменить объект списка, поскольку списки — изменяемые объекты. В данном случае

функция f изменила объект списка, который был передан ей вызывающим кодом в качестве второго аргумента, вызвав метод append данного объекта.

Виды аргументов

Аргументы, которые являются всего лишь выражениями, называются *позиционными*. Каждый позиционный аргумент предоставляет значение для параметра, соответствие с которым устанавливается на основании его позиции (порядка следования) в определении функции. Количество позиционных аргументов не обязано совпадать с количеством позиционных параметров, — если позиционных аргументов больше, чем позиционных параметров, то позиционные аргументы связываются с именованными параметрами, при условии, что таковые имеются, в соответствии с их позициями, а затем (если сигнатура функции включает специальную форму *args) с кортежем, связанным с args. Приведем соответствующий пример.

```
def f(a, b, c=23, d=42, *x):
    print(a, b, c, d, x)
f(1,2,3,4,5,6)              # вывод: (1, 2, 3, 4, (5, 6))
```

В вызове функции вслед за нулем или большим количеством позиционных параметров могут быть указаны нуль или более именованных аргументов, каждый из которых подчиняется следующему синтаксису:

идентификатор=выражение

В отсутствие параметра **kwds идентификатор должен совпадать с одним из имен параметров, используемых в инструкции def данной функции. Выражение предоставляет значение для параметра с этим именем. Многие встроенные функции не принимают именованных аргументов: при вызове таких функций должны использоваться только позиционные параметры. В то же время функции, написанные на Python, принимают как позиционные, так и именованные параметры, и их можно вызывать различными способами. В отсутствие необходимого количества позиционных аргументов именованным аргументам могут соответствовать позиционные параметры.

Вызов функции должен предоставлять, будь то посредством позиционных или именованных аргументов, ровно по одному значению для каждого из обязательных параметров и по нулю или одному значению для каждого из необязательных параметров.

```
def divide(divisor, dividend):
    return dividend // divisor
print(divide(12, 94))                  # вывод: 7
print(divide(dividend=94, divisor=12)) # вывод: 7
```

Как видите, оба вызова функции divide приводят к одинаковым результатам. Вы можете передавать именованные аргументы в тех случаях, когда считаете, что обозначение роли каждого аргумента и управление порядком следования аргументов позволит сделать код более понятным.

Именованные аргументы часто используются для связывания некоторых необязательных параметров с конкретными значениями, одновременно предоставляя возможность другим необязательным параметрам принимать значения по умолчанию.

```
def f(middle, begin='init', end='finis'):
    return begin+middle+end
print(f('tini', end=''))                    # вывод: inittini
```

Используя именованный аргумент `end=''`, вызывающий код определяет значение (пустую строку `''`) для третьего параметра функции `f`, `end`, но позволяет ее второму аргументу, `begin`, использовать значение по умолчанию, т.е. строку `'init'`. Обратите внимание на то, что эти аргументы могут передаваться как позиционные, несмотря на то, что соответствующие параметры определены как именованные. Вот пример использования предыдущей функции:

```
print(f('a','c','t'))                       # вывод: cat
```

Вызывая функцию, можно использовать в конце списка аргументов любую или одновременно обе специальные формы `*seq` и `**dct`. Если имеются обе формы, та из них, которая помечена двумя звездочками, должна указываться последней. Форма `*seq` передает элементы объекта `seq` функции в качестве позиционных аргументов (после передачи обычных позиционных аргументов, если таковые имеются, которые передаются с использованием обычного синтаксиса). В качестве объекта `seq` можно использовать любой итерируемый объект. Форма `**dct` передает функции элементы объекта `dct` в качестве именованных аргументов, где объектом `dct` должен быть словарь, все ключи которого являются строками. Ключ каждого элемента словаря — это имя параметра, а значение — значение аргумента.

Иногда вам может понадобиться передать аргумент в форме `*seq` или `**dct`, если параметры определены с использованием аналогичных форм, о чем шла речь в разделе "Параметры". Например, в процессе использования функции `sum_args`, определенной ранее в этом разделе (и вновь представленной здесь), у вас может возникнуть потребность в выводе суммы всех значений словаря `d`. Это можно легко сделать, используя форму `*seq`.

```
def sum_args(*numbers):
    return sum(numbers)
print(sum_args(*d.values()))
```

(Разумеется, то же самое можно сделать более непосредственным и простым способом с помощью вызова `print(sum(d.values()))`.)

Аргументы в форме `*seq` или `**dct` можно передавать и в тех случаях, когда эти формы отсутствуют в сигнатуре функции. Разумеется, в подобных случаях вы должны быть уверены в том, что итерируемый объект `seq` содержит подходящее количество элементов или, соответственно, что в словаре `dct` используется подходящее количество строк идентификаторов в качестве ключей; если эти условия не соблюдаются, генерируется исключение. Как отмечалось в разделе "Параметры, указываемые только как именованные

(версия v3)", в версии v3 параметры, указываемые как только именованные (в отличие от обычных именованных параметров), не могут сопоставляться с позиционными аргументами. Такие параметры должны сопоставляться с именованными аргументами, заданными явно или переданными с использованием формы `**dct`.

В соответствии с документом **PEP 448**, в версии v3 каждая из форм `*seq` и `**dct` может встречаться в вызове функции нуль или более раз.

Пространства имен

Параметры функции плюс любые имена, связанные (посредством операции присваивания или других инструкций, таких как `def`) в теле функции, образуют *локальное пространство имен* функции, также называемое *локальной областью видимости*. Каждая из таких переменных называется *локальной переменной* функции.

Переменные, не являющиеся локальными, называются *глобальными* (в отсутствие вложенных функций, которые мы вскоре обсудим). Глобальные переменные являются атрибутами объекта модуля (рассматриваются в разделе "Атрибуты объектов модулей" главы 6). В тех случаях, когда имя локальной переменной функции совпадает с именем глобальной переменной, это имя ссылается в теле функции не на глобальную переменную, а на локальную. В подобных случаях говорят, что в теле функции локальная переменная скрывает глобальную переменную с тем же именем.

Инструкция `global`

По умолчанию любая переменная, связанная в теле функции, является локальной переменной данной функции. Если в функции возникает необходимость в связывании или повторном связывании некоторой глобальной переменной (что не является хорошей практикой!), то первой инструкцией в теле данной функции должна быть следующая:

`global` *идентификаторы*

Здесь *идентификаторы* — это один или несколько идентификаторов, разделенных запятыми (`,`). Идентификаторы, указанные в инструкции `global`, ссылаются на глобальные переменные (т.е. на атрибуты объекта модуля), которые функции требуется связывать или повторно связывать (т.е. присваивать им значения или изменять их). Например, функцию `counter`, с которой вы уже встречались в разделе "Другие атрибуты объектов функций", можно реализовать, используя инструкцию `global` и глобальную переменную, а не атрибут объекта функции.

```
_count = 0
def counter():
    global _count
    _count += 1
    return _count
```

Если бы не было инструкции `global`, то функция `counter` возбудила бы исключение `UnboundLocalError`, поскольку в этом случае переменная `_count` в теле функции

была бы неинициализированной (несвязанной) локальной переменной. Несмотря на то что инструкция `global` позволяет успешно использовать описанный прием программирования, этот стиль не отличается элегантностью и не рекомендуется к применению. Как уже отмечалось, если у вас возникает необходимость в объединении состояния и поведения, для этого, как правило, лучше всего использовать объектно-ориентированные механизмы, о которых пойдет речь в главе 4.

Избегайте использования инструкции `global`

Никогда не задействуйте инструкцию `global`, если глобальная переменная лишь используется в теле функции (включая изменение объекта, связанного с этой переменной, если объект мутируемый). Применяйте эту инструкцию только тогда, когда тело функции изменяет связывание данной глобальной переменной (обычно посредством присваивания значения имени данной переменной). Хороший стиль программирования предполагает использование инструкции `global` исключительно в случае острой необходимости, поскольку ее наличие заставляет читателей кода вашей программы предполагать, что она включена в программу для каких-то полезных целей. В частности, если без нее невозможно обойтись, располагайте ее в теле функции в качестве первой инструкции.

Вложенные функции и вложенные области видимости

Инструкция `def`, находящаяся в теле некоторой функции, определяет так называемую *вложенную функцию*, а функция, в тело которой включена инструкция `def`, является *внешней* по отношению к вложенной. Код в теле вложенной функции может обращаться к локальным переменным внешней функции (но не изменять их связывание); такие переменные называются *свободными переменными* вложенной функции.

Во многих случаях проще всего обеспечить доступ вложенной функции к значению, не полагаясь на вложенные области видимости, а явно передавая значение в качестве аргумента функции. Если потребуется, значение аргумента можно связать во время выполнения инструкции `def`, используя данное значение в качестве значения по умолчанию для необязательного аргумента.

```
def percent1(a, b, c):
    def pc(x, total=a+b+c):
        return (x*100.0) / total
    print('Процентные доли:', pc(a), pc(b), pc(c))
```

А вот пример реализации той же функциональности за счет использования вложенных областей видимости.

```
def percent2(a, b, c):
    def pc(x):
        return (x*100.0) / (a+b+c)
    print('Процентные доли:', pc(a), pc(b), pc(c))
```

В данном конкретном случае функция `percent1` обладает некоторым, пусть даже незначительным, преимуществом: сумма `a+b+c` вычисляется только один раз, в то время как в функции `pc`, являющейся внутренней по отношению к функции `percent2`, она вычисляется три раза. В то же время, если внешняя функция изменяет связывание локальных переменных в промежутках между вызовами вложенной функции, то проведение повторных вычислений может оказаться необходимым. Вы должны осознанно подходить к выбору одного из этих двух подходов и использовать тот из них, который является наиболее оптимальным в каждом конкретном случае.

Вложенная функция, получающая значения из внешних локальных переменных, называется *замыканием*. Создание замыкания иллюстрирует следующий пример.

```
def make_adder(augend):
    def add(addend):
        return addend+augend
    return add
```

Иногда замыкания могут выступать в качестве исключения из общего правила, согласно которому объектно-ориентированные механизмы, рассмотренные в главе 4, предлагают наилучшие способы объединения данных и кода. В тех случаях, когда вам нужно конструировать вызываемые объекты таким образом, чтобы некоторые параметры имели фиксированные значения во время создания объектов, замыкания могут обеспечить более простой и эффективный способ решения этой задачи, чем классы. Например, результатом вызова `make_adder(7)` является функция, которая принимает единственный аргумент и прибавляет `7` к этому аргументу. Внешняя функция, возвращающая замыкание, является "фабрикой" элементов, принадлежащих к семейству функций, которые различаются определенными параметрами, такими, как значение аргумента `augend` из предыдущего примера. Часто это может содействовать устранению дублирования кода.

Только в версии v3: ключевое слово `nonlocal` действует аналогично ключевому слову `global`, но ссылается на имя из пространства имен некоторой внешней в лексическом смысле функции. Если оно встречается в определении функции, характеризующейся многократной глубиной вложения, то компилятор будет выполнять поиск имени переменной сначала в пространстве имен ближайшей окружающей функции, затем в области пространства имен функции, окружающей ближайшую окружающую функцию, и т.д. до тех пор, пока не найдет имя или, в случае отсутствия других окружающих функций, возбудит исключение.

Ниже приведен пример возможного только в версии v3 подхода к использованию вложенных функций для реализации функциональности "счетчика", которая в предыдущем разделе была реализована сначала с использованием атрибута функции, а затем — с использованием глобальной переменной.

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
```

```
        count += 1
        return count
    return counter

c1 = make_counter()
c2 = make_counter()
print(c1(), c1(), c1())                # вывод: 1 2 3
print(c2(), c2())                      # вывод: 1 2
print(c1(), c2(), c1())                # вывод: 4 3 5
```

Ключевым преимуществом данного подхода по сравнению с предыдущими является то, что вложенные функции, как и ООП-подход, позволяют создать независимые счетчики, в данном случае — `c1` и `c2`, причем каждое замыкание сохраняет собственное состояние и не взаимодействует с другим.

Реализация такого подхода возможна только в версии v3, поскольку он требует использования ключевого слова `nonlocal` для того, чтобы позволить внутренней функции `counter` изменять связывание свободной переменной `count`. В версии v2 для обеспечения такого же эффекта без изменения связывания свободной переменной вам пришлось бы прибегнуть к довольно тонкому приему.

```
def make_counter():
    count = [0]
    def counter():
        count[0] += 1
        return count[0]
    return counter
```

Совершенно очевидно, что вариант кода, реализуемый в версии v3, читается гораздо легче, поскольку не требует использования каких-либо уловок.

Лямбда-выражения

Если тело функции состоит исключительно из инструкции `return` *выражение*, то ее можно заменить так называемым лямбда-выражением, имеющим следующий вид:

`lambda` *параметры*: *выражение*

Лямбда-выражение — это анонимный эквивалент обычной функции, тело которой представлено единственной инструкцией `return`. Обратите внимание на то, что синтаксис лямбда-выражений не предполагает использования ключевого слова `return`. Лямбда-выражение можно использовать везде, где допускается использование ссылки на функцию. Лямбда-выражения могут быть весьма удобными в тех случаях, когда вы хотите использовать предельно простую функцию в качестве аргумента или возвращаемого значения. Ниже приведен пример, в котором лямбда-выражение используется в качестве аргумента встроенной функции `filter` (см. в табл. 7.2).

```
a_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
low = 3
```

```
high = 7
filter(lambda x: low<=x<high, a_list)   # возвращает: [3, 4, 5, 6]
```

(В версии v3 функция `filter` возвращает не список, а итератор.) Как альтернативный вариант, вы можете всегда использовать локальную инструкцию `def` для того, чтобы предоставить объекту функции имя, а затем использовать это имя в качестве аргумента или возвращаемого значения. Ниже предыдущий пример с функцией `filter` реализован с использованием локальной инструкции `def`.

```
a_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
def within_bounds(value, low=3, high=7):
    return low<=value<high
filter(within_bounds, a_list)          # возвращает: [3, 4, 5, 6]
```

Несмотря на то что иногда лямбда-выражения могут быть весьма удобными, обычно вместо них лучше использовать инструкции `def`: последние обладают большей общностью и делают код более удобочитаемым, поскольку вы можете выбрать для функции понятное имя, объясняющее ее назначение.

Генераторы

Если тело функции содержит одно или несколько ключевых слов `yield`, то это так называемый *генератор*, точнее — *функция-генератор*. Вызов генератора не приводит к выполнению тела функции. Вместо этого функция-генератор возвращает специальный объект итератора, так называемый *объект генератора* (который иногда, не совсем правильно, называют просто "генератор"). Этот объект обертывает тело функции, ее локальные переменные (включая параметры) и текущую точку выполнения, которой первоначально является начало функции.

При вызове метода `next` объекта генератора тело функции выполняется от текущей точки до следующего выражения `yield`, которое имеет следующий вид:

`yield` *выражение*

Вполне допустима пустая инструкция `yield`, не содержащая выражения, которая эквивалентна инструкции `yield None`. Когда выполняется инструкция `yield`, выполнение функции приостанавливается, текущая точка выполнения и локальные переменные сохраняются, а выработанное инструкцией `yield` значение выражения возвращается в результате вызова метода `next`. Когда вы вновь вызываете метод `next`, выполнение тела функции возобновляется с той точки, в которой оно было приостановлено, до следующей инструкции `yield`. При завершении выполнения тела функции или выполнении инструкции `return` итератор возбуждает исключение `StopIteration`, указывающее на завершение итерирования. В версии v2 инструкции `return` в генераторе не могут содержать выражений; в версии v3 выражение, указанное в инструкции `return`, если таковое имеется, передается в качестве аргумента результирующему исключению `StopIteration`.

С технической точки зрения в настоящее время, начиная с версии Python 2.5, `yield` — это не инструкция, а выражение, которое возвращает элемент, переданный

методу send объекта генератора: если *g* — объект генератора, то после вызова *g*.send(*значение*) значением yield будет являться *значение*; после вызова next(*g*) значением yield будет являться None. На этом базируется простейший способ реализации в Python так называемых *сопрограмм*, о которых детальнее будет рассказано в разделе "Схожесть генераторов и сопрограмм".

Функции-генераторы удобно задействовать для создания итераторов. Поскольку наиболее распространенным способом применения итераторов являются циклы for, то приведенный ниже код является типичным примером использования генератора (метод next вызывается неявно в инструкции for):

```
for переменная in генератор(аргументы):
```

Предположим, вам нужна последовательность чисел от 1 до N, расположенных сначала в возрастающем, а затем в убывающем порядке. В этом вам поможет генератор.

```
def updown(N):
    for x in range(1, N):
        yield x
    for x in range(N, 0, -1):
        yield x
for i in updown(3): print(i)    # вывод: 1 2 3 2 1
```

Ниже приведен пример генератора, который работает в некоторой степени аналогично встроенной функции range, но возвращает итератор по числам с плавающей точкой, а не по целым числам.

```
def frange(start, stop, stride=1.0):
    while start < stop:
        yield start
        start += stride
```

Функция frange лишь *аналогична* функции range, поскольку для простоты в ней предполагается, что аргументы start и stop являются обязательными, а шаг (stride) — положительным.

Генераторы гибче функций, возвращающих списки. Генератор может вернуть так называемый *неограниченный* итератор, вырабатывающий бесконечный поток результатов (такие итераторы следует использовать только в циклах, прерываемых другими способами, например посредством инструкции break). Кроме того, итераторы, создаваемые объектом генератора, выполняют так называемые *ленивые (отложенные) вычисления*: каждый очередной элемент вычисляется лишь тогда, когда полученный результат будет сразу же использован в других вычислениях, тогда как эквивалентная функция заранее выполняет все вычисления, для сохранения результатов которых может требоваться память большого объема. Поэтому, если все, что вам нужно, — это возможность итерирования по последовательности, то чаще всего вычисление последовательности с помощью генератора оказывается более эффективным по сравнению с функцией, возвращающей список. Если вызывающему коду нужен список всех элементов, полученных с помощью некоторого ограниченного

генератора *g(аргументы)*, то для этого достаточно всего лишь явно запросить создание соответствующего списка:

```
результирующий_список = list(g(аргументы))
```

Инструкция `yield from` (только в версии v3)

В случае многоуровневых итерационных вычислений, включающих инструкции `yield`, в версии v3 существует возможность повысить эффективность и ясность кода, используя инструкции вида `yield from` *выражение*, где *выражение* — итерируемый объект. Это инструкция, которая вырабатывает значения на основе выражения по одному за раз и передает их в вызывающую среду, позволяя избежать многократного использования инструкции `yield`. Рассмотренный ранее генератор последовательностей, первоначально возрастающих, а затем убывающих, может быть упрощен в версии v3.

```
def updown(N):
    yield from range(1, N)
    yield from range(N, 0, -1)
for i in updown(3): print(i) # prints: 1 2 3 2 1
```

В завершение отметим, что (только в версии v3) инструкции `yield from` обеспечивают возможность использования генераторов в качестве полноценных сопрограмм, которые подробно рассматриваются в главе 18.

Выражения-генераторы

Python предлагает еще более легкий способ создания простых генераторов — *выражения-генераторы*. Синтаксис выражений-генераторов аналогичен синтаксису генераторов списков (см. раздел "Генераторы списков"), за исключением того, что выражения-генераторы заключаются не в квадратные (`[]`), а в круглые (`()`) скобки. Семантика выражений-генераторов совпадает с семантикой генераторов списков, за исключением того, что итератор, получаемый с помощью выражения-генератора, вырабатывает по одному элементу за раз, тогда как генератор списка создает список всех результатов, целиком сохраняемый в памяти (поэтому использование выражений-генераторов в соответствующих ситуациях обеспечивает экономию памяти). Например, для получения суммы квадратов всех целых чисел, записываемых с помощью одной цифры, можно было бы использовать выражение `sum([x*x for x in range(10)])`, однако для получения того же результата лучше использовать выражение `sum(x*x for x in range(10))` (отличается от предыдущего отсутствием квадратных скобок), требующее меньшего объема памяти. Обратите внимание на то, что круглые скобки, обозначающие вызов функции, "по совместительству" выполняют обязанности ограничителей выражения (в использовании для этого дополнительной пары круглых скобок нет необходимости).

Схожесть генераторов и сопрограмм

Во всех современных версиях Python (как v2, так и v3) генераторы дополнительно улучшены за счет предоставления возможности организации обратной связи между

вызывающим кодом и генератором путем отправки генератору значения (или исключения) при выполнении инструкции `yield`. Это позволяет генераторам реализовывать сопрограммы, описанные в документе **PEP 342** (`https://www.python.org/dev/peps/pep-0342/`). Основное изменение по сравнению с прежними версиями Python заключается в том, что теперь `yield` является выражением, а не инструкцией, и поэтому имеет значение. При возобновлении выполнения генератора путем вызова метода `next` выражение `yield` имеет значение `None`. Чтобы передать генератору *g* значение *x* (таким образом, чтобы он получил *x* в качестве значения выражения, на котором он был приостановлен), следует вместо вызова `next(g)` использовать вызов `g.send(x)` (вызов `g.send(None)` эквивалентен вызову `next(g)`).

Другие усовершенствования генераторов в современных версиях Python касаются исключений, о чем пойдет речь в разделе "Генераторы и исключения" главы 5.

Рекурсия

Python поддерживает *рекурсию* (т.е. функция Python может вызывать саму себя), однако на глубину вложения рекурсивных вызовов наложены ограничения. По умолчанию Python прерывает рекурсию и возбуждает исключение `RecursionLimitExceeded` (раздел "Классы стандартных исключений" в главе 5), если обнаруживает, что глубина стека рекурсивных вызовов превысила 1000. Для этого предела можно установить другое значение с помощью функции `setrecursionlimit` модуля `sys`, описанной в табл. 7.3.

Однако возможность изменения рекурсивного предела не означает, что вы можете сделать его сколь угодно большим. Абсолютное максимальное значение этого предела зависит от платформы, на которой выполняется программа, и, в частности, от базовой операционной системы и библиотеки C времени выполнения, но в типичных случаях вы можете рассчитывать на рекурсию глубиной порядка нескольких тысяч уровней. При чрезмерно большой установленной глубине рекурсивных вызовов программа может завершиться аварийно. Такие выходящие из-под контроля рекурсии, выполняемые после установки с помощью функции `setrecursionlimit` слишком большого значения для предельного числа вложенных рекурсивных вызовов, превышающего возможности платформы, являются одной из немногих причин возможного краха программы на Python — настоящего глубокого краха, когда не срабатывает даже обычный защитный механизм исключений Python. Поэтому остерегайтесь "лечить" программу, в которой возникает исключение `RecursionLimitExceeded`, путем повышения разрешенной глубины вложения рекурсивных вызовов с помощью функции `setrecursionlimit`. В подобных случаев зачастую лучше изменить организацию программы таким образом, чтобы избавиться от рекурсии или хотя бы постараться уменьшить глубину вложения рекурсивных вызовов, которая нужна программе.

Читателям, знакомым с Lisp, Scheme и другими функциональными языками программирования, особенно важно знать, что Python *не* реализует оптимизацию *хвостовой рекурсии*, играющую важную роль в этих языках. В Python стоимость вызова (как

в отношении затрат времени, так и в отношении затрат памяти) одинакова для рекурсивных и не рекурсивных вызовов и зависит лишь от количества аргументов: она никак не зависит от того, является ли вызов "хвостовым" (т.е. является ли он последней операцией, выполняемой в вызывающем коде) или не хвостовым. По этой причине устранение рекурсий в Python приобретает еще более существенное значение.

Рассмотрим один из классических примеров использования рекурсии: обход дерева узлов. Предположим, у вас имеется древовидная структура данных, каждый узел которой представляет собой кортеж, состоящий из трех элементов. Первый из них, элемент 0 — это полезная нагрузка узла, элемент 1 — кортеж этого же типа для левого дочернего узла или значение `None`, а элемент 2 — аналогичный кортеж для правого дочернего узла. Например, выражение `(23, (42, (5, None, None), (55, None, None)), (94, None,None))` представляет следующее дерево полезных нагрузок.

```
    23
 42    94
5  55
```

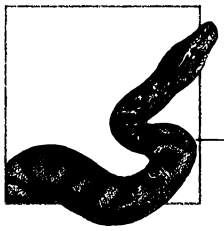
Мы хотим написать функцию-генератор, принимающую в качестве аргумента корень такого дерева, которая совершает обход дерева, вырабатывая значения полезной нагрузки в порядке следования узлов сверху вниз. Очевидным простейшим подходом является использование рекурсии, которую в версии v3 можно организовать следующим образом.

```
def rec(t):
    yield t[0]
    for i in (1, 2):
        if t[i] is not None: yield from rec(t[i])
```

Но если дерево очень глубокое, то рекурсия становится проблематичной. Чтобы избавиться от рекурсии, достаточно обрабатывать собственный стек — список, организованный по принципу LIFO (от англ. *last in, first out* — последним пришел, первым вышел), реализацию которого обеспечивают методы `append` и `pop`. Ниже показано, как это можно сделать в обеих версиях, v2 и v3.

```
def norec(t):
    stack = [t]
    while stack:
        t = stack.pop()
        yield t[0]
        for i in (2, 1):
            if t[i] is not None: stack.append(t[i])
```

Единственное, о чем здесь пришлось дополнительно позаботиться, — это о сохранении того же порядка возврата значений, что и в генераторе `rec`, посредством замены порядка следования индексов `(1, 2)`, в котором анализируются дочерние узлы, на `(2, 1)`, чтобы обратить поведение стека.



4

Объектно-ориентированный Python

Python — объектно-ориентированный язык программирования. В отличие от некоторых других объектно-ориентированных языков, Python не навязывает объектно-ориентированную парадигму: наряду с ней он поддерживает процедурное программирование с его модулями и функциями, поэтому вы вольны выбирать, какой из этих подходов окажется наиболее оптимальным для различных частей вашей программы. Парадигма объектно-ориентированного программирования (ООП) облегчает объединение состояния (данные) и поведения (код) в рамках удобных пакетов функциональности. Она также будет вам полезной, если вы захотите воспользоваться некоторыми объектно-ориентированными механизмами, рассмотренными в этой главе, такими как *наследование* или *специальные методы*. В отсутствие явной необходимости в использовании преимуществ, обеспечиваемых ООП-подходом, процедурная парадигма на основе модулей и функций может обеспечить более простое и удобное решение.

Современная объектная модель Python отличается от той, которая существовала много лет назад. В этой главе описывается исключительно *новая модель*, соответствующая так называемому *новому стилю*, которая отличается от прежней своей простотой, регулярностью и более широкими возможностями, и мы рекомендуем использовать только ее: всякий раз, когда мы говорим о классах или экземплярах, мы имеем в виду классы и экземпляры нового стиля. Однако в интересах обратной совместимости в версии v2 объектной моделью по умолчанию принята *устаревшая объектная модель*, также называемая *классической*, или *модель старого стиля*; в версии v3 в качестве объектной модели используется только модель нового стиля. Для получения информации, касающейся старой модели, вам следует обратиться к онлайн-документации 2002 года (`https://docs.python.org/release/2.1.3/ref/datamodel.html`).

Кроме того, в этой главе обсуждаются *специальные методы* (раздел "Специальные методы"), а также ряд более сложных понятий, таких как *абстрактные базовые классы* (раздел "Абстрактные базовые классы"), *декораторы* (раздел "Декораторы") и *метаклассы* (раздел "Метаклассы").

Классы и экземпляры

Если ваше знакомство с объектно-ориентированным программированием связано с использованием таких языков, как C++ или Java, то вы, вероятно, уже обладаете интуитивным пониманием того, что такое классы и экземпляры: *класс* — это определяемый пользователем тип, который вы *инстанциализируете* для создания *экземпляров*, т.е. объектов данного типа. Python поддерживает данные понятия через объекты своих классов и экземпляров.

Классы Python

Класс — это объект Python, обладающий следующими характеристиками.

- Объект класса можно вызвать так, как если бы это была функция. Такой вызов (другой часто встречающийся термин — *инстанциализация*) возвращает объект, называемый *экземпляром* класса; класс известен как *тип* экземпляра.
- Класс имеет атрибуты с произвольно выбираемыми именами; вы можете связывать эти атрибуты и ссылаться на них.
- Значениями атрибутов класса могут быть *дескрипторы* (включая функции; раздел "Дескрипторы") или обычные объекты данных.
- Атрибуты класса, связанные с функциями, называются *методами* класса.
- Метод может иметь специальное имя, присвоенное Python, с двумя ведущими и двумя замыкающими символами подчеркивания. Если класс предоставляет такие *специальные методы*, то Python неявно вызывает их, когда выполняет различные операции над экземплярами данного класса.
- Класс может *наследовать* атрибуты от других классов, т.е. делегировать объектам других классов поиск атрибутов, не найденных в самом классе.

Экземпляр класса — это объект Python с произвольно именованными атрибутами, которые можно связывать и на которые можно ссылаться. В свою очередь, класс может делегировать поиск атрибутов классам, от которых он наследуется, если таковые имеются.

В Python классы — это объекты (значения), обрабатываемые подобно другим объектам. Таким образом, класс можно передать в качестве аргумента при вызове функции. Точно так же функция может вернуть класс в качестве результата вызова. Класс, как и любой другой объект, можно связать с переменной (локальной или глобальной), элементом контейнера или атрибутом объекта. Классы также могут быть

ключами словаря. Тот факт, что классы Python — это обычные объекты, часто выражают в виде утверждения о том, что "классы — это объекты *первого класса*".

Инструкция `class`

Чаще всего объекты класса создаются с помощью составной инструкции `class`, представленной одним предложением, которая имеет следующий синтаксис.

```
class имя_класса(базовые_классы):
    инструкции
```

Здесь *имя_класса* — это идентификатор, т.е. переменная, которая связывается (возможно, повторно) с объектом класса после завершения выполнения инструкции `class`.

Выражение *базовые_классы* — это последовательность выражений, разделенных запятыми, значениями которых должны быть объекты классов. В разных языках программирования эти классы фигурируют под разными названиями. Вы можете называть их *базовыми классами*, *суперклассами* или *родительскими* по отношению к создаваемому классу — все эти определения означают одно и то же. Вы также можете говорить, что создаваемый класс *наследуется* от своего базового класса, *происходит* от него, *порождается* от него, является его *производным* классом, *подклассом* или *расширяет* его, в зависимости от того, с каким языком программирования вы знакомы. В этой книге мы, как правило, используем термин *расширение*. Производный класс также является *непосредственным подклассом* или *дочерним классом* своих базовых классов. В версии v3 (и только) базовые классы могут включать именованный аргумент `metaclass=...`, определяющий метакласс для данного класса (раздел "Определение метакласса в версии Python v3").

С точки зрения синтаксиса выражение *базовые_классы* не является обязательным. Для указания того, что класс создается без использования базовых классов, можно опустить выражение *базовые_классы* (вместе с окружающими его круглыми скобками, хотя этого можно и не делать) и поместить двоеточие непосредственно вслед за именем класса. Однако в версии v2 в интересах обратной совместимости класс, не имеющий базовых классов, рассматривается как класс старого стиля (если только в нем не был определен атрибут `__metaclass__` рассмотренный в разделе "Определение метакласса в версии Python v2"). Чтобы создать в версии v2 класс *C* без каких-либо "истинных" базовых классов, следует использовать инструкцию `class C(object):`. Поскольку любой тип расширяет встроенный тип `object`, то указание `object` в качестве значения выражения *базовые_классы* всего лишь означает, что *C* является классом нового стиля, а не старого. Если все предки класса (если таковые имеются) являются классами старого стиля и он не определяет атрибут `__metaclass__`, то в версии v2 данный класс соответствует старому стилю. При наличии этого атрибута класс, имеющий базовые классы, рассматривается как класс нового стиля (даже если среди его базовых классов имеются классы как старого, так

и нового стиля). В версии v3 *все* классы являются классами нового стиля (если вы хотите обеспечить обратную совместимость своего кода с версией v2, можете по-прежнему "наследовать от `object`" — вреда от этого не будет). Для единообразия мы всегда указываем в приводимых примерах `object` в качестве базового класса, а не оставляем класс "безбазовым". Если же вы программируете, используя версию v3, и ориентируетесь только на нее, то от такого "фиктивного" наследования лучше отказаться, что придаст более элегантный вид вашему коду.

Отношения подчиненности классов транзитивны: если класс *C1* расширяет *C2*, а *C2* расширяет *C3*, то *C1* расширяет *C3*. Встроенная функция `issubclass(C1, C2)`, принимающая два аргумента, которые оба является объектами классов, возвращает значение `True`, если *C1* расширяет *C2*; в противном случае возвращается значение `False`. Любой класс является собственным подклассом; поэтому вызов `issubclass(C, C)` возвращает значение `True` для любого класса *C*. Влияние базовых классов на функциональность классов, которые их расширяют, рассматривается в разделе "Наследование".

Непустая последовательность выделенных отступами инструкций, следующая за инструкцией `class`, называется *телом класса*. Тело класса выполняется немедленно как часть инструкции `class`. Новый класс не существует, и его имя остается несвязанным до тех пор, пока не закончится выполнение тела этой инструкции. Более подробно о том, что именно происходит в процессе выполнения инструкции `class`, рассказывается в разделе "Как метакласс создает класс".

Наконец, обратите внимание на то, что выполнение инструкции `class` еще не означает немедленного создания какого-либо экземпляра класса. Эта инструкция всего лишь определяет набор атрибутов, разделяемых всеми экземплярами, когда впоследствии они будут создаваться посредством вызова класса.

Тело класса

Как правило, атрибуты класса определяются в его теле. Этими атрибутами могут быть объекты дескрипторов (включая функции) или обычные объекты данных любого типа (атрибутом класса также может быть другой класс, поэтому, например, одна инструкция `class` может быть "вложенной" в другую инструкцию `class`).

Атрибуты объектов `class`

Обычно атрибуты объекта класса определяются путем связывания значения с идентификатором в теле класса.

```
class C1(object):
    x = 23
print(C1.x)              # вывод: 23
```

Объект класса *C1* имеет атрибут *x*, связанный со значением `23`, а выражение *C1.x* является ссылкой на этот атрибут.

Также допускается связывание или повторное связывание атрибутов класса вне тела класса.

```
class C2(object): pass
C2.x = 23
print(C2.x)              # вывод: 23
```

Как правило, читаемость программ улучшается, если связывание, а значит, и создание атрибутов осуществляется только в пределах тела класса. Однако, если вы хотите сохранять информацию о состоянии на уровне класса, а не на уровне экземпляров, вам может понадобиться повторное связывание атрибутов в других местах кода; Python позволяет это делать в случае необходимости. Между атрибутами класса, созданными в теле класса, и атрибутами, созданными или повторно связанными вне тела путем присваивания им значений, нет никакой разницы.

Как вскоре будет обсуждаться, атрибуты класса разделяются всеми его экземплярами.

Некоторые атрибуты класса неявно устанавливаются инструкцией `class`. Атрибут `__name__` — это строка идентификатора имени класса (*имя_класса*), используемая в инструкции `class`. Атрибут `__bases__` — это кортеж объектов классов, переданных инструкции `class` в качестве базовых. В следующем примере используется только что созданный класс `C1`.

```
print(C1.__name__, C1.__bases__)
# вывод: C1 (<type 'object'>,)
```

Кроме того, классы имеют атрибут `__dict__`, представляющий объект отображения, который используется классом для хранения других атрибутов (также известен под названием *пространство имен*). В классах это отображение доступно только для чтения.

В инструкциях, непосредственно входящих в тело класса, для ссылок на атрибуты класса должны использоваться простые, а не уточненные (полные) имена.

```
class C3(object):
    x = 23
    y = x + 22           # следует использовать только x, а не C3.x
```

Однако в инструкциях внутри методов, определенных в теле класса, следует ссылаться на атрибуты только с помощью уточненных имен, а не простых.

```
class C4(object):
    x = 23
    def amethod(self):
        print(C4.x)      # следует использовать C4.x или self.x,
                         # а не просто x!
```

Обратите внимание на то, что ссылки на атрибуты (т.е. выражения наподобие `C.s`) обладают более богатой семантикой по сравнению со ссылками, используемыми

для связывания атрибутов. Эти ссылки подробно обсуждаются в разделе "Основные сведения о ссылках на атрибуты".

Определения функций в теле класса

В большинстве случаев тело класса включает инструкции `def`, поскольку функции (называемые *методами* в этом контексте) являются важными атрибутами для большинства объектов классов. Инструкция `def` в теле класса подчиняется правилам, описанным в разделе "Функции" главы 3. Кроме того, метод, определенный в теле класса, имеет один обязательный параметр с общепринятым именем `self`, ссылающийся на экземпляр, для которого вызывается данный метод. Параметр `self` играет особую роль в вызовах методов, о чем подробнее будет рассказано в разделе "Связанные и несвязанные методы".

Вот пример класса, включающего определение метода.

```
class C5(object):
    def hello(self):
        print('Hello')
```

Класс может определить ряд специальных методов (методы, имена которых содержат два ведущих и два замыкающих символа подчеркивания и которые часто называют "магическими методами"), связанных с выполнением специфических операций над своими экземплярами. Специальные методы подробно обсуждаются в разделе "Специальные методы".

Частные переменные класса

Если в теле класса (или в теле метода, определенного в теле класса) встречается идентификатор с двумя ведущими символами подчеркивания (но *не заканчивающийся* символами подчеркивания), такой как `__идентификатор`, то компилятор Python неявно заменяет его идентификатором `_класс__идентификатор`, где *класс* — это имя класса. Это обеспечивает эффективный способ создания для атрибутов, методов, глобальных переменных и других сущностей "частных" имен внутри данного класса, которые снижают риск их случайного дублирования в других местах кода и, в частности, в подклассах.

В соответствии с общепринятым соглашением идентификаторы, начинающиеся с *одиночного* символа подчеркивания, рассматриваются как частные (закрытые) переменные области видимости их связывания, независимо от того, является эта область классом или не является. Компилятор Python не вынуждает вас следовать данному соглашению. Каждый программист самостоятельно решает для себя, следовать ему или нет.

Строка документирования класса

Если первой инструкцией в теле класса является строковый литерал, то компилятор связывает эту строку с атрибутом `__doc__` данного класса в качестве *строки*

документирования. Более подробную информацию о строках документирования можно получить в разделе "Строки документирования" главы 3.

Дескрипторы

Дескриптор — это любой объект, класс которого предоставляет специальный метод `__get__`. Дескрипторы, являющиеся атрибутами класса, управляют семантикой получения и установки атрибутов экземпляров данного класса. Грубо говоря, когда вы обращаетесь к атрибуту экземпляра, Python получает значение атрибута посредством вызова метода `__get__` для соответствующего дескриптора, если таковой существует.

```
class Const(object):        # перекрытие дескриптора, см. далее
    def __init__(self, value):
        self.value = value
    def __set__(self, *_): # игнорировать попытки установки
        pass
    def __get__(self, *_): # всегда возвращает постоянное значение
        return self.value

class X(object):
    c = Const(23)

x=X()
print(x.c)                  # вывод: 23
x.c = 42
print(x.c)                  # вывод: 23
```

Более подробно об этом можно прочитать в разделе "Основные сведения о ссылках на атрибуты".

Перекрывающие и неперекрывающие дескрипторы

Если класс дескриптора одновременно предоставляет специальный метод `__set__`, то такой дескриптор называют *перекрывающим* (или, в соответствии с устаревшей, но более распространенной и несколько сбивающей с толку терминологией, *дескриптором данных*). Если же класс дескриптора предоставляет метод `__get__` и не предоставляет метод `__set__`, то такой дескриптор называют *неперекрывающим*.

Например, класс объектов функций предоставляет метод `__get__`, но не предоставляет метод `__set__`, поэтому объекты функции являются неперекрывающими дескрипторами. Грубо говоря, если вы присваиваете значение атрибуту экземпляра с помощью соответствующего перекрывающего дескриптора, то Python устанавливает значение этого атрибута, вызывая метод `__set__` для данного дескриптора. Более подробно об этом можно прочитать в разделе "Атрибуты экземпляров объектов".

Экземпляры

Чтобы создать экземпляр класса, вызовите объект класса так, как если бы он был функцией. Каждый вызов возвращает новый экземпляр, типом которого является данный класс:

```
an_instance = C5()
```

Вы можете вызвать встроенную функцию `isinstance(i, C)`, передав ей объект класса в качестве аргумента *C*. Если объект *i* является экземпляром класса *C* или любого его подкласса, то функция `isinstance` возвращает значение `True`; в противном случае возвращается значение `False`.

Метод `__init__`

Если класс определяет или наследует метод `__init__`, то при вызове объекта класса этот метод автоматически вызывается для нового экземпляра, выполняя специфическую для него инициализацию. Передаваемые объекту класса аргументы должны соответствовать параметрам метода `__init__`, за исключением параметра `self`. Рассмотрим в качестве примера следующий класс.

```
class C6(object):
    def __init__(self, n):
        self.x = n
```

Для создания экземпляра класса `C6` можно воспользоваться следующим кодом:

```
another_instance = C6(42)
```

В типичных случаях метод `__init__` содержит инструкции, связывающие атрибуты экземпляра. Метод `__init__` не должен возвращать значение, отличное от `None`; нарушение этого условия приводит к возбуждению исключения `TypeError`.

Основным назначением метода `__init__` является связывание, а значит, создание атрибутов нового экземпляра класса. Как вскоре будет показано, также допускается связывание, повторное связывание и открепление атрибутов экземпляра вне метода `__init__`. Однако вы улучшите читаемость своего кода, если свяжете все атрибуты экземпляра класса первоначально в теле метода `__init__`.

Объекты классов, не имеющих метода `__init__` (и не наследующих его от базовых классов), следует вызывать без указания аргументов. В этом случае новые экземпляры класса не будут иметь специфических для них атрибутов.

Атрибуты экземпляров объектов

Создав экземпляр объекта, вы сможете обращаться к его атрибутам (данным и методам) посредством точечной нотации.

```
an_instance.hello()          # вывод: Hello
print(another_instance.x)    # вывод: 42
```

В Python ссылки на атрибуты, подобные только что приведенным, обладают богатой семантикой, о чем подробнее речь пойдет в разделе "Основные сведения о ссылках на атрибуты".

Объект экземпляра можно снабдить произвольным атрибутом, связав значение со ссылкой на атрибут.

```
class C7: pass
z = C7()
z.x = 23
print(z.x)                    # вывод: 23
```

Теперь объект экземпляра *z* имеет атрибут *x*, с которым связано значение 23, а выражение *z.x* ссылается на этот атрибут. Обратите внимание на то, что специальный метод `__setattr__`, если он предусмотрен, перехватывает любую попытку связывания атрибута. (Метод `__setattr__` описан в табл. 4.1.) Точно так же любая попытка связывания атрибута экземпляра, имя которого соответствует определенному в классе перекрывающему дескриптору, перехватывается методом дескриптора `__set__`: если `C7.x` — перекрывающий дескриптор, то присваивание *z.x*=23 выполнит вызов `type(z).x.__set__(z, 23)`.

В процессе создания экземпляра для него автоматически устанавливаются два атрибута. Для любого экземпляра *z* атрибут *z*.`__class__` — это объект класса, которому принадлежит *z*, а атрибут *z*.`__dict__` — отображение, используемое экземпляром *z* для хранения других своих атрибутов. Приведем соответствующий пример для ранее созданного экземпляра *z*:

```
print(z.__class__.__name__, z.__dict__)     # вывод: C7 {'x':23}
```

Вы можете повторно связать (но не открепить) один или оба этих атрибута, но необходимость в этом возникает лишь в редких случаях.

Для любого экземпляра *z*, любого объекта *x* и любого идентификатора *S* (за исключением идентификаторов `__class__` и `__dict__`) присваивание *z.S*=*x* эквивалентно вызову *z*.`__dict__['S']`=*x* (если только попытка связывания не перехватывается специальным методом `__setattr__` или специальным методом `__set__` перекрывающего дескриптора). Например, можно вновь использовать ранее созданный экземпляр *z*.

```
z.y = 45
z.__dict__['z'] = 67
print(z.x, z.y, z.z)          # вывод: 23 45 67
```

Между атрибутами экземпляра, создаваемыми с помощью операции присваивания, и теми, которые создаются посредством явного связывания записи в словаре *z*.`__dict__`, нет никакой разницы.

Идиома функции-фабрики

Часто требуется создавать экземпляры различных классов в зависимости от выполнения определенных условий или же не создавать новый экземпляр, если уже

существует экземпляр, доступный для повторного использования. Широко распространено заблуждение, в соответствии с которым решение подобных задач можно обеспечить посредством предоставления метода `__init__`, возвращающего требуемый объект, но такой подход нереализуем: при возврате методом `__init__` любого значения, кроме `None`, Python генерирует исключение. Наилучшим способом реализации гибкого процесса создания объекта является использование функции вместо непосредственного вызова объекта класса. Такие функции называются *фабриками*.

Вызов функции-фабрики — гибкий подход: функция может вернуть существующий повторно используемый экземпляр или создать новый посредством вызова подходящего класса. Предположим, вы хотите создать два почти взаимозаменяемых класса (`SpecialCase` и `NormalCase`) и гибко генерировать экземпляры каждого из них в зависимости от значения аргумента. Именно такое поведение реализует функция `appropriate_case` в приведенном ниже "игрушечном" примере (роль параметра `self` обсуждается в разделе "Связанные и несвязанные методы").

```
class SpecialCase(object):
    def amethod(self): print('special')
class NormalCase(object):
    def amethod(self): print('normal')
def appropriate_case(isnormal=True):
    if isnormal: return NormalCase()
    else: return SpecialCase()
aninstance = appropriate_case(isnormal=False)
aninstance.amethod()                          # вывод: special
```

Метод __new__

Каждый класс содержит (или наследует) метод `__new__` (методы классов рассматриваются в разделе "Методы класса"). Каждый раз, когда вы создаете новый экземпляр класса *C* посредством вызова `C(*args,**kwds)`, сначала автоматически вызывается метод `C.__new__(C,*args,**kwds)`. Возвращаемое методом `__new__` значение *x* Python использует в качестве вновь созданного экземпляра. Затем Python вызывает метод `C.__init__(x,*args,**kwds)`, но только в том случае, если *x* действительно является экземпляром класса *C* или любого из его подклассов (в противном случае *x* будет находиться в том состоянии, в каком его оставил метод `__new__`). Вышесказанное можно пояснить на примере инструкции `x=C(23)`, которая эквивалентна следующему коду.

```
x = C.__new__(C, 23)
if isinstance(x, C): type(x).__init__(x, 23)
```

Вызов метода `object.__new__` создает новый, неинициализированный экземпляр класса, переданного методу в качестве первого аргумента. Если класс имеет метод `__init__`, то другие аргументы (в случае их предоставления) игнорируются, но если метод `__new__` получает другие аргументы наряду с первым и класс, представленный первым аргументом, не имеет метода `__init__`, то генерируется исключение. Если

вы хотите перекрыть метод `__new__` в теле класса, то вам не нужно использовать инструкцию `__new__=classmethod(__new__)` или декоратор `@classmethod`, как это обычно делается: в данном контексте Python распознает имя `__new__` и обрабатывает его особым образом. Если впоследствии вам понадобится повторно связать атрибут `C.__new__` вне тела класса `C` (необходимость в этом возникает лишь в исключительно редких случаях), то для этого нужно будет использовать инструкцию `C.__new__=classmethod(`*метод*`)`.

Метод `__new__` способен обеспечить бо́льшую часть гибких возможностей функции-фабрики (см. раздел "Идиома функции-фабрики"). Он может выбирать, что следует сделать: вернуть существующий экземпляр или создать новый, в зависимости от того, что необходимо. Если методу `__new__` нужно создать новый экземпляр, то чаще всего он делегирует выполнение этой операции посредством вызова метода `object.__new__` или метода `__new__` другого суперкласса `C`. Приведенный ниже пример показывает, как можно перекрыть метод класса `__new__` для того, чтобы реализовать версию шаблона проектирования "Singleton" (Одиночка).

```
class Singleton(object):
    _singletons = {}
    def __new__(cls, *args, **kwds):
        if cls not in cls._singletons:
            cls._singletons[cls] = super(Singleton,cls).__new__(cls)
        return cls._singletons[cls]
```

(Встроенная функция `super` рассматривается в разделе "Кооперативный вызов методов суперклассов".)

Любой подкласс класса `Singleton` (не перекрывающий дополнительно метод `__new__`) может иметь ровно один экземпляр. Если такой подкласс определяет метод `__init__`, то он должен позаботиться о безопасности повторных (при каждом запросе на создание объекта) вызовов своего метода `__init__` для одного и только одного экземпляра класса. Это обусловлено тем, что всякий раз, когда вы инстанциализируете любой подкласс класса `Singleton`, содержащего определение метода `__init__`, этот метод вызывается для одного и только одного экземпляра, существующего для данного подкласса класса `Singleton`.

В версии v3 код приведенного примера можно упростить и придать ему более элегантный вид

Версия v3 позволяет написать для данного примера более простой и чистый код. Приведенный выше код примера работает в обеих версиях, v2 и v3. В версии v3 вы можете еще больше упростить код за счет того, что в этом случае вместо передачи суперклассов классу `Singleton` можно вызывать функцию `super` без аргументов.

Основные сведения о ссылках на атрибуты

Ссылка на атрибут — это выражение вида *x.имя*, где *x* — любое выражение, а *имя* — идентификатор, обозначающий *имя атрибута*. Многие разновидности объектов Python имеют атрибуты, но в тех случаях, когда *x* ссылается на класс или экземпляр, ссылки на атрибуты обладают богатой специальной семантикой. Вспомните о том, что методы также являются атрибутами, поэтому все, что говорится об атрибутах вообще, применимо и к вызываемым атрибутам (т.е. методам).

Пусть *x* — экземпляр класса C, являющегося потомком класса B. Оба класса и экземпляр имеют специальные атрибуты (данные и методы).

```
class B(object):
    a = 23
    b = 45
    def f(self): print('method f in class B')
    def g(self): print('method g in class B')
class C(B):
    b = 67
    c = 89
    d = 123
    def g(self): print('method g in class C')
    def h(self): print('method h in class C')
x = C()
x.d = 77
x.e = 88
```

Имена некоторых атрибутов начинаются и заканчиваются двойными символами подчеркивания. *C*.`__name__` — это строка `'C'`, имя класса. *C*.`__bases__` — это кортеж `(B,)`, являющийся кортежем базовых классов *C*. *x*.`__class__` — это класс *C*, т.е. класс, к которому принадлежит *x*. Когда вы обращаетесь к атрибуту с помощью одного из этих специальных имен, поиск атрибута и извлечение значения осуществляются непосредственно в объекте класса или экземпляра. Эти атрибуты нельзя открепить. Им можно на лету присваивать новые значения для изменения базовых классов или имени данного класса либо для изменения класса данного экземпляра, но необходимость в использовании этого более сложного приема возникает лишь в редких случаях.

Как у класса *C*, так и у экземпляра *x* имеется один специальный атрибут: отображение `__dict__`. Все остальные атрибуты класса или экземпляра, за исключением нескольких специальных атрибутов, хранятся в виде элементов атрибута `__dict__` класса или экземпляра.

Получение атрибута с помощью класса

В тех случаях, когда синтаксис *C.имя* используется для обращения к атрибуту объекта класса *C*, процедура поиска атрибута выполняется в два этапа.

1. Если '*имя*' — ключ в словаре *C*.__dict__, то выражение *C*.*имя* извлекает значение *v* из *C*.__dict__['*имя*']. Затем, если *v* — дескриптор (т.е. если type(*v*) предоставляет метод __get__), то значение *C*.*имя* — это результат вызова type(*v*).__get__(*v*, None, *C*). Если же *v* не является дескриптором, то значение *C*.*имя* равно *v*.
2. Если '*имя*' не является ключом в словаре *C*.__dict__, то выражение *C*.*имя* делегирует выполнение поиска базовым классам *C* в том смысле, что обходит в цикле все классы-предки класса *C* и пытается найти имя в каждом из них (используя *порядок разрешения методов*, рассмотренный в разделе "Порядок разрешения методов").

Получение атрибута с помощью экземпляра

В тех случаях, когда синтаксис *x*.*имя* используется для обращения к атрибуту экземпляра *x* класса *C*, процедура поиска атрибута выполняется в три этапа.

1. Если '*имя*' найдено в *C* (или в одном из классов-предков класса *C*) в качестве имени перекрывающего дескриптора *v* (т.е. если type(*v*) предоставляет методы __get__ и __set__), то:
   - значением *x*.*имя* является результат вызова type(*v*).__get__(*v*, *x*, *C*).
2. В противном случае, если '*имя*' является ключом в словаре *x*.__dict__, то:
   - *x*.*имя* извлекает и возвращает значение *x*.__dict__['*имя*'].
3. В противном случае *x*.*имя* делегирует выполнение поиска классу *x* (в соответствии с рассмотренной выше двухэтапной процедурой, используемой для поиска атрибута *C*.*имя*):
   - если найденное значение *v* — дескриптор, то общим результатом поиска атрибута вновь является результат вызова type(*v*).__get__(*v*, *x*, *C*);
   - если найденное значение *v* не является дескриптором, то общий результат поиска атрибута равен *v*.

Если найти атрибут не удается, Python генерирует исключение AttributeError. Однако, если процедура поиска атрибута *x*.*имя* выполняется, когда класс *C* определяет или наследует специальный метод __getattr__, то Python не возбуждает исключение, а вызывает метод *C*.__getattr__(*x*,'*имя*'), который либо возвращает подходящее значение, либо генерирует исключение, обычно AttributeError.

Выведем ссылки на атрибуты, которые были определены ранее:

```
print(x.e, x.d, x.c, x.b, x.a)          # вывод: 88 77 89 67 23
```

Поиск атрибутов *x*.*e* и *x*.*d* успешно завершается на этапе 2 процедуры поиска для экземпляров, поскольку в данном случае дескрипторы отсутствуют, а 'e' и 'd' являются ключами в словаре *x*.__dict__. Поэтому процесс поиска ограничивается этим этапом и возвращает значения 88 и 77. Для трех остальных ссылок процесс

доходит до этапа 3 процедуры поиска для экземпляров, и поиск выполняется в классе `x.__class__` (т.е. `C`). Поиск атрибутов `x.c` и `x.b` успешно завершается на этапе 1 процедуры поиска для класса, поскольку `'c'` и `'b'` являются ключами в словаре `C.__dict__`. Поэтому процесс поиска ограничивается этим этапом и возвращает значения `89` и `67`. Поиск атрибута `x.a` доходит до этапа 2 процедуры поиска для класса и выполняется в классе `C.__bases__[0]` (т.е. `B`). Выражение `'a'` является ключом в словаре `B.__dict__`. На этом процесс поиска атрибута `x.a` успешно завершается, и возвращается значение `23`.

Установка атрибута

Обратите внимание на то, что описанная поэтапная процедура поиска относится лишь к тем случаям, когда вы *обращаетесь* к атрибуту, а не *связываете* его, т.е. не присваиваете ему значение. Когда вы связываете (будь то на уровне класса или экземпляра) атрибут, имя которого не является специальным (и при этом метод `__setattr__` или метод `__set__` перекрывающего дескриптора не перехватывают связывание атрибута экземпляра), вы воздействуете лишь на соответствующую данному атрибуту запись в словаре `__dict__` (класса или экземпляра соответственно). Другими словами, в случае связывания атрибутов никакие процедуры поиска не выполняются, за исключением проверки наличия перекрывающих дескрипторов.

Связанные и несвязанные методы

Метод `__get__` объекта функции может вернуть объект *несвязанного метода* (unbound method; в версии v2), собственно объект функции (в версии v3) или объект *связанного метода* (bound method), обертывающий данную функцию. Основное различие между связанными и несвязанными методами заключается в том, что несвязанный метод (только в версии v2), в отличие от связанного, не ассоциируется с конкретным экземпляром.

В приведенных в предыдущем разделе фрагментах кода атрибуты `f`, `g` и `h` являются функциями, поэтому ссылка на любой из них возвращает объект метода, обертывающий соответствующую функцию. В качестве примера приведем следующую инструкцию:

```
print(x.h, x.g, x.f, C.h, C.g, C.f)
```

Эта инструкция выведет сначала три несвязанных метода, представленных строками вида

```
<bound method C.h of <__main__.C object at 0x8156d5c>>
```

а затем, в версии v2, три несвязанных метода, представленных строками вида

```
<unbound method C.h>
```

или, в версии v3, три объекта функций, представленные строками такого вида:

```
<function C.h at 0x102cabae8>
```

Связанные и несвязанные методы

Мы получаем связанные методы, если ссылка на атрибут относится к экземпляру *x*, и несвязанные (в версии v3 — объекты функций), если ссылка относится к классу *C*.

Поскольку связанный метод уже ассоциирован с конкретным экземпляром, его можно вызвать следующим образом:

```
x.h()                          # вывод: method h in class C
```

Здесь следует обратить внимание на один важный момент, заключающийся в том, что вы не должны передавать первый аргумент, `self`, следуя обычному синтаксису передачи аргументов. Вместо этого связанный метод экземпляра *x* неявно связывает параметр `self` с объектом *x*. Таким образом, тело метода может получать доступ к атрибутам экземпляра как к атрибутам объекта `self`, без явной передачи методу первого аргумента.

Однако несвязанный метод (только в версии v2) не ассоциирован с каким-либо конкретным экземпляром, поэтому при его вызове соответствующий экземпляр должен указываться в качестве первого аргумента, например:

```
C.h(x)                         # вывод: method h in class C
```

Несвязанные методы приходится вызывать намного реже, чем связанные. Важным случаем применения несвязанных методов является доступ к перекрытым (переопределенным) методам, которые обсуждаются в разделе "Наследование", однако даже для этих целей обычно лучше использовать встроенную функцию `super`, о которой пойдет речь в разделе "Кооперативный вызов методов суперклассов". Кроме того, несвязанные методы часто используются в функциях высших порядков. Например, для сортировки списка строк в алфавитном порядке, независимо от регистра, вполне можно использовать выражение вида `los.sort(key=str.lower)`.

Несвязанные методы (только в версии v2)

Как только что обсуждалось, попытка обращения к функциональному атрибуту через имя класса в версии v2 возвращает объект несвязанного метода, обертывающий данную функцию. В дополнение к атрибутам объекта функции, который он оборачивает, несвязанный метод имеет следующие три атрибута: `im_class` — объект класса, предоставляющего метод, `im_func` — обернутая функция и `im_self`, всегда имеющий значение `None`. Все эти атрибуты доступны только для чтения, и любая попытка присвоить им новые значения или открепить приводит к возникновению исключения.

Вы можете вызвать несвязанный метод точно так же, как вызвали бы функцию `im_func`, но при любом таком вызове в качестве первого аргумента должен указываться экземпляр класса `im_class` или его потомка. Другими словами, при вызове несвязанного метода необходимо явно передать ему объект экземпляра,

соответствующий первому формальному параметру обернутой функции (обычно именуемому как self).

Связанные методы

Если при попытке обращения к атрибуту через имя экземпляра процедура поиска находит объект функции, являющийся атрибутом класса данного экземпляра, то для получения значения атрибута она вызывает метод __get__ объекта функции. В этом случае вызов создает и возвращает объект *связанного метода*, обертывающий данную функцию.

Обратите внимание на то, что если процедура поиска атрибута находит объект функции в словаре *x*.__dict__, то операция обращения к атрибуту *не* создает связанный метод. В подобных случаях Python не рассматривает функцию в качестве дескриптора и не вызывает ее метод __get__. Вместо этого значением атрибута становится сам объект функции. Точно так же Python не создает связанных методов для вызываемых объектов, которые не являются обычными функциями, например для встроенных (не пользовательских) функций, поскольку такие вызываемые объекты не являются дескрипторами.

Связанный метод сходен с несвязанным в том, что в дополнение к атрибутам объекта функции, который он обертывает, он имеет три атрибута, доступных только для чтения. Как и в случае несвязанных методов, атрибут im_class — это объект класса, предоставляющего метод, а атрибут im_func — это обернутая функция. Однако в случае объекта связанного метода атрибут im_self относится к экземпляру *x*, из которого вы получили данный метод.

Вы используете связанный метод точно так же, как и его функцию im_func, но при его вызове не предоставляете явно аргумент, соответствующий первому формальному параметру (обычно именуемому как self). Когда вы вызываете связанный метод, он передает im_self в качестве первого аргумента функции im_func, прежде чем передать другие аргументы (если таковые имеются), предоставленные в точке вызова.

Давайте проанализируем низкоуровневые детали того, что происходит при вызове метода с использованием обычного синтаксиса вида *x.имя(аргумент)*.

```
def f(a, b): ... # функция f с двумя аргументами

class C(object):
    name = f
x = C()
```

В данном контексте *x* — экземпляр объекта класса *C*, *имя* — идентификатор, являющийся именем метода объекта *x* (атрибут *C*, значением которого является функция, в данном случае — функция *f*), а *аргумент* — любое выражение. Сначала Python проверяет, является ли '*имя*' в *C* именем атрибута, представляющего перекрывающий дескриптор, но это условие не выполняется: функции действительно являются дескрипторами, поскольку их тип определяет метод __get__, но не перекрывающими

дескрипторами, поскольку их тип не определяет метод `__set__`. Затем Python проверяет, является ли имя ключом в словаре `x.__dict__`, но это не так. Поэтому Python находит имя в классе `C` (все происходит точно так же и в том случае, если имя удается найти в одном из классов, указанных в атрибуте `__bases__` класса `C`). Python замечает, что значение атрибута, т.е объект функции `f`, является дескриптором. Поэтому Python вызывает метод `f.__get__(x, C)`, создающий объект связанного метода, в котором значением атрибута `im_func` является `f`, значением атрибута `im_class` — класс `C`, а значением атрибута `im_self` — экземпляр `x`. Затем Python вызывает этот объект связанного метода, передавая ему *аргумент* в качестве единственного аргумента. Связанный метод вызывает `im_func` (т.е. функцию `f`), используя значение атрибута `im_self` (т.е. `x`) в качестве первого аргумента и, таким образом, делая *аргумент* вторым аргументом. Результирующий общий эффект равносилен такому вызову:

```
x.__class__.__dict__['имя'](x, аргумент)
```

Когда выполняется тело функции связанного метода, его область видимости не соотносится никаким специальным образом с пространствами имен его объекта `self` или какого-либо класса. Его переменные являются локальными или глобальными, точно так же, как и в случае любой другой функции, о чем говорилось в разделе "Пространства имен" главы 3. Они никаким неявным образом не указывают на атрибуты объекта `self` или объекта какого-либо класса. Если методу необходимо обратиться к атрибуту своего объекта `self`, связать или открепить его, то он делает это, используя стандартный синтаксис обращения к атрибутам (например, `self.name`). Чтобы привыкнуть к отсутствию неявно создаваемых областей видимости, вам может потребоваться некоторое время (просто потому, что в этом отношении Python отличается от многих других объектно-ориентированных языков), но это позволяет упростить код, сделать его более понятным и избавиться от потенциальной неоднозначности.

Объекты связанных методов являются объектами первого класса: их можно использовать везде, где допускается использование вызываемого объекта. Поскольку связанный метод хранит ссылки на функцию, которую он обертывает, и объект `self`, для которого он выполняется, связанные методы предлагают мощную и гибкую альтернативу замыканиям (см. раздел "Вложенные функции и вложенные области видимости" в главе 3). Объект экземпляра, класс которого предоставляет специальный метод `__call__` (см. описание в табл. 4.1), предлагает другую действенную альтернативу. Каждая из этих конструкций позволяет объединять поведение (код) и состояние (данные) в одном вызываемом объекте. Замыкания проще, но их сфера применимости ограничена. Ниже приведен пример замыкания из раздела "Вложенные функции и вложенные области видимости" главы 3.

```
def make_adder_as_closure(augend):
    def add(addend, _augend=augend): return addend+_augend
    return add
```

Связанные методы и вызываемые экземпляры богаче по своим возможностям и намного гибче замыканий. Ниже приведен пример реализации этой же функциональности с помощью связанного метода.

```
def make_adder_as_bound_method(augend):
    class Adder(object):
        def __init__(self, augend): self.augend = augend
        def add(self, addend): return addend+self.augend
    return Adder(augend).add
```

Ту же функциональность можно реализовать с помощью вызываемого экземпляра (экземпляра, класс которого предоставляет специальный метод `__call__`).

```
def make_adder_as_callable_instance(augend):
    class Adder(object):
        def __init__(self, augend): self.augend = augend
        def __call__(self, addend): return addend+self.augend
    return Adder(augend)
```

С точки зрения вызывающего кода все эти функции-фабрики взаимозаменяемы, поскольку возвращаемые ими вызываемые объекты *полиморфны* (т.е. используются одинаковым способом). В терминах реализации замыкание обеспечивает самый простой подход; связанный метод и вызываемый экземпляр используют более гибкие, общие и мощные механизмы, но в данном простом примере эти дополнительные возможности излишни.

Наследование

Если вы пытаетесь сослаться на атрибут *C.имя* объекта класса *C* и строка *'имя'* не является ключом в словаре *C*.`__dict__`, то процедура поиска переходит к просмотру объектов каждого из классов, встречающихся в атрибуте *C*.`__bases__`, в определенном порядке (который по историческим причинам носит название *порядок разрешения методов*, или MRO (Method Resolution Order), но применяется ко всем атрибутам, а не только к методам). В свою очередь, базовые классы класса *C* могут иметь собственные базовые классы. Процедура поиска поочередно просматривает прямых и косвенных предков в соответствии с MRO и останавливается, когда находит требуемое имя.

Порядок разрешения методов

Поиск имени атрибута в классе выполняется путем обхода классов-предков в восходящем направлении и слева направо. Однако в случае множественного наследования (при котором граф наследования представляет собой направленный ациклический граф, а не дерево) этот простой подход может приводить к тому, что некоторые классы-предки будут проходиться дважды. В подобных случаях порядок разрешения оставляет в последовательности просмотра только *крайнее справа* вхождение любого конкретного класса.

Каждый класс и встроенный тип имеет специальный атрибут класса, `__mro__`, доступный только для чтения, который представляет собой кортеж типов, используемых в процессе разрешении методов и расположенных в соответствующем порядке. К атрибутам `__mro__` можно обращаться только через классы, а не экземпляры, а поскольку атрибуты `__mro__` доступны только для чтения, то их нельзя повторно связывать или откреплять. Более подробное описание и техническое обоснование всех аспектов использования MRO в Python можно найти в статье Мишеля Симионато *The Python 2.3 Method Resolution Order* (`https://www.python.org/download/releases/2.3/mro/`) и историческом очерке Гвидо ван Россума на сайте "Python History" (`http://python-history.blogspot.com/2010/06/method-resolution-order.html`).

Перекрытие атрибутов

Как вы только что видели, поиск атрибутов проводится в соответствии с правилами MRO (в типичных случаях — снизу вверх вдоль дерева наследования) и прекращается, как только удается найти атрибут. Классы-потомки всегда просматриваются раньше предков, поэтому, если подкласс определяет атрибут с тем же именем, что и в суперклассе, поиск возвращает определение атрибута из подкласса и на этом прекращается. В подобных случаях говорят, что подкласс *перекрывает* (переопределяет) определение, содержащееся в суперклассе. Рассмотрим следующий пример.

```
class B(object):
    a = 23
    b = 45
    def f(self): print('method f in class B')
    def g(self): print('method g in class B')
class C(B):
    b = 67
    c = 89
    d = 123
    def g(self): print('method g in class C')
    def h(self): print('method h in class C')
```

В этом коде класс *C* перекрывает атрибуты *b* и *g* своего суперкласса *B*. Заметьте, что, в отличие от некоторых других языков программирования, в Python разрешается перекрывать не только атрибуты данных, но и вызываемые атрибуты (методы).

Делегирование выполнения методам суперкласса

В ситуациях, когда подкласс *C* перекрывает метод *f* своего суперкласса *B*, телу метода *C.f* часто требуется делегировать выполнение части своей работы версии метода, реализованной в суперклассе. Иногда это можно сделать с помощью объекта, являющегося в версии v2 несвязанным методом, а в версии v3 — объектом функции, что приводит к одинаковым результатам. Рассмотрим соответствующий пример.

```
class Base(object):
    def greet(self, name): print('Welcome', name)
class Sub(Base):
    def greet(self, name):
        print('Well Met and', end=' ')
        Base.greet(self, name)
x = Sub()
x.greet('Alex')
```

Для делегирования выполнения суперклассу в теле метода `Sub.greet` используется несвязанный метод (в версии v2; в версии v3 это объект функции, использующий тот же синтаксис и работающий точно так же, как несвязанный метод), вызов которого осуществляется посредством ссылки на атрибут `Base.greet` суперкласса с передачей всех аргументов, включая `self`. Делегирование выполнения варианту метода, реализованному в суперклассе, — это наиболее распространенный способ использования несвязанных методов в версии v2.

Делегирование часто используется в отношении специального метода `__init__`. В Python, в отличие от некоторых других языков программирования, методы `__init__` базовых классов не вызываются автоматически при создании экземпляров. Ответственность за надлежащую инициализацию суперклассов возлагается на подклассы, что в необходимых случаях делается посредством механизма делегирования.

```
class Base(object):
    def __init__(self):
        self.anattribute = 23
class Derived(Base):
    def __init__(self):
        Base.__init__(self)
        self.anotherattribute = 45
```

Если бы метод `__init__` класса `Derived` не вызывал явно метод `__init__` класса `Base`, то экземпляры класса `Derived` инициализировались бы лишь частично, и поэтому лишились бы атрибута `anattribute`. Эта проблема не возникает, если подкласс не имеет собственного метода `__init__`, поскольку в таком случае он наследует данный метод от своего суперкласса. Поэтому в использовании следующего кода нет никакой необходимости.

```
class Derived(Base):
    def __init__(self):
    Base.__init__(self)
```

Никогда не используйте методы, которые всего лишь вызывают методы суперклассов

Никогда не определяйте семантически пустой метод `__init__` (т.е. метод, который всего лишь делегирует выполнение методу суперкласса): вместо этого просто наследуйте метод `__init__` суперкласса. Этот совет

касается всех методов, как специальных, так и не специальных, но по каким-то причинам плохая привычка писать подобные семантически пустые методы чаще всего проявляется именно в отношении метода `__init__`.

Кооперативный вызов методов суперклассов

Вызов версии метода суперкласса с использованием синтаксиса несвязанного метода становится довольно проблематичным в случае множественного наследования с ромбовидными графами. Рассмотрим следующие определения.

```
class A(object):
    def met(self):
        print('A.met')
class B(A):
    def met(self):
        print('B.met')
        A.met(self)
class C(A):
    def met(self):
        print('C.met')
        A.met(self)
class D(B,C):
    def met(self):
        print('D.met')
        B.met(self)
        C.met(self)
```

В этом случае при выполнении вызова `D().met()` метод `A.met()` будет вызван дважды. Как добиться того, чтобы реализация вызываемого метода в каждом из предков выполнялась только один раз? Решение заключается в том, чтобы использовать встроенный тип `super`. В версии v2 для этого требуется выполнить вызов `super(`*`класс, объект`*`)`, который возвращает специальный суперобъект указанного объекта. Поиск атрибута в этом суперобъекте начинается *после* указанного класса в соответствии с порядком MRO объекта. Поэтому в версии v2 предыдущий код можно переписать в следующем виде.

```
class A(object):
    def met(self):
        print('A.met')
class B(A):
    def met(self):
        print('B.met')
        super(B,self).met()
class C(A):
    def met(self):
        print('C.met')
        super(C,self).met()
```

```
class D(B,C):
    def met(self):
        print('D.met')
        super(D,self).met()
```

В версии v3, несмотря на приемлемость в ней синтаксиса версии v2 (обеспечивающего нормальное выполнение предыдущего фрагмента кода), семантика суперобъекта усовершенствована таким образом, что функцию `super` можно вызывать без указания аргументов.

Теперь вызов `D().met()` сопровождается ровно одним вызовом метода `met()` каждого класса. Если вы выработаете привычку всегда обращаться к методам суперклассов посредством вызовов `super`, то ваши классы будут аккуратно вписываться в любые сложные структуры наследования. В случае же простых структур наследования никаких неприятных последствий от использования такого подхода не будет.

Использование более грубого подхода, основанного на вызове методов суперклассов посредством синтаксиса несвязанных методов, может оказаться более предпочтительным лишь в тех ситуациях, когда один и тот же метод имеет разные сигнатуры в разных классах. Такие ситуации чреваты многими неприятностями, но если вам действительно приходится иметь с ними дело, то синтаксис несвязанных методов может оказаться наименьшим из зол. Предоставление методов с одинаковыми именами, но разными и несовместимыми сигнатурами в разных подклассах и суперклассах не только препятствует надлежащей реализации множественного наследования, но и входит в противоречие с такими фундаментальными принципами ООП, как полиморфизм между экземплярами базовых и производных классов.

"Удаление" атрибутов классов

Наследование и перекрытие обеспечивают простой и эффективный способ добавления или переопределения атрибутов классов (например, методов) без каких-либо отрицательных последствий (т.е. без изменения базовых классов, определяющих атрибуты) путем добавления и перекрытия атрибутов в подклассах. Однако в механизме наследования отсутствуют средства, позволяющие удалять (скрывать) атрибуты базовых классов столь же простым и безболезненным способом. В тех случаях, когда атрибут не определяется (не перекрывается) в подклассе, Python находит его определение в базовом классе. Если вам необходимо выполнить подобное "удаление", то это можно сделать несколькими способами:

- перекрыть метод и сгенерировать в теле метода исключение;
- хранить атрибуты в другом месте, отличном от словаря `__dict__` подкласса, и определить метод `__getattr__` для селективного делегирования;
- перекрыть метод `__getattribute__` для достижения того же результата.

Последний прием описан в разделе "Метод `__getattribute__`".

Встроенный тип object

Встроенный тип object — предок всех встроенных типов и классов нового стиля. Тип object определяет ряд специальных методов (документированы в разделе "Специальные методы"), которые реализуют семантику объектов, используемую по умолчанию.

__new__, __init__
: Экземпляр object можно создать непосредственно, вызвав тип object() без аргументов. В этом вызове неявно используются вызовы object.__new__ и object.__init__, которые создают и возвращают экземпляр объекта, не имеющий атрибутов (и даже словаря __dict__, предназначенного для хранения атрибутов). Такие объекты могут быть полезны в качестве "сигнальных", которые гарантированно не будут равны никакому другому объекту при сравнении.

__delattr__, __getattribute__, __setattr__
: Эти методы объекта object по умолчанию используются любым объектом для обработки ссылок на атрибуты (см. раздел "Основные сведения о ссылках на атрибуты").

__hash__, __repr__, __str__
: Функциям hash и repr и типу str в качестве аргумента может быть передан любой объект.

Подкласс object может перекрыть любой из этих методов и/или добавить другие.

Методы уровня класса

Python поддерживает два встроенных типа неперекрывающих дескрипторов, которые предоставляют классу две разновидности "методов уровня класса": *статические методы* и *методы класса*.

Статические методы

Статический метод — это метод, который можно вызывать для любого класса или любого экземпляра данного класса и который не обладает специальным поведением и не подчиняется ограничениям в отношении первого параметра, налагаемым на обычные методы, связанные и несвязанные. Статический метод может иметь любую сигнатуру; он может не иметь параметров, а первый параметр, если он предоставляется, не играет никакой особой роли. Статический метод можно рассматривать как обычную функцию, которую можно вызывать обычным способом, несмотря на тот факт, что она связана с атрибутом класса.

Несмотря на то что определение статических методов не является строго необходимым (вместо них всегда можно определить обычные функции вне класса), некоторые программисты охотно используют их в качестве элегантной синтаксической

альтернативы в тех случаях, когда назначение функции тесно связано со спецификой класса.

Чтобы создать статический метод, следует вызвать встроенный тип `staticmethod` и связать возвращенный им результат с атрибутом класса. Как и в случае любого связывания атрибутов, обычно это делается в теле класса, но также может быть сделано в другом месте. Единственным аргументом вызова `staticmethod` является функция, которая будет вызываться всякий раз, когда Python будет вызывать статический метод. Один из способов определения и вызова статического метода продемонстрирован в следующем примере.

```
class AClass(object):
    def astatic(): print('a static method')
    astatic = staticmethod(astatic)
an_instance = AClass()
AClass.astatic()                    # вывод: a static method
an_instance.astatic()               # вывод: a static method
```

В этом примере для функции, передаваемой методу `staticmethod`, и атрибута, который связывается с результатом, возвращаемым методом `staticmethod`, используется одно и то же имя. Вы не обязаны так поступать, но в целом это неплохой подход, и мы рекомендуем всегда придерживаться его. Кроме того, Python предлагает специальный упрощенный синтаксис для поддержки такого стиля (раздел "Декораторы").

Методы класса

Метод класса — это метод, который можно вызвать для класса или любого экземпляра данного класса. Python связывает первый параметр этого метода с классом или классом экземпляра, для которого вызывается метод, однако сам метод не связывается с экземпляром, как в случае обычных связанных методов. В качестве имени первого параметра метода класса общепринято использовать имя `cls`.

Несмотря на то что определение методов класса никогда не является строго необходимым (вместо них всегда можно определить вне класса обычные функции, принимающие объект класса в качестве первого параметра), они предлагают элегантную синтаксическую альтернативу таким функциям (в частности, по той причине, что их удобно перекрывать в подклассах, если в этом возникает необходимость).

Для создания метода класса следует вызвать встроенный тип `classmethod` и связать возвращенный им результат с атрибутом класса. Как и в случае любого связывания атрибутов класса, обычно это делается в теле класса, но это также может быть сделано в другом месте. Единственным аргументом метода `classmethod` является функция, которая будет вызываться всякий раз, когда Python вызывает метод класса. Один из способов определения и вызова метода класса продемонстрирован в следующем примере.

```
class ABase(object):
    def aclassmet(cls): print('a class method for', cls.__name__)
```

```
    aclassmet = classmethod(aclassmet)
class ADeriv(ABase): pass
b_instance = ABase()
d_instance = ADeriv()
ABase.aclassmet()            # вывод: a class method for ABase
b_instance.aclassmet()       # вывод: a class method for ABase
ADeriv.aclassmet()           # вывод: a class method for ADeriv
d_instance.aclassmet()       # вывод: a class method for ADeriv
```

В этом примере для функции, передаваемой методу `classmethod`, и атрибута, который связывается с результатом, возвращаемым методом `classmethod`, используется одно и то же имя. Вы не обязаны так поступать, но в целом это неплохой подход, и мы рекомендуем всегда придерживаться его. Кроме того, Python предлагает специальный упрощенный синтаксис для поддержки такого стиля, о чем рассказано в разделе "Декораторы".

Свойства

Python предоставляет встроенный тип перекрывающего дескриптора, который можно использовать для создания свойств экземпляров классов.

Свойство — это вычисляемый атрибут экземпляра, обладающий специальной функциональностью. К нему можно обращаться, а также изменять или удалять его, используя обычный синтаксис (например, `print(`*x.свойство*`)`, *x.свойство*`=23`, `del` *x.свойство*). Однако, вместо того чтобы следовать обычной семантике ссылок на атрибуты, попытки такого доступа к свойствам приводят к вызову для экземпляра *x* методов, которые вы указываете в качестве аргументов встроенного типа `property`. Один из способов определения свойства, доступного только для чтения, продемонстрирован в следующем примере.

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def get_area(self):
        return self.width * self.height
    area = property(get_area, doc='площадь прямоугольника)
```

Каждый экземпляр `r` класса `Rectangle` имеет синтетический, доступный только для чтения атрибут `r.area`, вычисляемый на лету в методе `r.get_area()` путем перемножения сторон прямоугольника. Строка документирования `Rectangle.area.__doc__` представляет строку `'площадь прямоугольника'`. Атрибут `r.area` доступен только для чтения (попытка связать или открепить его приведет к возбуждению исключения), поскольку при вызове типа `property` мы определили для него метод `get`, но не определили методы `set` и `del`.

Свойства работают аналогично специальным методам `__getattr__`, `__setattr__` и `__delattr__` (раздел "Универсальные специальные методы"), но они проще

и быстрее. Чтобы создать свойство, следует вызвать встроенный тип property и связать возвращаемый результат с атрибутом класса. Как и во всех случаях связывания атрибутов классов, обычно это делается в теле класса, но также может быть сделано в другом месте. В теле класса *C* для этого можно использовать следующий синтаксис:

```
attrib = property(fget=None, fset=None, fdel=None, doc=None)
```

Если *x* — экземпляр класса *C* и вы обращаетесь к атрибуту *x.attrib*, то Python вызывает для *x* метод, переданный конструктору свойства в качестве аргумента *fget*, без передачи ему каких-либо аргументов. При выполнении операции присваивания *x.attrib = значение* Python вызывает метод, указанный в качестве аргумента *fset*, передавая ему значение в качестве единственного аргумента. При выполнении операции удаления `del x.attrib` Python вызывает метод, переданный конструктору свойства в качестве аргумента *fdel*, без передачи ему каких-либо аргументов. Переданный конструктору свойства аргумент *doc* Python использует в качестве строки документирования данного атрибута. Все параметры, передаваемые конструктору property, являются необязательными. Операции, соответствующие отсутствующим аргументам, являются запрещенными (при попытке выполнения такой операции Python генерирует исключение). Так, в примере с классом Rectangle свойство area доступно только для чтения, поскольку мы указали аргумент только для параметра *fget*, но не для параметров *fset* и *fdel*.

Более элегантный синтаксис создания свойств в классе предлагает использование декоратора property (раздел "Декораторы").

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    @property
    def area(self):
        '''площадь прямоугольника'''
        return self.width * self.height
```

Чтобы использовать этот синтаксис, вы должны предоставить метод getter, имя которого совпадает с именем желаемого свойства; строка документирования метода становится строкой документирования свойства. Если вы хотите добавить также методы, устанавливающие и/или удаляющие свойство, используйте декораторы с именами (в данном примере) area.setter и area.deleter и присвойте декорированным таким способом методам имена, совпадающие с именем свойства.

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    @property
    def area(self):
```

```
        '''площадь прямоугольника'''
        return self.width * self.height
    @area.setter
    def area(self, value):
        scale = math.sqrt(value/self.area)
        self.width *= scale
        self.height *= scale
```

Чем полезны свойства

Свойства важны тем, что их существование позволяет сделать совершенно безопасным (что настоятельно рекомендуется) предоставление общего доступа к атрибутам данных как части общедоступного интерфейса класса. Если в будущих версиях вашего класса или других классов, полиморфных по отношению к данному, вам когда-либо понадобится, чтобы при попытке обратиться к атрибуту, а также изменить его связывание или открепить его, выполнялся некоторый код, то вы можете быть уверены в том, что сможете превратить простой атрибут в свойство и добиться желаемого эффекта, не вмешиваясь в любой другой код, использующий ваш класс (известный как "клиентский код"). Это позволит вам избежать таких неуклюжих идиом, как *методы-получатели* и *методы-установщики*, используемые в тех объектно-ориентированных языках программирования, в которых свойства или эквивалентные им механизмы отсутствуют. Например, в Python клиентский код может использовать естественные идиомы наподобие

```
some_instance.widget_count += 1
```

вместо уродливых конструкций вложенных методов-получателей и методов-установщиков:

```
some_instance.set_widget_count(some_instance.get_widget_count() + 1)
```

Если у вас когда-либо возникнет настоятельная необходимость в написании методов с именами наподобие `get_this` или `set_that`, оберните их для ясности свойствами.

Свойства и наследование

Свойства наследуются точно так же, как любой другой атрибут. Однако существует одна особенность, которая может стать ловушкой для тех, кто о ней не знает: при попытке доступа к свойству вызываются методы, определенные в том классе, в котором было определено само свойство, без использования внутренних возможностей дополнительного перекрытия методов в подклассах. Рассмотрим соответствующий пример.

```
class B(object):
    def f(self): return 23
    g = property(f)
class C(B):
```

```
    def f(self): return 42
c = C()
print(c.g)                              # вывод: 23, а не 42
```

Доступ к свойству `c.g` инициирует вызов `B.f`, а не `C.f`, как можно было бы ожидать. Причина довольно проста: конструктор свойства получает (прямо или косвенно, посредством синтаксиса декоратора) *объект функции* `f` (это происходит во время выполнения инструкции `class` для класса `B`, так что данным объектом является функция `B.f`). Поэтому тот факт, что подкласс `C` впоследствии переопределяет имя `f`, не имеет значения, поскольку свойство не выполняет поиск этого имени, а просто использует объект функции, переданный конструктору во время создания свойства. Если вам потребуется обойти описанную проблему, вы всегда сможете это сделать, самостоятельно введя дополнительный уровень косвенности в процедуру поиска атрибутов.

```
class B(object):
    def f(self): return 23
    def _f_getter(self): return self.f()
    g = property(_f_getter)
class C(B):
    def f(self): return 42
c = C()
print(c.g)                          # вывод: 42, как и ожидалось
```

Здесь объектом функции, хранящимся в свойстве, является `B._f_getter`, который, в свою очередь, выполняет поиск имени `f` (поскольку он вызывает метод `self.f()`), поэтому перекрытие `f` обеспечивает желаемый эффект. Здесь как нельзя более уместна знаменитая фраза Дэвида Уиллера: "В компьютерной науке введение дополнительного уровня косвенности позволяет решить любую проблему"[1].

Атрибут `__slots__`

Обычно каждый экземпляр объекта *x* любого класса *C* имеет словарь `x.__dict__`, который Python использует для того, чтобы предоставить вам возможность связывать произвольные атрибуты объекта *x*. Вы можете сэкономить немного памяти (за счет того, что объекту *x* будет позволено иметь только предопределенный набор атрибутов) и организовать хранение атрибутов в так называемых *слотах*, определив в классе *C* атрибут `__slots__`, представляющий собой последовательность (обычно кортеж) строк (обычно идентификаторов). Непосредственный экземпляр *x* класса *C*, имеющего атрибут `__slots__`, не имеет словаря `x.__dict__`, и любая попытка связывания любого атрибута *x*, имя которого отсутствует в атрибуте `C.__slots__`, приведет к возникновению исключения.

Использование слотов позволяет уменьшить расход памяти в случае небольших экземпляров объектов, которые могут справляться со своими задачами без

[1] Ради справедливости приведем обычно опускаемое окончание этой фразы:"...кроме, разумеется, проблемы чрезмерно большого количества косвенностей".

привлечения произвольных именованных атрибутов. Атрибут __slots__ стоит добавлять только в классы, количество экземпляров которых может быть настолько большим, что экономия нескольких десятков байт на экземпляр имеет существенное значение, т.е. в классы, для которых количество одновременно существующих экземпляров может исчисляться даже не тысячами, а миллионами. В отличие от других атрибутов класса, атрибут __slots__ работает в соответствии с приведенным описанием только в том случае, если он был связан посредством присваивания ему значения в теле класса. Никакое последующее изменение или удаление атрибута __slots__, равно как и его наследование от базового класса, не окажет никакого влияния. Ниже показано, как добавить атрибут __slots__ в определенный ранее класс Rectangle для того, чтобы уменьшить размер его экземпляров (пусть даже за счет некоторой потери гибкости).

```
class OptimizedRectangle(Rectangle):
    __slots__ = 'width', 'height'
```

Нам не нужно определять слот для свойства area. Атрибут __slots__ не налагает ограничения на свойства, он ограничивает только обычные атрибуты экземпляров, которые находились бы в словаре __dict__ экземпляра, если бы атрибут __slots__ не был определен.

Метод __getattribute__

Все обращения к атрибутам экземпляра проходят через специальный метод __getattribute__. Этот метод происходит от объекта object, в котором он реализует детали семантики обращения к атрибутам, документированные в разделе "Основные сведения о ссылках на атрибуты". Однако вы можете перекрыть метод __getattribute__ для специальных целей, таких как сокрытие унаследованных атрибутов класса в экземплярах подклассов. Ниже приведен пример реализации списка, не имеющего метода append.

```
class listNoAppend(list):
    def __getattribute__(self, name):
        if name == 'append': raise AttributeError(name)
        return list.__getattribute__(self, name)
```

Экземпляр *x* класса listNoAppend почти неотличим от встроенного объекта списка, за исключением того, что его производительность существенно ниже и при любом обращении к методу *x*.append генерируется исключение.

Методы экземпляра

Для всех атрибутов, включая вызываемые атрибуты (методы), возможно связывание на уровне отдельных экземпляров. Связывание метода, как и любого другого атрибута (за исключением тех, которые связаны с перекрывающими дескрипторами), на уровне экземпляра скрывает связывание на уровне класса: поиск связывания

атрибута в классе не выполняется, если его удается найти на уровне экземпляра. При обращении к вызываемому атрибуту, связанному на уровне экземпляра, преобразования, описанные в разделе "Связанные и несвязанные методы", не выполняются: ссылка на атрибут возвращает в точности тот же вызываемый объект, который ранее был непосредственно связан с атрибутом экземпляра.

Однако для определенных на уровне экземпляра связываний специальных методов, которые Python неявно вызывает при выполнении различных операций (раздел "Специальные методы"), это работает не так, как можно было бы ожидать. В подобных неявных вызовах всегда используются связывания специальных методов, определенные на уровне класса.

```
def fake_get_item(idx): return idx
class MyClass(object): pass
n = MyClass()
n.__getitem__ = fake_get_item
print(n[23])        # приводит к следующему:
# Traceback (most recent call last):
# File "<stdin>", line 1, in ?
# TypeError: unindexable object
```

Наследование от встроенных типов

Класс может наследоваться от встроенного типа. Однако групповое расширение встроенных типов, прямое или косвенное, возможно только в случае, если для этих типов специально предусмотрен подобный уровень совместимости. Python не поддерживает безусловную, не подчиняющуюся никаким ограничениям, возможность группового наследования от произвольных встроенных типов. Обычно класс нового стиля расширяет максимум один основной встроенный тип в дополнение к типу `object`, который является суперклассом для всех встроенных типов и классов и не налагает никаких ограничений на групповое наследование. Например, попытка использовать следующую инструкцию приведет к возбуждению исключения `TypeError` и выводу сообщения о несовместимости базовых типов:

```
class noway(dict, list): pass
```

Появление таких сообщений указывает на прямую или косвенную попытку наследования от нескольких встроенных типов, совместное использование которых на столь глубоком уровне не было изначально предусмотрено.

Специальные методы

Класс может определять или наследовать специальные методы (методы, имена которых начинаются и заканчиваются двойными символами подчеркивания; также известны как "магические" методы). Каждый специальный метод ассоциирован с определенной операцией. Python неявно вызывает специальный метод всякий раз, когда

над экземпляром объекта выполняется соответствующая операция. В большинстве случаев возвращаемым значением специального метода является результат выполнения операции, а попытки выполнения операции, для которой не определен специальный метод, приводят к возникновению исключения.

В этом разделе мы будем отдельно обращать ваше внимание на те случаи, когда эти общие правила оказываются неприменимыми. В последующем изложении *x* — это экземпляр класса *C*, над которым выполняется операция, а *y* — другой операнд. Параметр `self` каждого метода также ссылается на экземпляр объекта *x*. В следующих разделах всякий раз, когда приводятся вызовы вида *x*.__*метод*__(...), вы должны понимать, что, строго говоря, речь идет о вызовах вида *x*.__class__.__*метод*__(*x*, ...).

Универсальные специальные методы

Некоторые специальные методы ассоциированы с универсальными операциями. Класс, который определяет или наследует эти методы, позволяет своим экземплярам управлять данными операциями. Эти операции можно разделить на следующие категории.

Инициализация и финализация
: Класс может управлять инициализацией своих экземпляров (весьма распространенное требование) посредством специальных методов `__new__` и `__init__`, а также их финализацией (редко встречающееся требование) посредством метода `__del__`.

Представление в виде строки
: Класс может управлять способом представления своих экземпляров в виде строк, используемым Python, посредством специальных методов `__repr__`, `__str__`, `__format__`, `__bytes__` (только версия v3) и `__unicode__` (только версия v2).

Сравнение, хеширование и использование в булевом контексте
: Класс может управлять способом сравнения своих экземпляров с другими объектами (методы `__lt__`, `__le__`, `__gt__`, `__ge__`, `__eq__`, `__ne__`), тем, как словари используют экземпляры в качестве ключей, а множества — в качестве элементов (метод `__hash__`), а также тем, каким значениям — истинным или ложным — должны соответствовать экземпляры в булевых контекстах (метод `__nonzero__` в версии v2, метод `__bool__` в версии v3).

Получение, связывание и удаление атрибутов
: Класс может управлять доступом к атрибутам своих экземпляров (получением, связыванием, удалением) посредством специальных методов `__getattribute__`, `__getattr__`, `__setattr__` и `__delattr__`.

Вызываемые экземпляры

Экземпляр является вызываемым подобно любому объекту функции, если его класс имеет специальный метод `__call__`.

Универсальные специальные методы документированы в табл. 4.1.

Таблица 4.1. Универсальные специальные методы

Метод	Описание
`__bytes__`	`__bytes__(self)` В версии v3 при вызове функции `bytes(x)` вызывается метод `x.__bytes__()`, если он предоставлен. Если класс предоставляет оба специальных метода, `__bytes__` и `__str__`, оба они должны возвращать эквивалентные строки (байтовые или строковые соответственно)
`__call__`	`__call__(self[,` *аргументы...*`])` Когда вы выполняете вызов `x([`*аргументы...*`])`, Python транслирует эту операцию в вызов `x.__call__([`*аргументы...*`])`. Аргументы операции вызова соответствуют параметрам метода `__call__` за исключением первого. Первый параметр, для которого общепринято использовать имя `self`, ссылается на *x*, и Python автоматически предоставляет его неявным образом, как и при вызове любого другого связанного метода
`__dir__`	`__dir__(self)` Когда вы выполняете вызов `dir(x)`, Python транслирует эту операцию в вызов `x.__dir__()`, который должен вернуть отсортированный список атрибутов *x*. Если класс объекта *x* не имеет метода `__dir__`, то вызов `dir(x)` самостоятельно выполняет интроспекцию для возврата списка атрибутов *x*, стремясь предоставить наиболее существенную, а не полную информацию
`__del__`	`__del__(self)` В промежутке времени между распознанием объекта *x* сборщиком мусора как неиспользуемого и его удалением Python вызывает метод `x.__del__()` для того, чтобы позволить объекту *x* выполнить финальные операции (например, проверить, освобождены ли ценные внешние ресурсы). Если метод `__del__` отсутствует, то после попадания объекта *x* в сборщик мусора Python не выполняет никаких завершающих операций (типичный случай, поскольку лишь немногие классы нуждаются в определении метода `__del__`). Python игнорирует значение, возвращаемое методом `__del__`, и не выполняет никаких неявных вызовов методов `__del__` суперклассов класса *C*. Любые необходимые финальные операции, включая делегирование вызовов, если оно необходимо, должны выполняться с помощью явного вызова метода

Метод	Описание
	`C.__del__`. Например, если у класса `C` имеется базовый класс `B`, нуждающийся в финализации, то код метода `C.__del__` должен содержать вызов `super(C, self).__del__()` (или, в версии v3, вызов `super().__del__()`). Заметьте, что метод `__del__` не равнозначен инструкции `del`, рассмотренной в разделе "Инструкции `del`" главы 3. В общем случае метод `__del__` не является наилучшим выходом в тех ситуациях, когда выполнение финальных операций должно быть гарантированным и своевременным. Для этого следует использовать инструкцию `try/finally`, которая обсуждается в разделе "Инструкция `try/finally`" главы 5 (или, что еще лучше, инструкцию `with`, которая обсуждалась в разделе "Инструкция `with`" главы 3). Экземпляры классов, определяющих метод `__del__`, не могут участвовать в процессе сборки мусора, в котором объекты связаны циклическими ссылками, что обсуждается в разделе "Сборка мусора" главы 13. Поэтому необходимо особенно тщательно избегать образования циклических ссылок, включающих подобные объекты, и определять метод `__del__` лишь при отсутствии других разумных альтернатив
`__delattr__`	`__delattr__(self, имя)` При каждом запросе на откреплние атрибута `x.y` (в типичных случаях посредством инструкции `del x.y`) Python вызывает метод `x.__delattr__('y')`. Все рассуждения, касающиеся метода `__setattr__`, который обсуждается далее, также относятся к методу `__delattr__`. Python игнорирует значение, возвращаемое методом `__delattr__`. В случае отсутствия метода `__delattr__` Python транслирует инструкцию `del x.y` в инструкцию `del x.__dict__['y']`
`__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, `__ne__`	`__eq__(self, объект)`, `__ge__(self, объект)`, `__gt__(self, объект)`, `__le__(self, объект)`, `__lt__(self, объект)`, `__ne__(self, объект)` При выполнении операций сравнения `x==y`, `x>=y`, `x>y`, `x<=y`, `x<y` и `x!=y` вызывается соответствующий специальный метод из приведенного выше списка методов, которые должны возвращать значение `False` или `True`. Каждый из методов может вернуть значение `NotImplemented`, чтобы запросить обработку сравнения альтернативными способами (например, Python может попытаться выполнить сравнение `y>x` вместо `x<y`). Наилучшая практика заключается в том, чтобы определить только один метод проверки неравенства (обычно `__lt__`) плюс метод `__eq__` и применить к классу декоратор `functools.total_ordering` (описан в табл. 7.4), чтобы избавиться от необходимости повторения шаблонного кода для других операторов сравнения и избежать риска возникновения логических противоречий при реализации операторов сравнения

Метод	Описание
`__getattr__`	`__getattr__(self, имя)` Если атрибут `x.y` не удается получить с помощью обычной поэтапной процедуры поиска (что обычно приводит к возникновению исключения `AttributeError`), Python вызывает метод `x.__getattr__('y')`. Если атрибут удается найти обычными способами (например, среди ключей словаря `x.__dict__` или в классе `x.__class__`), Python не вызывает метод `__getattr__`. Если вы хотите, чтобы Python вызывал метод `__getattr__` при каждом обращении к атрибуту, сохраните атрибуты в другом месте (например, в другом словаре, на который можно ссылаться через атрибут с частным именем) или перекройте метод `__getattribute__`. Если метод `__getattr__` не может найти `y`, он должен сгенерировать исключение `AttributeError`
`__getattribute__`	`__getattribute__(self, имя)` При каждой попытке обращения к атрибуту `x.y` Python вызывает метод `x.__getattribute__('y')`, который должен получить и вернуть значение атрибута или сгенерировать исключение `AttributeError`. Всю обычную семантику доступа к атрибутам (`x.__dict__`, `C.__slots__`, атрибуты класса `C`, `x.__getattr__`) обеспечивает метод `object.__getattribute__`. Если класс `C` перекрывает метод `__getattribute__`, то он должен реализовать всю семантику обращения к атрибутам, которую он намерен предложить. Чаще всего наиболее удобным способом реализации семантики доступа к атрибутам является делегирование вызова (например, путем вызова `object.__getattribute__(self, ...)` как части операции, выполняемой вашей перекрытой версией метода `__getattribute__`).  **Перекрытие метода `__getattribute__` замедляет доступ к атрибутам** Если класс перекрывает метод `__getattribute__`, то доступ к атрибутам экземпляров класса замедляется, поскольку перекрывающий код выполняется при каждом обращении к атрибуту
`__hash__`	`__hash__(self)` При вызове функции `hash(x)` вызывается метод `x.__hash__()` (и так во всех других контекстах, требующих знания хеш-кода, а именно в случае использования `x` в качестве ключа словаря, как в выражении `D[x]`, где `D` — словарь, или в случае использования `x` в качестве элемента множества). Метод `__hash__` должен возвращать целочисленное значение, такое, что `x==y` подразумевает `hash(x)==hash(y)`, причем для любого заданного объекта это значение всегда должно быть одним и тем же.

Метод	Описание
	В случае отсутствия метода `__hash__` вместо него при вызове `hash(`*x*`)` вызывается функция `id(`*x*`)`, при условии, что отсутствует также метод `__eq__`. Поведение функции `hash(`*x*`)` остается таким же во всех контекстах, требующих знания хеш-кода. Любой объект *x*, для которого вызов `hash(`*x*`)` возвращает результат, а не генерирует исключение, называется *хешируемым объектом*. Если метод `__hash__` отсутствует, но метод `__eq__` имеется, вызов функции `hash(`*x*`)` генерирует исключение (и так во всех других контекстах, требующих знания хеш-кода). В этом случае *x* не является хешируемым объектом и потому не может быть ключом в словаре или элементом множества. Как правило, метод `__hash__` определяют лишь для неизменяемых объектов, для которых также определен метод `__eq__`. Заметьте, что если существует любой *y*, такой, что *x*==*y*, даже если *y* относится к другому типу, и оба объекта, *x* и *y* являются хешируемыми, то вы должны убедиться в том, что `hash(`*x*`)==hash(`*y*`)`
`__init__`	`__init__(self[,` *аргументы*`...])` При создании экземпляра *x* класса *C* с помощью вызова *C*`([`*аргументы*`...])` Python вызывает метод *x*`.__init__([`*аргументы*`...])`, предоставляя *x* возможность инициализировать себя. Если метод `__init__` отсутствует (т.е. наследуется от объекта `object`), то класс *C* следует вызывать без аргументов: *C*`()`. В этом случае вновь созданный объект *x* не будет иметь специфических для экземпляра атрибутов. Python не выполняет неявные вызовы методов `__init__` суперклассов класса *C*. Метод *C*`.__init__` должен явно выполнить необходимую инициализацию, в соответствующих случаях делегируя вызовы. Например, если класс *C* имеет базовый класс *B*, который должен быть инициализирован без аргументов, то код в *C*`.__init__` должен явно вызвать функцию `super(`*C*`, self).__init__()` (или, в версии v3, выполнить вызов `super().__init__()`). В то же время для метода `__init__` наследование работает точно так же, как и для любого другого метода или атрибута: если класс *C* не перекрывает метод `__init__` самостоятельно, то он наследует его от первого из суперклассов, указанных в списке `__mro__`, чтобы перекрыть `__init__`, как и любой другой атрибут. Метод `__init__` должен возвращать значение `None`, в противном случае вызов класса приведет к возникновению исключения `TypeError`
`__new__`	`__new__(cls[,` *аргументы*`...])` Когда выполняется вызов `C([`*аргументы*`...])`, Python получает новый экземпляр *x*, который вы создаете, вызывая метод `C.__new__(`*C*`, [`*аргументы*`...])`. Каждый класс имеет метод класса `__new__`

Метод	Описание
	(чаще всего получаемый путем наследования от `object`), который может вернуть любое значение *x*. Другими словами, метод `__new__` не обязан возвращать новый экземпляр *C*, хотя обычно он используется именно для этого. Тогда и только тогда, когда возвращаемое методом `__new__` значение *x* действительно является экземпляром *C* или любого подкласса *C* (нового или существующего), Python неявно вызывает метод `__init__` для экземпляра *x* (с тем же списком аргументов `[`*аргументы*`...]`, который первоначально передавался методу `__new__`).
	 **Инициализируйте неизменяемые объекты в `__new__`, все остальные — в `__init__`** Поскольку в большинстве случаев инициализацию экземпляров можно выполнить как в методе `__init__`, так и в методе `__new__`, у вас может возникнуть вопрос относительно того, какой из этих вариантов лучше. Ответ очень прост: для инициализации экземпляров всегда используйте только метод `__init__`, кроме тех случаев, когда для выполнения инициализации с помощью метода `__new__` имеются веские причины. (В случае неизменяемых типов изменение их экземпляров в методе `__init__` с целью их инициализации невозможно, и это есть тот особый случай, когда вся инициализация должна быть выполнена в методе `__new__`.) Эта подсказка немного упрощает жизнь, поскольку `__init__` — метод экземпляра, в то время как `__new__` — специализированный метод класса
`__nonzero__`	`__nonzero__(self)`
	При оценке значения *x* в качестве истинного или ложного (см. раздел "Булевы значения" на стр. 103) — например, при вызове `bool(`*x*`)` — в версии v2 вызывается метод *x*`.__nonzero__()`, который должен возвращать значение `True` или `False`. Когда метод `__nonzero__` отсутствует, Python вызывает метод `__len__` и рассматривает *x* в качестве ложного значения, если вызов `x.__len__()` возвращает `0` (поэтому, чтобы проверить, является ли контейнер непустым, не стоит использовать выражение `if len(container)>0:`; для этого достаточно использовать выражение `if container:`). Если отсутствуют оба метода, `__nonzero__` и `__len__`, то Python всегда считает *x* истинным значением.
	В версии v3 для этих целей используется специальный метод с более подходящим названием: `__bool__`
`__repr__`	`__repr__(self)`
	При вызове функции `repr(`*x*`)` (который неявно выполняется в интерактивном интерпретаторе, если *x* является результатом

Метод	Описание
	выражения) вызывается метод `x.__repr__()`, который получает и возвращает полное строковое представление `x`. Если метод `__repr__` отсутствует, Python использует строковое представление, заданное по умолчанию. Метод `__repr__` должен возвращать строку с однозначно интерпретируемой информацией относительно `x`. В идеальном случае, если это возможно, строка должна представлять собой такое выражение, что `eval(repr(x))==x` (но не переусердствуйте в достижении этой цели)
`__setattr__`	`__setattr__(self, имя, значение)` При каждом запросе на связывание атрибута `x.y` (как правило, это происходит во время выполнения инструкций присваивания `x.y=значение`, но также, например, при вызове функции `setattr(x, 'y', значение)`) Python вызывает метод `x.__setattr__('y', значение)`. Python всегда выполняет вызов `__setattr__` для *любого* связывания атрибута `x` — основное отличие от метода `__getattr__` (в этом отношении метод `__setattr__` ближе к методу `__getattribute__`). Чтобы избежать рекурсии при связывании атрибутов `x` посредством вызова `x.__setattr__`, он должен изменить непосредственно словарь `x.__dict__` (например, с помощью вызова `x.__dict__[имя]=значение`); еще лучше, если метод `__setattr__` делегирует установку атрибута суперклассу посредством вызова `super(C, x).__setattr__('y', значение)` или, в версии v3, посредством вызова `super().__setattr__('y', значение)`). Python игнорирует значение, возвращаемое методом `__setattr__`. Если метод `__setattr__` отсутствует (т.е. наследуется от `object`) и `C.y` не является перекрывающим дескриптором, то обычно Python транслирует инструкцию `x.y=z` в инструкцию `x.__dict__['y']=z`
`__str__`	`__str__(self)` Встроенный тип `str(x)` и функция `print(x)` вызывают метод `x.__str__()` для получения неформального, компактного строкового представления `x`. Если метод `__str__` отсутствует, Python вызывает вместо него метод `x.__repr__`. Метод `__str__` должен возвращать удобную для чтения человеком строку, даже если она будет предоставлять лишь некую приблизительную информацию
`__unicode__`	`__unicode__(self)` В версии v2 при вызове функции `unicode(x)` вызывается метод `x.__unicode__()`, если он есть, даже когда имеется также метод `x.__str__()`. Если класс предоставляет оба специальных метода, `__unicode__` и `__str__`, то оба они должны возвращать эквивалентные строки (строку Unicode и простую текстовую строку соответственно)

Метод	Описание
__format__	__format__(self, format_string='') При вызове функции format(*x*) вызывается метод *x*.__format__(), а при вызове функции format(*x*, *строка_формата*) — метод *x*.__format__(*строка_формата*). Ответственность за интерпретацию строки формата возлагается на класс (каждый класс может определить собственный небольшой "язык" спецификаций формата по аналогии со спецификациями, реализованными во встроенных типах, которые обсуждаются в разделе "Форматирование строк" главы 8). Если метод __format__ унаследован от объекта object, то он делегирует вызов методу __str__ и не принимает непустую строку формата

Специальные методы контейнеров

Экземпляр может быть *контейнером* (последовательностью, отображением или множеством — это взаимно исключающие понятия[2]). Чтобы быть максимально полезными, контейнеры должны предоставлять специальные методы __getitem__, __setitem__, __delitem__, __len__, __contains__ и __iter__ плюс неспециальные методы, обсуждаемые в следующих разделах. Во многих случаях подходящие реализации неспециальных методов могут быть получены путем расширения соответствующих абстрактных базовых классов из модуля collections, таких как Sequence, MutableSequence и т.д., которые рассматриваются в разделе "Абстрактные базовые классы".

Последовательности

В каждом из специальных методов, обеспечивающих доступ к элементам, последовательность, включающая *L* элементов, должна допускать любое целочисленное значение индекса *ключ*, такое, что -L<=*ключ*<L[3]. Для совместимости со встроенными последовательностями отрицательные индексы *ключ* (0>*ключ*>=-*L*) должны быть эквивалентны индексам *ключ*+*L*. Если значение *ключ* относится к недопустимому типу, то попытка обращения к элементу по индексу должна генерировать исключение TypeError. Если значение *ключ* относится к допустимому типу, но выходит за границы диапазона, то попытка обращения к элементу по индексу должна генерировать исключение IndexError. Для классов последовательностей, которые не определяют метод __iter__, инструкция for, а также встроенные функции, принимающие итерируемые аргументы, работают в соответствии с этими требованиями. Каждый специальный метод последовательности для доступа к элементам также должен, если это осуществимо, принимать в качестве аргумента индекса экземпляр встроен-

[2] Расширения от независимых производителей могут определять также другие типы контейнеров, не являющиеся ни последовательностями, ни отображениями, ни множествами.

[3] Нижняя граница включается в диапазон, верхняя — исключается, что является обычным для Python.

ного типа slice, атрибуты start, step и stop которого должны быть целыми числами или иметь значение None. Синтаксис выделения срезов основывается на этом требовании (раздел "Срезы контейнеров").

Последовательность также должна допускать конкатенацию (с другой последовательностью того же типа) с помощью оператора + и повторения с помощью оператора * (умножение на целое число). Поэтому последовательность должна иметь специальные методы __add__, __mul__, __radd__ и __rmul__, рассмотренные в разделе "Специальные методы для числовых объектов". Кроме того, изменяемые последовательности должны иметь эквивалентные методы __iadd__ и __imul__, выполняющие указанные операции на месте. Последовательность должна иметь понятные правила сравнения с другой последовательностью такого же типа, реализуя *лексикографическое* сравнение, как это сделано для списков и кортежей. (Для удовлетворения этих требований одного наследования от абстрактных базовых классов Sequence или MutableSequence, увы, недостаточно; такое наследование предоставляет лишь метод __iadd__.)

Каждая последовательность должна иметь неспециальные методы, рассмотренные в разделе "Методы списков" главы 3: count и index в любом случае и, если последовательность изменяемая, также методы append, insert, extend, pop, remove, reverse и sort с теми же сигнатурами и семантикой, что и соответствующие методы списков. (За исключением метода sort, для удовлетворения этих требований достаточно наследования от абстрактных базовых классов Sequence или MutableSequence.)

Изменяемая последовательность должна быть хешируемой в том и только в том случае, если таковыми являются все ее элементы. Тип последовательности может налагать определенные ограничения на ее элементы (например, допускать только строковые элементы), но это необязательно.

Отображения

В случае получения недопустимого значения аргумента *ключ* допустимого типа специальные методы, обеспечивающие доступ к элементам отображений, должны генерировать исключение KeyError, а не IndexError. Любое отображение должно определять неспециальные методы, рассмотренные в разделе "Методы словарей" главы 3: copy, get, items, keys, values и, в версии v2, iteritems, iterkeys и itervalues. В версии v2 специальный метод __iter__ должен быть эквивалентным методу iterkeys (в версии v3 он должен быть эквивалентным методу keys, который в версии v3 имеет ту же семантику, какую метод iterkeys имеет в версии v2). Изменяемое отображение должно определять также методы clear, pop, popitem, setdefault и update. (Наследование от абстрактных базовых классов позволяет удовлетворить эти требования, за исключением метода copy.)

Неизменяемое отображение должно быть хешируемым, если хешируемыми являются все его элементы. Тип отображения может налагать на свои ключи те или иные ограничения (например, допускать только хешируемые ключи или, при более специфическом ограничении, только строковые), но это необязательно. Любое

отображение должно иметь понятные правила сравнения с другим отображением того же типа, по крайней мере, для отношений равенства и неравенства и, необязательно, для отношений порядка.

Множества

Множества — особый вид контейнеров. Это контейнеры, которые не являются ни последовательностями, ни отображениями. Они не могут индексироваться, но имеют размер (количество элементов) и являются итерируемыми. Кроме того, множества поддерживают ряд операций (`&`, `|`, `^`, `-`, а также операции проверки принадлежности и сравнения) и эквивалентных им неспециальных методов (`intersection`, `union` и др.). Если вы реализуете контейнер наподобие множества, то он должен быть полиморфным по отношению к встроенным множествам Python, рассмотренным в разделе "Множества" главы 3. (Наследования от абстрактных базовых классов достаточно для удовлетворения этих базовых требований.)

Неизменяемый тип, подобный множеству, должен быть хешируемым, если таковыми являются все его элементы. Тип, подобный множеству, может налагать на свои элементы те или иные ограничения (например, допускать только хешируемые элементы или, при более специфическом ограничении, только целочисленные элементы), но это необязательно.

Срезы контейнеров

При получении, связывании или откреплении срезов контейнера *x*, таких как `x[i:j]` или `x[i:j:k]` (на практике это используется лишь в отношении последовательностей), Python вызывает специальный метод объекта *x*, соответствующий выполняемой операции доступа, передавая ему в качестве аргумента *ключ* объект встроенного типа, называемый *объектом среза*. Объект среза имеет атрибуты `start`, `stop` и `step`. Атрибут, значение которого опущено в выражении среза, принимает значение `None`. Например, при выполнении операции `del x[:3]` автоматически вызывается метод `x.__delitem__(y)`, где *y* — это объект среза, в котором атрибут `y.stop` равен `3`, а каждый из атрибутов `y.start` и `y.step` равен `None`. Ответственность за надлежащую интерпретацию аргументов, передаваемых специальным методам объекта контейнера *x* вместе с объектом среза, возлагается на объект *x*. Для этого удобно использовать метод `indices` объекта среза: будучи вызванным с передачей ему размера (длины) контейнера в качестве единственного аргумента, он возвращает кортеж из трех неотрицательных индексов, пригодных для использования в качестве атрибутов `start`, `stop` и `step`, обеспечивающих индексирование каждого элемента среза в цикле. Например, применительно к специальному методу `__getitem__` класса последовательности широко используется следующая идиома, обеспечивающая полную поддержку срезов.

```
def __getitem__(self, index):
  # Специальное рекурсивное выделение срезов
  if isinstance(index, slice):
```

```
    return self.__class__(self[x]
                          for x in range(*index.indices(len(self))))
# Проверить индекс, также обрабатывая
# отрицательные значения индексов
if not isinstance(index, numbers.Integral): raise TypeError
if index < 0: index += len(self)
if not (0 <= index < len(self)): raise IndexError
# Теперь индекс является корректным целым числом
# из диапазона от 0 до range(len(self))
...остальная часть кода __getitem__, обрабатывающая доступ
к одиночному элементу...
```

В этой идиоме используется синтаксис выражения-генератора и предполагается, что метод `__init__` вашего класса может быть вызван с итерируемым аргументом для создания подходящего нового экземпляра класса.

Методы контейнеров

Функциональность контейнеров обеспечивают специальные методы `__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__` и `__contains__` (табл. 4.2).

Таблица 4.2. Специальные методы контейнеров

Метод	Описание

`__contains__`	Булева операция *y* `in` *x* вызывает метод *x*`.__contains__(`*y*`)`. Если объект *x* подобен последовательности или множеству, то метод `__contains__` должен возвращать значение `True`, когда *y* равно значению элемента, содержащегося в *x*. Если *x* — отображение, то метод `__contains__` должен возвращать значение `True`, когда *y* равно значению ключа в *x*. В противном случае метод `__contains__` должен возвращать значение `False`. Если метод `__contains__` отсутствует, то Python выполняет операцию *y* `in` *x*, затрачивая на это время, пропорциональное `len(`*x*`)`, следующим образом:
	`for z in x:`
	`  if y==z: return True`
	`return False`

`__delitem__`	`__delitem__(self,` *ключ*`)` В ответ на запрос открепления элемента или среза контейнера *x* (обычно посредством выполнения операции `del` *x*`[`*ключ*`]`) Python вызывает метод *x*`.__delitem__(`*ключ*`)`. Контейнер *x* должен иметь метод `__delitem__` только в том случае, если *x* — изменяемый объект, позволяющий удалять его элементы (и, возможно, срезы)
`__getitem__`	`__getitem__(self,` *ключ*`)` При попытке доступа к элементам с использованием выражения *x*`[`*ключ*`]` (т.е. при попытке получения элементов с помощью индекса или среза контейнера *x*) Python вызывает метод *x*`.__getitem__(`*ключ*`)`. Все контейнеры (за исключением подобных множествам) должны иметь метод `__getitem__`

Метод	Описание
__iter__	__iter__(self) В ответ на запрос выполнения цикла по всем элементам *x* (обычно в форме for *элемент* in *x*) Python вызывает метод *x*.__iter__() для получения итератора объекта *x*. Встроенная функция iter(*x*) также вызывает метод *x*.__iter__(). Если метод __iter__ отсутствует, то функция iter(*x*) синтезирует и возвращает объект итератора, который обертывает *x* и вырабатывает значения *x*[0], *x*[1] и т.д. до тех пор, пока при попытке получения одного из этих индексированных элементов не возникнет исключение IndexError, указывающее на исчерпание контейнера. Однако наилучший выход заключается в том, чтобы предусматривать метод __iter__ для каждого создаваемого класса контейнера
__len__	__len__(self) При вызове метода len(*x*) автоматически вызывается метод *x*.__len__() (то же самое происходит при вызове любой другой встроенной функции, которой необходимо знать количество элементов в контейнере *x*). Функция __len__ должна возвращать целое число, представляющее количество элементов в контейнере *x*. Кроме того, Python вызывает метод *x*.__len__() для оценки *x* в булевом контексте, если отсутствует метод __nonzero__ (__bool__ в версии v3); в этом случае контейнер рассматривается как ложное значение тогда и только тогда, когда он пуст (т.е. когда его размер равен 0). Все контейнеры должны иметь метод __len__, за исключением тех случаев, когда определение количества содержащихся в контейнере элементов является слишком дорогой операцией
__setitem__	__setitem__(self, *ключ*, *значение*) В ответ на запрос связывания элемента или среза *x* (обычно в форме операции присваивания *x*[*ключ*]=*значение*) Python вызывает метод *x*.__setitem__(*ключ*, *значение*). Контейнер *x* должен иметь метод __setitem__ только в том случае, если *x* — изменяемый объект, допускающий добавление и/или связывание элементов и, возможно, срезов

Абстрактные базовые классы

Абстрактные базовые классы — важный шаблон проектирования в ООП: эти классы нельзя использовать непосредственно для создания экземпляров, и существуют они лишь для того, чтобы их могли расширять реальные классы (более привычная разновидность классов, экземпляры которых можно создавать непосредственно).

В соответствии с одним из рекомендованных в ООП подходов никогда не следует расширять реальный класс: если два конкретных класса имеют так много общего, что у вас возникает искушение сделать один из них наследником другого, то вместо этого лучше создать абстрактный базовый класс, который вберет в себя все, что у них есть общего, и предоставить каждому реальному классу возможность расширить абстрактный базовый класс. Такой подход позволяет избежать многих тонких подвохов и ловушек, таящихся в механизме наследования.

Python предлагает достаточно мощную поддержку абстрактных классов, чтобы они стали объектами первого класса в объектной модели Python.

Модуль abc

Модуль abc стандартной библиотеки предоставляет метакласс ABCMeta и в версии v3 — класс ABC (создание подкласса ABC делает ABCMeta метаклассом и не имеет никаких других эффектов).

Используя abc.ABCMeta в качестве метакласса по отношению к любому классу *C*, вы превращаете класс *C* в абстрактный базовый класс и предоставляете метод класса *C*.register, который принимает один аргумент: этим аргументом может быть любой существующий класс (или встроенный тип) *X*.

Вызов метода *C*.register(*X*) превращает *X* в виртуальный подкласс *C*, т.е. вызов issubclass(*X*, *C*) возвращает значение True, но *C* не появляется в списке *X*.__mro__, а *X* не наследует методы и другие атрибуты *C*.

Разумеется, также допускается, чтобы новый класс *Y* наследовался от класса *C* обычным способом. В этом случае *C* появляется в списке *Y*.__mro__, а *Y* наследует все методы *C*, как при обычном создании подклассов.

Кроме того, абстрактный базовый класс *C* может, хотя это и не является обязательным, перекрывать метод класса __subclasshook__, который функция issubclass(*X*, *C*) вызывает с одним аргументом *X*, где *X* — любой класс или тип. Если метод *C*.__subclasshook__(*X*) возвращает значение True, то это же значение возвращает и функция issubclass(*X*, *C*); если метод *C*.__subclasshook__(*X*) возвращает False, то это же значение возвращает и функция issubclass(*X*, *C*); если метод *C*.__subclasshook__(*X*) возвращает значение NotImplemented, то функция issubclass(*X*, *C*) продолжает выполнение, как обычно.

Кроме того, модуль abc предоставляет декоратор abstractmethod (а также декоратор abstractproperty, но последний признан устаревшим в версии v3, в которой декораторы abstractmethod и abstractproperty теперь оказывают один и тот же эффект). Абстрактные методы и свойства могут иметь реализации (доступные в подклассах посредством встроенной функции super), однако цель создания абстрактных методов и свойств заключается в том, чтобы вы могли инстанциализировать любой невиртуальный подкласс *X* абстрактного базового класса *C* только в том случае, если *X* перекрывает каждое абстрактное свойство и каждый метод класса *C*.

Абстрактные базовые классы модуля collections

Модуль collections предоставляет множество абстрактных базовых классов. Начиная с Python 3.4 абстрактные базовые классы хранятся в модуле collections.abc (но в целях обратной совместимости к ним по-прежнему возможен непосредственный доступ через модуль collections: последний вариант доступа перестанет работать в одном из ближайших будущих выпусков версии v3).

Некоторые базовые классы лишь характеризуют любой класс, определяющий или наследующий конкретный абстрактный метод (табл. 4.3).

Таблица 4.3. Абстрактные базовые классы

Класс	Описание
`Callable`	Любой класс, имеющий метод `__call__`
`Container`	Любой класс, имеющий метод `__contains__`
`Hashable`	Любой класс, имеющий метод `__hash__`
`Iterable`	Любой класс, имеющий метод `__iter__`
`Sized`	Любой класс, имеющий метод `__len__`

Другие абстрактные базовые классы модуля `collections` расширяют один или несколько предыдущих, добавляют дополнительные абстрактные методы и предоставляют *примесные методы*, реализуемые в терминах абстрактных методов. (Расширяя любой абстрактный базовый класс в реальном классе, необходимо перекрыть абстрактные методы. Вы также можете перекрыть некоторые или все примесные методы, если это улучшит производительность, но делать это необязательно — их можно просто наследовать, если результирующая производительность вас устраивает.)

Ниже приведен набор абстрактных базовых классов (ABC), непосредственно расширяющих предыдущие.

ABC	Расширяемый класс	Абстрактные методы	Примесные методы
`Iterator`	`Iterable`	`__next__` (в версии v2 — `next`)	`__iter__`
`Mapping`	`Container`, `Iterable`, `Sized`	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `__eq__`, `__ne__`, `get`, `items`, `keys`, `values`
`MappingView`	`Sized`		`__len__`
`Sequence`	`Container`, `Iterable`, `Sized`	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `count`, `index`
`Set`	`Container`, `Iterable`, `Sized`	`__contains__`, `__iter__`, `__len__`	`__and__`, `__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, `__ne__`, `__or__`, `__sub__`, `__xor__`, `isdisjoint`

И наконец, ниже приведен набор абстрактных базовых классов (ABC), дополнительно расширяющих предыдущие.

ABC	Расширяемый класс	Абстрактные методы	Примесные методы
`ItemsView`	`MappingView`, `Set`		`__contains__`, `__iter__`

ABC	Расширяемый класс	Абстрактные методы	Примесные методы
KeysView	MappingView, Set		__contains__, __iter__
MutableMapping	Mapping	__delitem__, __getitem__, __iter__, __len__, __setitem__	Методы отображений плюс clear, pop, popitem, setdefault, update
MutableSequence	Sequence	__delitem__, __getitem__, __len__, __setitem__, insert	Методы последовательностей плюс __iadd__, append, extend, pop, remove, reverse
MutableSet	Set	__contains__, __iter__, __len__, add, discard	Методы множеств плюс __iand__, __ior__, __isub__, __ixor__, clear, pop, remove
ValuesView	MappingView		__contains__, __iter__

Дополнительную информацию и примеры применения можно найти в онлайн-документации Python по адресу `https://docs.python.org/2/library/collections.html#collections-abstract-base-classes`.

Модуль numbers

Модуль `numbers` содержит иерархию абстрактных базовых классов, представляющих различные типы чисел. Список этих классов приводится ниже.

Number Корень иерархии: числа *любого* типа (от него не требуется поддержка каких-либо конкретных операций)

Complex Расширяет класс `Number`; должен поддерживать (посредством подходящих специальных методов) преобразование в типы `complex` и `bool`, операции +, -, *, /, ==, !=, `abs()` и непосредственно метод `conjugate()` и свойства `real` и `imag`

Real Расширяет класс `Complex`; дополнительно должен поддерживать (посредством соответствующих специальных методов) преобразование в тип `float`, функции `math.trunc()`, `round()`, `math.floor()`, `math.ceil()`, `divmod()`, операции //, %, <, <=, >, >=

Rational Расширяет класс `Real`; дополнительно должен поддерживать свойства `numerator` и `denominator`

Integral Расширяет класс `Rational`; дополнительно должен поддерживать (посредством подходящих специальных методов) преобразование в тип `int`, операцию ** и побитовые операции <<, >>, &, ^, |, ~

Замечания относительно реализации собственных числовых типов можно найти в онлайн-документации Python по адресу `https://docs.python.org/2/library/numbers.html`.

Специальные методы для числовых объектов

Экземпляр может поддерживать числовые операции посредством множества специальных методов. Некоторые классы, не являющиеся числовыми, также поддерживают специальные методы, приведенные в табл. 4.4, для перегрузки таких операторов, как + и *. В частности, последовательности должны иметь специальные методы `__add__`, `__mul__`, `__radd__` и `__rmul__` (см. раздел "Последовательности").

Таблица 4.4. Специальные методы для числовых объектов

Метод	Описание
`__abs__`, `__invert__`, `__neg__`, `__pos__`	`__abs__(self)`, `__invert__(self)`, `__neg__(self)`, `__pos__(self)` Эти методы вызываются при выполнении унарных операций `abs(x)`, `~x`, `-x` и `+x` соответственно
`__add__`, `__mod__`, `__mul__`, `__sub__`	`__add__(self,`*объект*`)`, `__mod__(self,`*объект*`)`, `__mul__(self,`*объект*`)`, `__sub__(self,`*объект*`)` Обычно эти методы вызываются при выполнении арифметических операций *x*+*y*, *x*%*y*, *x***y* и *x*-*y* соответственно
`__div__`, `__floordiv__`, `__truediv__`	`__div__(self,`*объект*`)`, `__floordiv__(self,`*объект*`)`, `__truediv__(self,`*объект*`)` Обычно эти методы вызываются операторами *x*/*y* и *x*//*y* для выполнения арифметических операций деления. В версии v2 оператор / вызывает метод `__truediv__`, если он имеется, вместо метода `__div__` в ситуациях, когда требуется деление без усечения до целого (см. раздел "Арифметические операции" в главе 3). В версии v3 метод `__div__` отсутствует и есть только методы `__truediv__` и `__floordiv__`
`__and__`, `__lshift__`, `__or__`, `__rshift__`, `__xor__`	`__and__(self,`*объект*`)`, `__lshift__(self,`*объект*`)`, `__or__(self,`*объект*`)`, `__rshift__(self,`*объект*`)`, `__xor__(self,`*объект*`)` Обычно эти методы вызываются при выполнении побитовых операций *x*&*y*, *x*<<*y*, *x*|*y*, *x*>>*y* и *x*^*y* соответственно
`__complex__`, `__float__`, `__int__`, `__long__`	`__complex__(self)`, `__float__(self)`, `__int__(self)`, `__long__(self)` Обычно эти методы вызываются встроенными типами `complex(x)`, `float(x)`, `int(x)` и (только в версии v2) `long(x)` соответственно
`__divmod__`	`__divmod__(self,`*объект*`)` Встроенная функция `divmod(x,y)` вызывает метод `x.__divmod__(y)`, который должен возвращать пару (*частное*, *остаток*), эквивалентную паре (*x*//*y*, *x*%*y*)

Метод	Описание
__hex__, __oct__	`__hex__(self), __oct__(self)` Только в версии v2: встроенная функция `hex(`*x*`)` вызывает метод *x*`.__hex__()`, а встроенная функция `oct(`*x*`)` вызывает метод *x*`.__oct__()`. Каждый из этих специальных методов должен возвращать строку, представляющую значение *x* в шестнадцатеричной и восьмеричной системах счисления соответственно. В версии v3 эти специальные методы отсутствуют: встроенные функции `hex` и `oct` воздействуют непосредственно на результат вызова специального метода `__index__` для их операнда
__iadd__, __idiv__, __ifloordiv__, __imod__, __imul__, __isub__, __itruediv__	`__iadd__(self,`*объект*`), __idiv__(self,`*объект*`), __ifloordiv__(self, `*объект*`), __imod__(self,`*объект*`), __imul__(self,`*объект*`), __isub__(self,`*объект*`), __itruediv__(self,`*объект*`)` Эти методы вызываются составными операциями присваивания *x*`+=`*y*, *x*`/=`*y*, *x*`//=`*y*, *x*`%=`*y*, *x*`*=`*y*, *x*`-=`*y* и *x*`/=`*y* соответственно. Каждый метод должен изменять *x* на месте и возвращать `self`. Определяйте эти методы, если *x* — изменяемый объект (т.е. когда *x* можно изменить на месте)
__iand__, __ilshift__, __ior__, __irshift__, __ixor__	`__iand__(self,`*объект*`), __ilshift__(self,`*объект*`), __ior__(self,`*объект*`), __irshift__(self,`*объект*`), __ixor__(self,`*объект*`)` Эти методы вызываются составными операциями присваивания *x*`&=`*y*, *x*`<<=`*y*, *x*`|=`*y*, *x*`>>=`*y* и *x*`^=`*y* соответственно. Каждый метод должен изменять *x* на месте и возвращать `self`
__index__	`__index__(self)` Подобен методу `__int__`, но предполагается, что его должны предоставлять только типы, являющиеся альтернативными реализациями целых чисел (другими словами, типы, для которых возможно точное отображение всех экземпляров на целые числа). Например, из всех встроенных типов только тип `int` (и в версии v2 — тип `long`) предоставляет метод `__index__`; однако типы `float` и `str` не предоставляют этот метод, хотя и предоставляют метод `__int__`. Метод `__index__` внутренне используется при индексировании и выделении срезов последовательностей для получения целочисленных индексов
__ipow__	`__ipow__(self,`*объект*`)` Операция комбинированного присваивания *x*`**=`*y* вызывает метод *x*`.__ipow__(`*y*`)`. Метод `__ipow__` должен изменять *x* на месте и возвращать объект `self`

Метод	Описание
__pow__	`__pow__(self,`*объект*`[,`*модуль*`])` Операции *x****y* и `pow(`*x*`,`*y*`)` вызывают метод *x*`.__pow__(`*y*`)`, тогда как операция `pow(`*x*`,`*y*`,`*z*`)` вызывает метод *x*`.__pow__(`*y*`,`*z*`)`. Метод *x*`.__pow__(`*y*`,`*z*`)` должен возвращать значение, равное результату вычисления выражения *x*`.__pow__(`*y*`)%`*z*
__radd__, __rdiv__, __rmod__, __rmul__, __rsub__	`__radd__(self,`*объект*`)`, `__rdiv__(self,`*объект*`)`, `__rmod__(self,`*объект*`)`, `__rmul__(self,`*объект*`)`, `__rsub__(self,`*объект*`)` Эти методы вызываются для *x* операциями *y*+*x*, *y*/*x*, *y*%*x*, *y***x* и *y*-*x* соответственно, если *y* не имеет необходимого метода `__add__`, `__div__` и т.д. или если этот метод возвращает значение `NotImplemented`
__rand__, __rlshift__, __ror__, __rrshift__, __rxor__	`__rand__(self,`*объект*`)`, `__rlshift__(self,`*объект*`)`, `__ror__(self,`*объект*`)`, `__rrshift__(self,`*объект*`)`, `__rxor__(self,`*объект*`)` Эти методы вызываются для *x* операциями *y*&*x*, *y*<<*x*, *y*|*x*, *y*>>*x* и *x*^*y* соответственно, если *y* не имеет необходимого метода `__and__`, `__lshift__` и т.д. или если этот метод возвращает значение `NotImplemented`
__rdivmod__	`__rdivmod__(self,`*объект*`)` Встроенная функция `divmod(`*y*`,`*x*`)` вызывает метод *x*`.__rdivmod__(`*y*`)`, если *y* не имеет метода `__divmod__` или если этот метод возвращает значение `NotImplemented`. Метод `__rdivmod__` должен возвращать пару `(`*остаток*`,`*частное*`)`
__rpow__	`__rpow__(self,`*объект*`)` Операции *y****x* и `pow(`*y*`,`*x*`)` вызывают метод *x*`.__rpow__(`*y*`)`, если *y* не имеет метода `__pow__` или если этот метод возвращает значение `NotImplemented`. В данном случае формы с тремя аргументами не существует

Декораторы

В Python часто приходится использовать так называемые *функции высшего порядка* — вызываемые объекты, которые принимают функцию в качестве аргумента и возвращают функцию в качестве результата. Например, в теле класса можно использовать такие типы дескрипторов, как `staticmethod` и `classmethod` (см. раздел "Методы уровня класса").

```
def f(cls, ...):
  ...код f опущен...
f = classmethod(f)
```

Однако размещение вызова `classmethod` после определения функции затрудняет чтение кода: просматривая определение *f*, читатель еще не знает, что впоследствии *f* станет методом класса, а не методом экземпляра. Читаемость кода улучшится, если упоминание о методе класса будет *предшествовать* инструкции `def`, а не *следовать за ней*. В подобных ситуациях используйте синтаксис *декораторов*.

```
@classmethod
def f(cls, ...):
  ...код f опущен...
```

Декоратор должен размещаться непосредственно перед инструкцией `def`. Наличие декоратора означает, что сразу вслед за инструкцией `def` должна быть выполнена инструкция *f*=`classmethod(`*f*`)` (где *f* — имя, которое определяет инструкция `def`). В более общем случае декоратор `@`*выражение* вычисляет *выражение* (которое должно быть именем, возможно, уточненным, или вызовом) и связывает результат с временным внутренним именем (скажем, `__aux`). За любым декоратором должна сразу же следовать инструкция `def` (или `class`), и это означает, что сразу же после выполнения инструкции `def` или `class` (для любого имени *f*, определяемого инструкцией `def` или `class`) автоматически выполняется инструкция *f*=`__aux(`*f*`)`. Объект, связанный с внутренним именем `__aux`, называется *декоратором*, который *декорирует* функцию или класс *f*.

Декораторы очень удобно использовать в качестве вспомогательных функций при работе с некоторыми функциями высшего порядка. Декораторы можно применять к любым инструкциям `def` или `class`, а не только к тем, которые встречаются в теле класса. Вы также можете создавать собственные декораторы, которые представляют собой функции высшего порядка, принимающие объект функции или класса в качестве аргумента и возвращающие объект функции или класса в качестве результата. Ниже приведен пример простого декоратора, который не изменяет декорируемую функцию, а просто выводит строку документирования данной функции во время ее определения.

```
def showdoc(f):
    if f.__doc__:
        print('{}: {}'.format(f.__name__, f.__doc__))
    else:
        print('{}: No docstring!'.format(f.__name__))
    return f

@showdoc
def f1(): """a docstring"""
# вывод: f1: a docstring

@showdoc
def f2(): pass
# вывод: f2: No docstring!
```

Модуль `functools` стандартной библиотеки предлагает удобный декоратор `wraps`, улучшающий поведение декораторов, создаваемых с использованием обычной идиомы "обертывания" объектов.

```
import functools
def announce(f):
    @functools.wraps(f)
    def wrap(*a, **k):
        print('Calling {}'.format(f.__name__))
        return f(*a, **k)
    return wrap
```

Декорирование функции *f* с помощью декоратора `@announce` приводит к тому, что перед каждым вызовом *f* будет выводиться строка, извещающая о вызове данной функции. Благодаря декоратору `functools.wraps(f)` функция-обертка располагает информацией об имени обернутой ею функции и ее строке документирования, что, например, может пригодиться для вывода справки по декорированной функции с помощью встроенной функции `help`.

Метаклассы

Любой объект, даже объект класса, имеет тип. В Python типы и классы также являются объектами первого класса. Тип объекта класса также называется *метаклассом* данного класса[4]. Поведение любого объекта главным образом определяется его типом. То же самое можно сказать и в отношении классов: поведение класса главным образом определяется его метаклассом. Понятие метакласса относится к категории продвинутых тем, и при первом чтении вы можете пропустить оставшуюся часть этого раздела. Однако изучение метаклассов поможет вам глубже понять особенности Python, а в некоторых случаях даже определить собственные, нестандартные метаклассы[5].

Определение метакласса в версии Python v2

В версии v2, прежде чем выполнить инструкцию `class`, базовые классы собираются в кортеж *t* (пустой, если базовые классы отсутствуют), а затем выполняются инструкции тела класса, причем определенные в нем имена сохраняются во временном словаре *d*. После этого Python определяет метакласс *M* для нового объекта класса *C*, создаваемого с помощью инструкции `class`.

Если `'__metaclass__'` является ключом в словаре *d*, то *M* — это соответствующее ему значение `d['__metaclass__']`. Таким образом, вы можете явно управлять метаклассом класса *C* посредством связывания атрибута `__metaclass__` в теле класса *C* (для

[4] Строго говоря, рассматривать метакласс как тип класса *C* можно только применительно к экземплярам класса *C*, а не к самому классу, но это довольно тонкое отличие, и на практике, как правило, на это никто не обращает внимания.

[5] **Новое в версии 3.6**: несмотря на то что в версии 3.6 метаклассы работают точно так же, как и в версии 3.5, в версии 3.6 предусмотрены более простые способы изменения логики создания классов (см. документ **PEP 487**), которые часто могут служить неплохой альтернативой использованию пользовательских метаклассов.

улучшения читаемости эту инструкцию лучше всего помещать в тело класса в качестве первой инструкции сразу же после строки документирования). В противном случае, если кортеж `t` не является пустым (т.е., если у `C` имеется один или несколько базовых классов), то `M` — это метакласс, являющийся *граничным* среди всех метаклассов базовых классов `C` (т.е. метакласс базового класса `C`, являющийся подклассом всех других метаклассов базовых классов `C`; если же ни один метакласс базового класса `C` не является подклассом всех остальных, то это ошибка, и Python генерирует исключение)[6].

Именно поэтому наследование от `object` в версии v2 указывает на то, что `C` является классом нового стиля (а не старого — понятие, которое существует лишь в связи с поддержкой обратной совместимости и которое в данной книге не рассматривается). Поскольку `type(object)` — это `type`, то класс `C`, который наследуется от объекта `object` (или другого встроенного типа), получает тот же метакласс, что и `object` (т.е. метакласс класса `C`, `type(C)`, — это также `type`). Таким образом, в версии v2 "являться классом нового стиля" — это то же самое, что "иметь `type` в качестве своего метакласса".

Если класс `C` не имеет базовых классов, но в текущем модуле есть глобальная переменная `__metaclass__`, то в версии v2 метаклассом `M` является значение данной переменной. Это позволяет по умолчанию делать содержащиеся в модуле классы, не имеющие базовых классов, классами нового стиля, а не старого, для чего достаточно поместить в начале модуля (до инструкций `class`) следующую инструкцию:

```
__metaclass__ = type
```

Если ни одно из этих условий не выполняется (т.е., если метакласс не указан явно, не унаследован и не определен с помощью глобальной переменной модуля), то в версии v2 метаклассом по умолчанию является `types.ClassType` (который делает класс классом старого стиля).

Определение метакласса в версии Python v3

В версии v3 инструкция `class` принимает необязательные именованные аргументы (указываемые после базовых классов, если таковые имеются). Наиболее важный именованный аргумент — `metaclass`, который, если он имеется, идентифицирует метакласс нового класса. Другие именованные аргументы допускаются лишь при наличии метакласса, отличного от `type`, и в этом случае они передаются методу `__prepare__` метакласса (способ использования этих именованных параметров определяется исключительно методом `__prepare__`). Если именованный аргумент `metaclass` отсутствует, то в версии v3 метакласс определяется наследованием точно так же, как и в версии v2, или же, в случае классов, для которых базовые классы не указаны явно, принимает заданное по умолчанию значение `type`.

[6] В более точной (хотя и витиеватой) формулировке: если метаклассы базовых классов класса `C` не образуют иерархическую структуру, включающую свою нижнюю границу, — т.е. если метакласс первого базового класса `C` не является подклассом всех остальных метаклассов, — то Python генерирует исключение, диагностирующее конфликт метатипов.

В версии v3 метакласс имеет необязательный метод `__prepare__`, который Python вызывает, как только определяет метакласс.

```
class MC:
    def __prepare__(classname, *classbases, **kwargs):
        return {}
  ...остальная часть кода MC опущена...

class X(onebase, another, metaclass=MC, foo='bar'):
  ...код тела X опущен...
```

В данном случае этот вызов эквивалентен вызову `MC.__prepare__('X', onebase, another, foo='bar')`. Метод `__prepare__`, если он имеется, должен возвращать отображение (чаще всего — просто словарь) — Python использует его в качестве объекта *d*, содержащего описание пространства имен, в котором он выполняет тело класса. Если необходимо, чтобы в списке имен методов метакласса был сохранен тот порядок методов, в котором они создавались, то метод `__prepare__` может вернуть экземпляр класса `collections.OrderedDict` (раздел "Класс `OrderedDict`" в главы 7). Это позволяет метаклассам в версии v3 использовать класс для естественного представления таких конструкций, как схемы баз данных, в которых упорядочивание атрибутов имеет значение (что невозможно сделать в версии v2)[7]. Если метод `__prepare__` отсутствует, то в версии v3 в качестве объекта *d* используется словарь (как это делается в версии v2 во всех случаях).

Как метакласс создает класс

Определив метакласс *M*, Python вызывает *M* с тремя аргументами: имя класса (строка), кортеж базовых классов *t* и словарь (или, в версии v3, другое отображение *d*, формируемое методом `__prepare__`), в котором только что закончилось выполнение тела класса. Этот вызов возвращает объект класса *C*, который Python затем связывает с именем класса, завершая выполнение инструкции `class`. Заметьте, что в действительности описанный процесс представляет собой создание экземпляра типа *M*, поэтому вызов *M* выполняется как `M.__init__(C, строка_имени, t, d)`, где *C* — значение, возвращаемое вызовом `M.__new__(M, строка_имени, t, d)`, как это происходит при инстанциализации любого класса.

После того как Python создаст объект класса *C*, между классом *C* и его типом (`type(C)`, обычно *M*) существует такая же взаимосвязь, как между любым объектом и его типом. Например, если вы вызовете объект класса *C* (для создания экземпляра *C*), то вызов `M.__call__` выполнится с объектом класса *C* в качестве первого аргумента.

Обратите внимание на преимущество, предоставляемое в данном контексте подходом, описанным в разделе "Методы экземпляра", при котором поиск специальных

[7] **Новое в версии 3.6:** в привлечении в данном случае класса `OrderedDict` больше нет необходимости — в версии 3.6 гарантируется, что в используемом отображении сохраняется порядок ключей.

методов выполняется только в классе, а не в экземпляре — ключевое отличие объектной модели "нового стиля" от традиционной модели "старого стиля", используемой в версии v2. Вызов класса *C* с целью его инстанциализации должен выполнять вызов `M.__call__` независимо от того, имеет или не имеет *C* атрибут (метод) экземпляра `__call__` (т.е. независимо от того, являются или не являются экземпляры *C* вызываемыми). В старой объектной модели, в которой методы экземпляра перекрывают методы класса для неявно вызываемых специальных методов, это просто невозможно. В объектной модели Python (которая представляет "новый стиль" в версии v2 и является единственной в v3) удается избежать того, чтобы связь между классом и его метаклассом была особым случаем. Отказ от введения особых случаев является одним из ключевых преимуществ языка Python: в нем предусмотрено несколько простых общих правил, которые применяются повсеместно.

Определение и использование собственных метаклассов

В определении пользовательских метаклассов нет ничего сложного: для этого достаточно наследовать от типа `type` и перекрыть некоторые из его методов. При решении большинства этих задач можно обойтись и без метаклассов, используя такие специальные методы, как `__new__`, `__init__`, `__getattribute__` и т.п. Однако пользовательские метаклассы работают быстрее, поскольку специальная обработка выполняется лишь во время создания класса, что является сравнительно редкой операцией. Пользовательский метакласс позволяет определить в виде фреймворка целую категорию классов, которые чудесным образом приобретают поведение, заложенное вами в коде, совершенно независимо от того, какие специальные методы определены в самих классах.

Часто в качестве удобного альтернативного способа, позволяющего явно изменять какой-либо определенный класс, удобно использовать декораторы классов (см. раздел "Декораторы"). Однако декораторы не наследуются и поэтому должны явно применяться к каждому интересующему вас классу. С другой стороны, метаклассы наследуются. В действительности, когда вы определяете пользовательский метакласс *M*, то обычно определяете также пустой во всех остальных отношениях класс *C* с метаклассом *M*, чтобы другие классы, которым требуется метакласс *M*, могли просто наследовать от класса *C*.

Некоторые варианты поведения объектов классов можно настраивать только в метаклассах. В следующем примере продемонстрировано, как использовать метакласс для изменения формата строкового представления объектов класса.

```
class MyMeta(type):
    def __str__(cls):
        return 'Отличный класс {!r}'.format(cls.__name__)
class MyClass(metaclass=MyMeta):
    # В версии v2 следует удалить 'metaclass=' из вызова
    # и использовать в качестве тела класса следующую инструкцию:
    # __metaclass__ = MyMeta
x = MyClass()
print(type(x))     # вывод: Отличный класс 'MyClass'
```

Пример создания пользовательского метакласса

Предположим, что, программируя на Python, вы заметили, что вам не хватает типа данных наподобие `struct` языка C, т.е. объекта, который всего лишь хранил бы упорядоченную группу атрибутов с фиксированными именами (к этому близок тип `collections.namedtuple`, рассмотренный в разделе "Класс `namedtuple`" главы 7, однако именованные кортежи — неизменяемый тип данных, и иногда это может не устраивать вас). В Python не составляет труда определить обобщенный класс `Bunch`, удовлетворяющий не только этим требованиям, но и другим.

```
class SimpleBunch(object):
    def __init__(self, **fields):
        self.__dict__ = fields
p = SimpleBunch(x=2.3, y=4.5)
print(p)    # вывод: <__main__.SimpleBunch object at 0x00AE8B10>
```

Пользовательский метакласс позволяет использовать тот факт, что имена атрибутов фиксируются во время создания класса. Представленный в листинге 4.1 код определяет метакласс `MetaBunch` и класс `Bunch`, используя которые мы можем писать код наподобие следующего.

```
class Point(Bunch):
    """ Класс point не содержит ничего, кроме переменных x and y,
        представляющих координаты со значениями по умолчанию,
        равными 0.0, и переменной color (цвет) со значением
        по умолчанию, равным 'gray', за исключением методов
        __init__, __repr__ и других атрибутов, которые будут
        неявно добавлены Python и метаклассом.
    """

    x = 0.0
    y = 0.0
    color = 'gray'
# примеры использования класса Point
q = Point()
print(q)    # вывод: Point()
p = Point(x=1.2, y=3.4)
print(p)    # вывод: Point(y=3.399999999, x=1.2)
```

В этом коде вызовы функции `print` выводят строковые представления экземпляров `Point` в удобочитаемом виде. Экземпляры `Point` потребляют мало памяти, а в отношении производительности в основном сравнимы с экземплярами простого класса `SimpleBunch` из предыдущего примера (отсутствуют дополнительные накладные расходы, обусловленные неявными вызовами специальных методов). Приведенный ниже листинг 4.1 довольно внушителен и требует знания таких аспектов Python, как строки (глава 8) и модуль `warnings` (раздел "Фильтры" в главе 16), которые рассматриваются в последующих главах. В листинге 4.1 вместо общепринятого

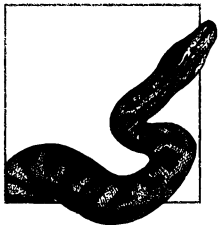
идентификатора `cls` используется идентификатор `mcl`, чтобы было понятно, что речь идет о метаклассе.

Код примера работает в обеих версиях, v2 и v3, за исключением того, что для версии v2 требуется небольшая корректировка синтаксиса, что отражено в строке документирования класса `Bunch`.

Листинг 4.1. Метакласс `MetaBunch`

```
import collections
import warnings

class MetaBunch(type):
    """
    Метакласс для улучшенного класса "Bunch": неявно определяет
    атрибуты __slots__, __init__ and __repr__ на основании
    переменных, связанных в области видимости класса.
    Инструкция class для экземпляра MetaBunch (т.е. для класса,
    метаклассом которого является MetaBunch) должна определять
    в области видимости класса только атрибуты данных
    (и, возможно, специальные методы, но НЕ __init__ и __repr__).
    MetaBunch удаляет атрибуты данных из области видимости класса,
    упрятывая их в находящийся в области видимости класса словарь
    __dflts__, и помещает в класс список __slots__, содержащий
    имена этих атрибутов, метод __init__, принимающий в качестве
    необязательных именованных аргументов каждое из них (используя
    значения из __dflts__ в качестве значений по умолчанию для
    аргументов, которые не указаны), и метод __repr__,
    отображающий атрибуты, которые отличается от своих значений по
    умолчанию (вывод __repr__ может передаваться методу __eval__
    для создания эквивалентного экземпляра в соответствии с
    обычными соглашениями в отношении этого, если каждый из
    атрибутов, значение которого отличается от значения по
    умолчанию, также соответствует данным соглашениям).

    В версии v3 порядок следования атрибутов данных остается
    тем же, что и в теле класса; в версии v2 это не гарантируется.
    """
    def __prepare__(name, *bases, **kwargs):
        # полезно в v3 и безопасно, хотя и бесполезно, в v2
        return collections.OrderedDict()

    def __new__(mcl, classname, bases, classdict):
        """ Все должно делаться в __new__, поскольку именно
            в type.__new__ учитываются значения из __slots__.
        """
        # определить __init__ и __repr__ как локальные функции,
        # которые мы будем использовать в новом классе
        def __init__(self, **kw):
            """ Упрощенный __init__: сначала задаем все атрибуты
```

```
            по умолчанию, а затем переопределяем те из них,
            которые переданы в аргументе kw.
        """
        for k in self.__dflts__:
            setattr(self, k, self.__dflts__[k])
    for k in kw:
        setattr(self, k, kw[k])
    def __repr__(self):
        """ Умный __repr__: для лаконичности отображает лишь
            атрибуты, отличающиеся от значений по умолчанию.
        """
        rep = ['{}={!r}'.format(k, getattr(self, k))
                for k in self.__dflts__
                if getattr(self, k) != self.__dflts__[k]
               ]
        return '{}({})'.format(classname, ', '.join(rep))
    # создать newdict, который будет использоваться в
    # качестве словаря класса для нового класса
    newdict = { '__slots__':[],
        '__dflts__':collections.OrderedDict(),
        '__init__':__init__, '__repr__':__repr__, }
    for k in classdict:
        if k.startswith('__') and k.endswith('__'):
            # методы "с двойными подчеркиваниями":
            # копировать в newdict или вывести
            # предупреждение о наличии конфликтов
            if k in newdict:
                warnings.warn(
                    "Can't set attr {!r} in bunch-class {!r}".
                    format(k, classname))
            else:
                newdict[k] = classdict[k]
        else:
            # переменные класса; сохранение имени в __slots__,
            # а имени и значения - в виде элемента в __dflts__
            newdict['__slots__'].append(k)
            newdict['__dflts__'][k] = classdict[k]
    # делегирование оставшейся работы type.__new__
    return super(MetaBunch, mcl).__new__(
                 mcl, classname, bases, newdict)

class Bunch(metaclass=MetaBunch):
    """ Наследование от Bunch можно использовать для получения
        нового метакласса.

        В версии v2 следует убрать задание аргумента
        (metaclass=MetaBunch) и использовать вместо этого
        инструкцию __metaclass__=MetaBunch в теле класса.
    """
    pass
```



5
Исключения

Python использует исключения для того, чтобы сообщить об ошибках и других проблемах, возникающих во время выполнения программы. *Исключение* — это объект, указывающий на возникновение ошибки или другой исключительной ситуации. Когда обнаруживается ошибка, Python возбуждает (генерирует) исключение, сигнализирующее о проблеме, и передает объект исключения механизму распространения исключений. Исключения можно генерировать вручную с помощью инструкции `raise`.

Под *обработкой* исключения понимают получение объекта исключения от механизма распространения исключений и выполнение восстановительных действий. Если программа не обрабатывает исключение, то ее выполнение немедленно прекращается и выводится трассировочная информация. Однако программа может обработать исключение и продолжить выполнение, несмотря на возникшие ошибки.

В Python исключения используются также для индикации особых ситуаций, которые не являются следствием ошибок или других проблем. Например, как отмечалось в разделе "Итераторы" главы 3, вызов встроенной функции `next` для итератора, элементы которого полностью исчерпаны, сопровождается генерацией исключения `StopIteration`. Это не является ошибкой. Более того, эту ситуацию нельзя рассматривать как необычную, поскольку элементы большинства итераторов в конечном счете исчерпываются. Поэтому оптимальные стратегии проверки и обработки исключений в Python отличаются от тех, которые в других языках программирования могли бы считаться наилучшими, и данные отличия анализируются в разделе "Стратегии контроля ошибок". В этой главе также обсуждаются модуль `logging` стандартной библиотеки Python (раздел "Журналирование ошибок") и инструкция `assert` (раздел "Инструкция `assert`").

Инструкция try

Механизм обработки исключений Python предоставляется инструкцией try. Это составная инструкция, которая может принимать одну из следующих двух форм:

- предложение try, за которым следует одно или несколько предложений except (и ровно одно необязательное предложение else);
- предложение try, за которым следует ровно одно предложение finally.

Инструкция try также может иметь предложения except (и необязательное предложение else), за которыми следует предложение finally (раздел "Инструкция try/except/finally").

Инструкция try/except

Инструкция try в форме try/except выглядит так.

```
try:
    инструкции
except [выражение [as цель]]:
    инструкции
[else:
    инструкции]
```

Эта форма инструкции try содержит одно или несколько предложений except и необязательное предложение else.

Тело каждого блока except называется *обработчиком исключений*. Обработчики исключений выполняются только в том случае, если выражение, указанное в инструкции except, соответствует объекту исключения, распространяющегося из блока try. В приведенной выше инструкции try выражение — это класс (или кортеж классов, заключенных в круглые скобки), причем данному выражению при сопоставлении соответствует любой экземпляр указанных классов или любых их подклассов. Необязательный параметр *цель* — это идентификатор переменной, которую Python связывает с объектом исключения непосредственно перед выполнением обработчика исключения. Кроме того, обработчик может получить текущий объект исключения, вызвав функцию exc_info модуля sys (см. описание в табл. 7.3).

Ниже приведен пример инструкции try в форме try/except.

```
try:
    1/0
except ZeroDivisionError:
    print('Перехвачена попытка деления на 0')
```

Если инструкция try содержит несколько предложений except, то механизм распространения исключений просматривает их поочередно в том порядке, в каком они

указаны, и выполняет в качестве обработчика первое из предложений except, в котором найдено соответствие объекту исключения.

Обработчики специфических исключений должны предшествовать обработчикам более общих исключений

Обработчики специфических исключений всегда должны предшествовать обработчикам более общих исключений. Несоблюдение этого правила приведет к тому, что обработчики более специфических исключений никогда не смогут быть выполнены.

В последнем из предложений except выражение может отсутствовать. Такое предложение обрабатывает любое исключение, достигшее его в процессе своего распространения. Обычно необходимость в подобной безусловной обработке исключений возникает, хотя и редко, в функциях-обертках, которые, прежде чем возбудить исключение, должны выполнить некоторые дополнительные операции (раздел "Инструкция raise").

Избегайте использования "пустых" предложений except, в теле которых не возбуждаются никакие исключения

Избегайте использования "пустых" предложений except (т.е. предложений except без выражения), если в них не возбуждается вручную какое-либо исключение: такой небрежный стиль может затруднить поиск причин возникновения ошибки, поскольку подобные инструкции слишком общие и могут маскировать ошибки в коде и другие виды логических ошибок.

Распространение исключения прекращается, как только найден обработчик, выражение которого соответствует объекту исключения. Если одна инструкция try вложена (лексически, в исходном коде, или динамически, в вызываемых функциях) в предложение try другой инструкции try, то в процессе распространения исключений первым достигается обработчик, установленный внутренней инструкцией try, и если этот обработчик соответствует данному исключению, то он обрабатывает его. Такой сценарий не всегда может соответствовать тому, что вам нужно, что продемонстрировано в следующем примере.

```
try:
    try:
        1/0
    except:
        print('Перехвачено исключение')
except ZeroDivisionError:
    print('Перехвачена попытка деления на 0')
# вывод: Перехвачено исключение
```

В этом примере из-за того, что обработчик более специфического исключения ZeroDivisionError установлен во внешнем блоке try, он маскируется "всеядным" обработчиком, установленным с помощью предложения except во внутреннем блоке try, и ни на что не влияет, поскольку исключение не распространяется за пределы внутреннего блока try. Более подробно о распространении исключений можно прочитать в разделе "Распространение исключений".

Необязательное предложение else инструкции try/except выполняется только в случае успешного выполнения предложения try. Другими словами, если в теле блока try возникает исключение или его выполнение прекращается одной из инструкций break, continue или return, то предложение else не выполняется. Обработчики, установленные инструкцией try/except, охватывают лишь предложение try, но не предложение else. Предложение else удобно использовать для того, чтобы избежать случайной обработки непредвиденных исключений.

```
print(repr(value), '- ', end=' ')
try:
    value + 0
except TypeError:
    # Тип - не число, возможно - строка?
    try:
        value + ''
    except TypeError:
        print('не число и не строка')
    else:
        print('разновидность строки')
else:
    print('разновидность числа')
```

Инструкция try/finally

Инструкция try в форме try/finally выглядит так.

```
try:
    инструкции
finally:
    инструкции
```

Эта форма содержит ровно одно предложение finally (и не может содержать предложение else, кроме тех случаев, когда инструкция try содержит также одно или несколько предложений except, о чем пойдет речь в разделе "Инструкция try/except/finally").

Предложение finally устанавливает так называемый *обработчик очистки*. Код этого обработчика выполняется всегда, независимо от того, каким образом было завершено выполнение предложения try. В случае распространения исключения из блока try выполнение предложения прекращается, но выполняется обработчик очистки, и исключение продолжает распространяться. Обработчик очистки

выполняется также в том случае, когда предложение try было полностью выполнено или его выполнение было прервано одной из инструкций break, continue или return.

Обработчики очистки, устанавливаемые инструкцией try/finally, предлагают надежный способ явного указания завершающего (финального) кода, который должен быть выполнен при любых обстоятельствах, чтобы обеспечить гарантированное пребывание программы и/или внешних ресурсов (например, файлов, баз данных и сетевых соединений) в нужном состоянии, однако выполнение таких гарантированных финальных операций лучше всего поручать менеджеру контекста, используемому в инструкции with (см. раздел "Инструкция with" в главе 3). Ниже приведен пример использования инструкции try в форме try/finally.

```
f = open(some_file, 'w')
try:
    do_something_with_file(f)
finally:
    f.close()
```

А вот так выглядит соответствующий код, который делает то же самое, но записан в более лаконичной и удобочитаемой форме с использованием инструкции with.

```
with open(some_file, 'w') as f:
    do_something_with_file(f)
```

Избегайте использования инструкций `break` и `return` в предложениях `finally`

Инструкция continue не может входить непосредственно в предложение finally, хотя допускается, чтобы оно содержало инструкцию break или return. Однако использование таких инструкций делает программу менее понятной: выполнение любой из них прекращает процесс распространения исключения, в то время как большинство программистов не будут ожидать, что процесс распространения исключений может прерваться в предложении finally. Такой подход будет сбивать с толку людей, читающих ваш код, и его следует избегать.

Инструкция `try/except/finally`

Инструкция try/except/finally вида

```
try:
...защищаемое предложение...
except ...выражение...:
...код обработчика исключений...
finally:
...код очистки...
```

эквивалентна следующей вложенной инструкции:

```
try:
    try:
        ...защищаемое предложение...
    except ...выражение...:
        ...код обработчика исключений...
finally:
    ...код очистки...
```

Инструкция `try` может иметь несколько предложений `except` и одно необязательное предложение `else`, расположенные перед завершающим предложением `finally`. При любом варианте результат всегда будет аналогичен только что продемонстрированному, т.е. будет таким же, как и в случае вложения инструкции `try/except` со всеми ее предложениями `except` и `else`, если таковые имеются, во внешнюю инструкцию `try/finally`.

Инструкция `with` и менеджеры контекста

Инструкция `with` — это составная инструкция, которая имеет следующий синтаксис.

```
with выражение [as имя_переменной]:
    инструкции
```

Семантика использования инструкции `with` эквивалентна следующему коду.

```
_normal_exit = True
_manager = выражение
имя_переменной = _manager.__enter__()
try:
    инструкции
except:
    _normal_exit = False
    if not _manager.__exit__(*sys.exc_info()):
        raise
    # Исключение не распространяется, если __exit__
    # возвращает значение true
finally:
    if _normal_exit:
        _manager.__exit__(None, None, None)
```

Здесь `_manager` и `_normal_exit` — произвольные внутренние имена, которые не используются нигде в текущей области видимости. Если опустить необязательную часть `as` *имя_переменной* инструкции `with`, то Python все равно вызовет метод `_manager.__enter__()`, но при этом не свяжет результат с каким-либо именем и при выходе из блока вызовет метод `_manager.__exit__()`. Объект, возвращаемый выражением и имеющий методы `__enter__` и `__exit__`, называется *менеджером контекста*.

Инструкция with — это воплощение в Python известной идиомы C++ "получение ресурса есть инициализация" (https://ru.wikipedia.org/wiki/Получение_ресурса_есть_инициализация). Вам нужно лишь написать классы менеджеров контекста, т.е. классы, обладающие двумя специальными методами: __enter__ и __exit__. Метод __enter__ должен вызываться без аргументов. Метод __exit__ должен вызываться с тремя аргументами, которые все имеют значение None, если выполнение тела завершается без распространения исключений, а в противном случае задают тип, значение и трассировочную информацию об исключении. Такой подход обеспечивает то же самое гарантированное выполнение завершающего кода, которое обеспечивается парами конструктор/деструктор по отношению к автоматическим переменным C++ и инструкциям try/finally в Python или Java. Кроме того, это предоставляет возможность выполнения разного завершающего кода для разных типов исключений, если таковые возникают, а также блокирования распространения исключения путем возврата значения true методом __exit__, если в этом возникнет необходимость.

В приведенном ниже примере, который носит исключительно иллюстративный характер, показано, как гарантировать заключение в теги <name> и </name> текста, выводимого на печать.

```
class tag(object):
    def __init__(self, tagname):
        self.tagname = tagname
    def __enter__(self):
        print('<{}>'.format(self.tagname), end='')
    def __exit__(self, etyp, einst, etb):
        print('</{}>'.format(self.tagname))
# будет использовано в качестве тега
tt = tag('sometag')
...
with tt:
    ...Инструкции вывода на печать будут обрамлены
    парой открывающих/закрывающих тегов...
```

Еще проще создавать менеджеры контекста с помощью декоратора contextmanager из модуля contextlib стандартной библиотеки Python, который превращает функцию-генератор в фабрику менеджеров контекста. Модуль contextlib также предлагает функцию-обертку closing (вызывающую метод close некоторого объекта после вызова метода __exit__) плюс дополнительные, более совершенные утилиты в версии v3.

В предположении, что модуль contextlib был предварительно импортирован, менеджер контекста tag может быть реализован с помощью следующего кода:

```
@contextlib.contextmanager
def tag(tagname):
    print('<{}>'.format(tagname), end='')
```

```
    try:
        yield
    finally:
        print('</{}>'.format(tagname))
# используется так же, как и до этого
```

Гораздо больше примеров вместе с подробным их обсуждением вы найдете в документе **PEP 343** (`https://www.python.org/dev/peps/`).

Новое в версии 3.6: теперь модуль `contextlib` включает дополнительный класс `AbstractContextManager`, который можно использовать в качестве базового для создания менеджеров контекста.

Генераторы и исключения

Чтобы облегчить взаимодействие генераторов с исключениями, допускается использовать инструкции `yield` в теле инструкций `try/finally`. Кроме того, объекты генераторов могут иметь два других родственных метода: `throw` и `close`. Предположим, что *g* — объект генератора, созданный с помощью функции-генератора. Тогда в версии v2 метод `throw` имеет сигнатуру

```
g.throw(exc_type, exc_value=None, exc_traceback=None]
```

а в версии v3 — сигнатуру

```
g.throw(exc_value)
```

Вызов метода `g.throw` кодом, вызывающим генератор, приводит к тому же результату, который был бы получен при выполнении инструкции `raise` с теми же аргументами в той точке `yield`, в которой был приостановлен генератор *g*.

Метод `close` генератора не имеет аргументов. Вызов метода `g.close()` приводит к тому же результату, который был бы получен при вызове метода `g.throw(GeneratorExit())`. `GeneratorExit` — это встроенный класс исключения, являющийся прямым потомком класса `BaseException`. После выполнения всех необходимых завершающих операций по освобождению ресурсов метод `close` генератора должен повторно возбудить (или распространить) исключение `GeneratorExit`. Кроме того, генераторы имеют так называемый *финализатор* (специальный метод `__del__`), полностью эквивалентный методу `close`.

Распространение исключений

При возникновении исключения управление передается механизму распространения исключений. Обычный поток управления программы прерывается, и Python выполняет поиск подходящего обработчика исключений. Обработчики исключений устанавливаются посредством предложений `except` инструкции `try`. Обработчики обрабатывают исключения, возникающие в теле предложения `try`, а также исключения, распространяющиеся из функций, вызываемых этим кодом прямо или

косвенно. Если для возникшего в предложении try исключения имеется соответствующий обработчик, то выполнение этого предложения прекращается и управление передается обработчику исключений, а после выхода из него — инструкции, которая располагается сразу после инструкции try.

Если инструкция, при выполнении которой возникло исключение, находится вне предложения try, имеющего подходящий обработчик, то выполнение функции, содержащей данную инструкцию, прекращается и исключение распространяется "вверх" вдоль стека вызовов функций до тех пор, пока не достигнет инструкции, вызвавшей данную функцию. Если вызов функции, выполнение которой было прервано, находится в теле предложения try, имеющего подходящий обработчик, то выполнение данного предложения try прекращается и управление передается обработчику исключений. В противном случае выполнение функции, содержащей указанный вызов, прекращается, а вышеописанный процесс распространения исключения повторяется, продолжая развертывание стека вызовов до тех пор, пока не будет найден подходящий обработчик.

Если найти подходящий обработчик не удается, то по умолчанию программа выводит сообщение об ошибке в стандартный поток ошибок (файл *sys.stderr*). Сообщение об ошибке включает трассировочную информацию, содержащую подробные сведения о функциях, выполнение которых было прервано в процессе распространения исключения. Заданное по умолчанию поведение Python, касающееся вывода сообщений об ошибках, можно поменять, изменив связывание атрибута sys.excepthook (см. описание в табл. 7.3). После вывода сообщения об ошибке Python возвращается в интерактивный сеанс или прекращает выполнение в отсутствие активного интерактивного сеанса. В случае исключений класса SystemExit выполнение программы прекращается без вывода сообщения об ошибке, после чего осуществляется выход из интерактивного сеанса, если программа выполнялась в интерактивном режиме.

Для демонстрации того, как работает механизм распространения исключений, воспользуемся следующими функциями.

```
def f():
    print('в f, перед 1/0')
    1/0    # возбуждает исключение ZeroDivisionError
    print(в f, после 1/0')

def g():
    print('в g, перед f()')
    f()
    print('в g, после f()')

def h():
    print('в h, перед g()')
    try:
        g()
        print('в h, после g()')
```

```
    except ZeroDivisionError:
        print('перехвачено исключение ZD')
    print('выход из функции h')
```

При вызове функции *h* выводится следующая информация.

```
>>> h()
в h, перед g()
в g, перед f()
в f, перед 1/0
перехвачено исключение ZD
выход из функции h
```

Как видите, ни одна из инструкций, выводящих строки "после", не выполнилась, поскольку все они "отсекались" механизмом распространения исключений.

Функция *h* содержит инструкцию `try` и вызывает в теле предложения `try` функцию *g*. В свою очередь функция *g* вызывает функцию *f*, которая выполняет деление на 0, тем самым возбуждая исключение класса `ZeroDivisionError`. Исключение распространяется вплоть до предложения `except` в функции *h*. Во время фазы распространения исключения выполнение функций *f* и *g* прекращается, и именно поэтому ни одно из их сообщений "после" не выводится. Выполнение предложения `try` также прекращается во время фазы распространения исключения, поэтому его сообщение "после" также не выводится. После выхода из обработчика исключения выполнение программы возобновляется с инструкции, располагающейся сразу вслед за обработчиком в конце блока `try/except` функции *h*.

Инструкция `raise`

Исключение можно возбудить вручную с помощью инструкции `raise`. Это простая инструкция, которая имеет следующий синтаксис (работает в обеих версиях, v2 и v3):

```
raise [выражение]
```

Использование инструкции `raise` без указания выражения разрешено только обработчику исключений (или функции, вызываемой им прямо или косвенно). Такая инструкция `raise` повторно возбуждает объект того же исключения, которое было получено обработчиком. При этом выполнение обработчика прекращается, и механизм распространения исключений продолжает просмотр стека вызовов в поиске других подходящих обработчиков. Инструкцию `raise`, не содержащую выражение, удобно использовать в тех случаях, когда обработчик обнаруживает, что не способен обработать полученное исключение или может обработать его лишь частично, поэтому исключение продолжает распространяться, чтобы позволить обработчикам, находящимся выше в стеке вызовов, выполнить собственные операции по обработке исключения и очистке ресурсов.

Если выражение указано, оно должно представлять экземпляр класса, наследуемого от встроенного класса `BaseException`, и в этом случае Python возбуждает данный экземпляр.

Только в версии v2 выражение может также предоставлять объект класса, инстанциализированный инструкцией `raise` для возбуждения результирующего экземпляра, за которым следует другое выражение, предоставляющее аргумент для инстанциализации. Мы рекомендуем игнорировать подобные сложности, которые отсутствуют в версии v3 и без которых можно обойтись в обеих версиях: для этого достаточно инстанциализировать объект исключения, который вы хотите возбудить, и возбудить этот экземпляр.

Вот пример типичного варианта использования инструкции `raise`.

```
def cross_product(seq1, seq2):
    if not seq1 or not seq2:
        raise ValueError('Аргументы seq не должны быть пустыми')
    return [(x1, x2) for x1 in seq1 for x2 in seq2]
```

Функция `cross_product` возвращает список всех пар, включающих по одному элементу из последовательностей, заданных аргументами, но сначала тестирует оба аргумента. Если оба они пустые, функция возбуждает исключение `ValueError`, а не просто возвращает пустой список, как это обычно происходит в случае генераторов списков.

Проверяйте лишь то, что вам необходимо

Функции `cross_product` не требуется проверять, являются ли последовательности *seq1* и *seq2* итерируемыми: если обе они не являются таковыми, то генератор списка самостоятельно возбудит соответствующее исключение, предположительно — `TypeError`.

После возникновения исключения, возбужденного либо непосредственно Python, либо с помощью инструкции `raise`, решение о том, обработать его или позволить ему распространяться далее вверх по стеку вызовов, принимает вызывающий код.

Не используйте инструкцию `raise` для выполнения дублирующего, избыточного контроля ошибок

Используйте инструкцию `raise` только для возбуждения дополнительных исключений в тех случаях, когда в иных обстоятельствах они не считались бы ошибкой, но спецификация вашей программы определяет их как ошибки. Не используйте инструкцию `raise` для дублирования контроля ошибок, который уже неявно осуществляется интерпретатором Python вместо вас.

Объекты исключений

Исключения — это экземпляры подклассов BaseException. Любое исключение имеет атрибут args, который представляет собой кортеж аргументов, используемых для создания экземпляра. Эта специфическая для каждого вида ошибок информация оказывается полезной для диагностики программы и выполнения восстановительных действий. Некоторые классы исключений интерпретируют аргументы и предоставляют их вам в удобном виде как именованные аргументы.

Иерархия стандартных исключений

Исключения являются экземплярами подклассов BaseException. (Только в версии v2 в интересах обратной совместимости допускается обработка некоторых других объектов в качестве "исключений", но в этой книге вопросы, связанные с поддержкой таких объектов, не рассматриваются.)

Очень важно знать иерархическую структуру классов исключений, поскольку она определяет, какие именно предложения except ответственны за обработку того или иного исключения. Большинство классов исключений расширяют класс Exception, однако классы KeyboardInterrupt, GeneratorExit и SystemExit наследуются непосредственно от класса BaseException и не являются подклассами Exception. Поэтому, например, обработчик, заданный в предложении except Exception as e:, не будет перехватывать исключения KeyboardInterrupt, GeneratorExit и SystemExit (обработчики исключений рассматриваются в разделе "Инструкция try/except"). Обычно экземпляры SystemExit возбуждаются посредством функции exit из модуля sys (см. описание в табл. 7.3). Исключение GeneratorExit рассматривается в разделе "Генераторы и исключения". Когда пользователь нажимает клавиши прерывания — комбинации клавиш <Ctrl+C>, <Ctrl+Break> или любую другую аналогичную комбинацию, — генерируется исключение KeyboardInterrupt.

Только в версии v2 введен еще один подкласс исключения Exception: StandardError. Только классы StopIteration и Warning наследуются непосредственно от класса Exception (класс StopIteration является частью итерационного протокола, рассмотренного в разделе "Итераторы" главы 3; класс Warning рассматривается в разделе "Фильтры" главы 16). Все остальные стандартные исключения наследуются от класса StandardError. Однако в версии v3 это излишнее усложнение устранено. Поэтому в версии v2 иерархия наследования от класса BaseException в целом выглядит примерно так.

```
BaseException
  Exception
    StandardError
      AssertionError, AttributeError, BufferError, EOFError,
      ImportError, MemoryError, SystemError, TypeError
      ArithmeticError
```

```
      FloatingPointError, OverflowError, ZeroDivisionError
    EnvironmentError
      IOError, OSError
    LookupError
      IndexError, KeyError
    NameError
      UnboundLocalError
    RuntimeError
      NotImplementedError
    SyntaxError
      IndentationError
    ValueError
      UnicodeError
        UnicodeDecodeError, UnicodeEncodeError
  StopIteration
  Warning
GeneratorExit
KeyboardInterrupt
SystemExit
```

Существуют и другие подклассы исключений (в частности, у класса `Warning` имеется множество подклассов), но приведенная структура отражает основные черты иерархии классов в версии v2. В версии v3 эта структура проще, поскольку класс `StandardError` в ней отсутствует (а все его прямые подклассы являются прямыми подклассами класса `Exception`); то же самое относится к классам `EnvironmentError` и `IOError` (класс `IOError` — синоним класса `OSError`, который в версии v3 является прямым подклассом класса `Exception`), как показано в следующей сжатой таблице.

```
BaseException
  Exception
    AssertionError
    ...
    OSError    # уже не является подклассом EnvironmentError
    RuntimeError
    StopIteration
    SyntaxError
    ValueError
    Warning
  GeneratorExit
  KeyboardInterrupt
  SystemExit
```

Три подкласса `Exception` никогда не инстанциализируются непосредственно. Они предназначены исключительно для того, чтобы упростить написание кода предложений `except`, обрабатывающего широкий диапазон родственных ошибок. Ниже приведено краткое описание каждого из этих трех классов.

`ArithmeticError`
: Базовый класс для исключений, которые обусловлены ошибками, возникающими при выполнении арифметических операций (исключения `OverflowError`, `ZeroDivisionError`, `FloatingPointError`).

`EnvironmentError`
: Базовый класс для исключений, обусловленных внешними причинами (исключения `IOError`, `OSError`, `WindowsError`; только в версии v2).

`LookupError`
: Базовый класс для исключений, которые генерируются контейнером при получении недопустимого ключа или индекса (исключения `IndexError`, `KeyError`).

Классы стандартных исключений

При возникновении общих ошибок времени выполнения возбуждаются следующие классы исключений.

`AssertionError`
: Не выполняется условие, указанное в инструкции `assert`.

`AttributeError`
: Ошибка при попытке доступа к атрибуту или его установки.

`FloatingPointError`
: Ошибка при выполнении операций над числами с плавающей точкой. Расширяет класс `ArithmeticError`.

`IOError`
: Ошибка при выполнении операций ввода-вывода (например, отсутствие свободного места на диске, не удается найти файл или отсутствие необходимых разрешений для доступа к файлу). В версии v2 расширяет класс `EnvironmentError`, в версии v3 — синоним `OSError`.

`ImportError`
: Инструкции `import` (рассматривается в разделе "Инструкция `import`" главы 6) не удается найти импортируемый модель или имя, импортируемое из модуля[1].

`IndentationError`
: Синтаксический анализатор обнаружил ошибку, обусловленную неправильным использованием отступов. Расширяет класс `SyntaxError`.

`IndexError`
: Целое число, используемое для индексирования последовательности, выходит за границы допустимого диапазона (использование нецелого числа в качестве

[1] **Новое в версии 3.6:** `ModuleNotFoundError` — новый, явный подкласс `ImportError`.

индекса элемента последовательности приводит к возбуждению исключения `TypeError`). Расширяет класс `LookupError`.

`KeyError`
: Ключ, используемый для индексирования отображения, отсутствует в отображении. Расширяет класс `LookupError`.

`KeyboardInterrupt`
: Возникает, когда пользователь нажимает клавиши прерывания (комбинации клавиш <Ctrl+C>, <Ctrl+Break>, клавиша <Delete> и другие клавиши, в зависимости от платформы, обрабатывающей события клавиатуры).

`MemoryError`
: Нехватка памяти.

`NameError`
: Попытка обращения к переменной, имя которой не найдено.

`NotImplementedError`
: Возбуждается абстрактными базовыми классами для индикации отсутствия требуемого переопределения метода в конкретном подклассе.

`OSError`
: Возбуждается функциями модуля `os` (рассматривается в разделах "Модуль `os`" главы 10 и "Выполнение других программ" главы 14) для индикации платформозависимых ошибок. В версии v2 расширяет класс `EnvironmentError`. Только в версии v3 имеет множество подклассов, о которых речь пойдет в разделе "Класс `OSError` и его подклассы (только в версии v3)".

`OverflowError`
: Возникает в тех случаях, когда результат выполнения операции над целыми числами слишком велик для представления в виде целого числа. Расширяет класс `ArithmeticError`.

Исключение `OverflowError` оставлено только для обеспечения обратной совместимости

В современных версиях Python эта ошибка не может возникать: целые числа, слишком большие для представления в виде целого числа, неявно преобразуются в "длинные целые" (тип `long`) без возбуждения исключения (в действительности в версии v3 тип `long` отсутствует, и все целые числа принадлежат к типу `int`). Это стандартное исключение остается встроенным в интересах обеспечения обратной совместимости (для поддержки старого кода, который возбуждает это исключение или перехватывает его). Не используйте это исключение в новом коде.

`SyntaxError`
: Синтаксический анализатор столкнулся с синтаксической ошибкой.

`SystemError`
: Интерпретатор столкнулся с внутренней ошибкой в самом Python или внешнем модуле. Информацию о таких исключениях следует направить авторам и группам поддержки Python или данного модуля расширения, сопроводив ее подробным описанием проблемы и, по возможности, исходным кодом, позволяющим воспроизвести возникновение данного исключения.

`TypeError`
: Попытка применения операции или функции к объекту несоответствующего типа.

`UnboundLocalError`
: Возникает при попытке обращения к несвязанной локальной переменной. Расширяет класс `NameError`.

`UnicodeError`
: Возникает при попытке преобразования строки Unicode в байтовую строку или наоборот.

`ValueError`
: Возникает при попытке применения операции или функции к объекту, который имеет подходящий тип, но недопустимое значение, и при этом ситуация не может быть описана более специфичным исключением (например, `KeyError`).

`WindowsError`
: Возбуждается функциями модуля `os` (рассматривается в разделах "Модуль `os`" главы 10 и "Выполнение других программ" главы 14) для индикации ошибок, специфических для платформы Windows. Расширяет класс `OSError`.

`ZeroDivisionError`
: Делитель (правый операнд оператора `/`, `//` или `%` или второй аргумент встроенной функции `divmod`) равен нулю. Расширяет класс `ArithmeticError`.

Класс `OSError` и его подклассы (только в версии v3)

В версии v3 класс `OSError` вбирает многие ошибки, такие как `IOError` и `socket.error`, которым в версии v2 соответствуют отдельные классы. Взамен этого в версии v3 класс `OSError` наследуется многими полезными подклассами, обеспечивающими более элегантную обработку ошибок, возникающих в окружении. Для получения более подробной информации обратитесь к онлайн-документации (`https://docs.python.org/3/library/exceptions.html#os-exception`).

В качестве примера рассмотрим следующую задачу: требуется прочитать содержимое некоторого файла; если такого файла не существует, вернуть строку, заданную по умолчанию; распространить любое другое исключение, препятствующее

чтению файла (кроме того, которое обусловлено отсутствием файла). Вот так может выглядеть вариант кода, переносимый между версиями v2/v3.

```
import errno

def read_or_default(filepath, default):
    try:
        with open(filepath) as f:
            return f.read()
    except IOError as e:
        if e.errno == errno.ENOENT:
            return default
        else:
            raise
```

В версии v3 этот код можно значительно упростить за счет использования подкласса `OSError`.

```
def read_or_default(filepath, default):
    try:
        with open(filepath) as f:
            return f.read()
    except FileNotFoundError:
        return default
```

Подкласс `FileNotFoundError` класса `OSError`, доступный только в версии v3, позволяет упростить решение этой часто встречающейся задачи и выразить его в виде кода более непосредственным образом.

Исключения, "обертывающие" другие исключения или трассировочную информацию

Иногда приходится сталкиваться с исключением, возникающим в процессе обработки другого исключения. В версии v2 с этим мало что можно сделать, разве что, возможно, создать собственные классы исключений и организовать их обработку, как показано в следующем разделе. В этом отношении в версии v3 предлагается гораздо более существенная поддержка. В версии v3 каждый экземпляр исключения хранит собственный трассировочный объект. Используя метод `with_traceback`, вы можете создать другой экземпляр исключения, содержащий другой трассировочный объект. Более того, в версии v3 автоматически запоминается информация о том, какое исключение обрабатывается в качестве "контекста" любого исключения, возникшего в процессе обработки.

Рассмотрим следующий пример, выполнение которого намеренно прерывается необрабатываемым исключением.

```
try: 1/0
except ZeroDivisionError:
    1+'x'
```

В версии v2 будет выведена следующая информация, в которой информация об ошибке, вызванной делением на нуль, оказывается скрытой.

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Сравните это с выводом, который будет получен в версии v3.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Как нетрудно заметить, в последнем случае отображается информация, касающаяся как первоначального, так и вновь возникшего исключений.

Если этого вам недостаточно, то в версии v3 существует возможность возбуждать исключения с помощью инструкции `raise e from ex`, которая позволяет указать оба объекта исключений: `e` — распространяющееся исключение и *ex* — его "причина". Для получения более подробных сведений и соответствующих пояснений следует обратиться к документу **PEP 3134** (`http://legacy.python.org/dev/peps/pep-3134/`).

Пользовательские классы исключений

Расширив любой класс стандартных исключений, вы сможете определить собственный класс исключения. Зачастую такой подкласс всего лишь добавляет строку документирования.

```
class InvalidAttribute(AttributeError):
    """Используется для индикации недопустимых атрибутов"""
```

Помещайте в любой пустой класс или функцию строку документирования вместо инструкции `pass`

Как уже отмечалось в разделе "Инструкция `pass`" главы 3, при создании пустого класса вовсе не обязательно использовать инструкцию `pass`. Все правила Python будут соблюдены, если тело класса будет содержать только строку документирования (которую всегда следует включать в интересах документирования класса). В соответствии с установившейся практикой все "пустые" классы (независимо от того, являются ли они классами исключений), как и "пустые" функции, всегда должны содержать строку документирования вместо инструкции `pass`.

С учетом семантики инструкции try/except возбуждение такого определенного пользователем класса исключения, как InvalidAttribute, почти эквивалентно возбуждению исключения соответствующего стандартного суперкласса, AttributeError. Любое предложение except, способное обработать исключение AttributeError, может обработать также исключение InvalidAttribute. Кроме того, клиентский код, которому известен класс вашего пользовательского исключения InvalidAttribute, может обработать его особым образом, не будучи обязанным обрабатывать все остальные случаи, охватываемые классом AttributeError, если это не запланировано заранее.

```
class SomeFunkyClass(object):
  """Показан лишь основной код"""
  def __getattr__(self, name):
    """Лишь поясняет вид ошибки доступа к атрибуту"""
    if name.startswith('_'):
      raise InvalidAttribute, 'Неизвестный частный атрибут '+name
    else:
      raise AttributeError, 'Неизвестный атрибут '+name
```

Теперь клиентский код может быть более избирательным по отношению к своим обработчикам.

```
s = SomeFunkyClass()
try:
    value = getattr(s, thename)
except InvalidAttribute, err:
    warnings.warn(str(err))
    value = None
# Во всех остальных случаях возбуждения исключения AttributeError
# оно лишь распространяется, а не перехватывается
```

Определяйте и генерируйте пользовательские классы исключений

Определение и генерирование пользовательских классов исключений вместо обычных стандартных в своих модулях — отличная идея: используя определенные вами исключения, вы упростите вызывающему коду обработку исключений, происходящих из вашего модуля, отдельно от всех остальных исключений.

Отдельного внимания заслуживают пользовательские исключения, выступающие в качестве оболочек по отношению к другим исключениям и добавляющие дополнительную информацию (эта возможность будет для вас особенно полезной, если вы работаете с версией v2, в которой непосредственная поддержка обертывания одних исключений другими не предоставляется). Для получения информации о текущем исключении можно использовать функцию exc_info из модуля sys (см. описание в табл. 7.3). Ваш пользовательский класс исключений может быть определен примерно следующим образом.

```
import sys
class CustomException(Exception):
    """Обертывает произвольное исключение, если оно возникло,
       и добавляет другую информацию."""
    def __init__(self, *args):
        Exception.__init__(self, *args)
        self.wrapped_exc = sys.exc_info()
```

Ниже приведен типичный пример использования этого класса в функции-обертке.

```
def call_wrapped(callable, *args, **kwds):
    try:
        return callable(*args, **kwds)
    except:
        raise CustomException('Исключение, распространенное'
                              'из обернутой функции')
```

Пользовательские исключения и множественное наследование

При работе с пользовательскими исключениями вам иногда будет удобно использовать множественное наследование от стандартного класса исключения и специального класса пользовательского исключения из своего модуля, как показано в приведенном ниже фрагменте.

```
class CustomAttributeError(CustomException, AttributeError):
"""Исключение AttributeError, которое является также
   исключением CustomException."""
```

Теперь, когда ваш код будет возбуждать исключение `CustomAttributeError`, оно будет перехватываться как вызывающим кодом, предназначенным для перехвата всех случаев возникновения исключения `AttributeError`, так и кодом, предназначенным только для перехвата всех случаев возникновения исключения в вашем модуле.

Используйте множественное наследование в пользовательских исключениях

Всякий раз, когда вам приходится принимать решение относительно того, генерировать ли конкретный тип стандартного исключения, например `AttributeError`, или пользовательское исключение, которое вы определили в своем модуле, рассмотрите возможность использования описанного подхода, основанного на множественном наследовании. Убедитесь в том, что вы отчетливо задокументировали этот аспект вашего модуля: поскольку данный прием не относится к числу широко используемых, пользователи вашего модуля могут не ожидать этой возможности, если вы не укажете в понятной и явной форме, что именно вы делаете.

Другие исключения, используемые в стандартной библиотеке

Многие из модулей стандартной библиотеки Python определяют собственные классы исключений, эквивалентные пользовательским классам исключений, которые могут быть определены в ваших модулях. В типичных случаях все функции в упомянутых модулях стандартной библиотеки могут возбуждать исключения этих классов в дополнение к исключениям, входящим в стандартную иерархию, рассмотренную в разделе "Классы стандартных исключений". Например, в версии v2 модуль `socket` предоставляет класс `socket.error`, который происходит непосредственно от встроенного класса `Exception`, и несколько подклассов ошибок с именами `sslerror`, `timeout`, `gaierror` и `herror`. Все функции и методы в модуле `socket`, кроме стандартных исключений, могут возбуждать исключения класса `socket.error` и его подклассов. Мы будем рассматривать основные случаи применения таких классов исключений в главах, посвященных модулям стандартной библиотеки, которые их предоставляют.

Стратегии контроля ошибок

В большинстве языков программирования, поддерживающих исключения, последние возбуждаются лишь в редких случаях. В Python использование исключений считается уместным во всех случаях, когда это приводит к упрощению программы и повышению ее надежности, даже если в результате исключения возбуждаются довольно часто.

Подходы LBYL и EAFP

В соответствии с распространенной в других языках программирования идиомой, на которую иногда ссылаются с помощью аббревиатуры LBYL (Look Before You Leap — хорошенько осмотрись, прежде чем прыгать), перед тем, как пытаться выполнить какую-либо операцию, следует убедиться в том, что ее выполнению ничто не помешает. Такой подход нельзя считать идеальным по ряду причин:

- проверочный код может ухудшить читаемость программы и затруднить понимание основной ее части, выполняющейся в отсутствие ошибок;
- проверка может дублировать значительную часть работы, которая будет выполняться при совершении самой операции;
- программист может легко ошибиться, опустив некоторые необходимые виды проверки;
- между моментами времени, соответствующими выполнению проверки и попытке выполнения операции, ситуация может измениться.

В большинстве случаев в Python более предпочтительна другая идиома: попытка выполнения операции в предложении `try` и обработка возможных исключительных ситуаций в предложениях `except`. Суть этой идиомы хорошо передает девиз Грейс

Хоппер — контр-адмирала ВМФ США и создательницы первого в мире компилятора для языка программирования: "It's Easier to Ask Forgiveness than Permission" ("Проще просить прощения, чем получить разрешение"), или, сокращенно, EAFP. Идиома EAFP лишена всех без исключения недостатков, присущих идиоме LBYL. Ниже приведен пример функции, использующей идиому LBYL.

```
def safe_divide_1(x, y):
    if y==0:
        print('Обнаружена попытка деления на 0')
        return None
    else:
        return x/y
```

В этом случае первым бросается в глаза проверочный код, а основной код оказывается скрытым где-то в конце функции. Ниже приведена эквивалентная функция, в которой используется идиома EAFP.

```
def safe_divide_2(x, y):
    try:
        return x/y
    except ZeroDivisionError:
        print('Обнаружена попытка деления на 0')
        return None
```

В случае идиомы EAFP основной код располагается в самом начале функции в предложении `try`, а исключительные ситуации обрабатываются в предложении `except`, лексически следующем за основным кодом.

Как правильно использовать идиому EAFP

Идиома EAFP представляет собой неплохую стратегию обработки ошибок, но она не является панацеей. В частности, не стоит "разбрасывать сети слишком широко", пытаясь отлавливать ошибки, появления которых вы не ожидаете и которые поэтому не требуется перехватывать. Типичный случай подобного риска проиллюстрирован в приведенном ниже примере (см. описание встроенной функции `getattr` в табл. 7.2).

```
def trycalling(obj, attrib, default, *args, **kwds):
    try:
        return getattr(obj, attrib)(*args, **kwds)
    except AttributeError:
        return default
```

В функции `trycalling` делается попытка вызова метода `attrib` для объекта `obj`, но в случае отсутствия метода с таким именем возвращается значение `default`. Однако данная функция в том виде, в каком она представлена выше, делает *не только* это: она также непреднамеренно скрывает любые исключительные ситуации, в которых исключение `AttributeError` возбуждается в теле искомого метода, и возвращает в подобных случаях значение `default`, не сопровождая это выводом каких-либо

предупреждающих сообщений. Тем самым ошибки в другом коде могут остаться незамеченными. Чтобы функция выполняла именно то, для чего она предназначена, необходимо проявить большую тщательность при написании ее кода.

```
def trycalling(obj, attrib, default, *args, **kwds):
    try:
        method = getattr(obj, attrib)
    except AttributeError:
        return default
    else:
        return method(*args, **kwds)
```

В этом варианте реализации функции `trycalling` вызов функции `getattr`, помещенный в предложение `try`, а значит, охраняемый обработчиком в предложении `except`, отделяется от вызова метода, помещенного в инструкцию `else` и поэтому имеющего возможность распространять любые исключения. Правильный подход к использованию идиомы EAFP предполагает частое использование предложения `else` в инструкциях `try/except` (что соответствует более явному, а значит, лучшему стилю программирования, чем тот, который предполагает размещение неохраняемого кода вслед лишь за полной инструкцией `try/except`).

Обработка ошибок в крупных программах

В крупных программах очень легко допустить ошибку, создавая слишком широкие инструкции `try/except`, особенно если вы уже успели убедиться в мощи идиомы EAFP в качестве общей стратегии контроля ошибок. Комбинация `try/except` оказывается очень широкой, если перехватывает слишком много различных ошибок или ошибку, которая может возникать в многочисленных местах. Последний фактор может стать проблемой в тех случаях, когда вам нужно точно определить, что и где именно пошло не так, а в трассировочной информации стека вызовов отсутствуют необходимые детали (или же вы отключаете, частично или полностью, вывод этой информации). Чтобы повысить эффективность обработки ошибок, вы должны проводить отчетливое различие между ошибками и исключительными ситуациями, появления которых ожидаете (а значит, знаете, как их обрабатывать), и непредвиденными ошибками и исключительными ситуациями, свидетельствующими о наличии огрехов в вашей программе.

Некоторые ошибки или аномалии поведения программы в действительности не являются ошибками и вообще чем-то аномальным: они могут быть всего лишь специальными случаями, возможно — редкими, но тем не менее ожидаемыми, которые вы решаете обрабатывать посредством подхода EAFP, а не LBYL, чтобы избежать многих внутренних недостатков, присущих последнему из упомянутых подходов. В подобных случаях вы должны лишь обрабатывать аномалии, зачастую даже без их журналирования или вывода сообщений о них.

Следите за тем, чтобы ваши конструкции `try/except` были достаточно узкими

Тщательно следите за тем, чтобы ваши конструкции `try/except` оставались как можно более узкими. Используйте небольшие предложения `try`, включающие компактные фрагменты кода, которые не содержат большого количества вызовов других функций, и очень специфические кортежи классов исключений, если они необходимы, для дополнительного анализа деталей исключений в коде вашего обработчика, а также повторно возбуждайте исключение, если вам известно, что оно не относится к тем случаям, которые должны обрабатываться данным обработчиком.

Ошибки и аномалии, зависящие от вводимых пользователем данных или других внешних условий, которые вы не можете контролировать, всегда должны быть в определенной степени ожидаемыми, именно потому, что порождающие их причины находятся вне зоны вашего контроля. В подобных случаях вы должны концентрировать внимание на мягкой обработке аномалий, заключающейся в журналировании их точной природы и выводе соответствующих сообщений и обеспечении возможности продолжения выполнения программы без разрушения ее внутреннего и постоянно хранимого состояния. В таких условиях конструкции `try/except` также должны быть достаточно узкими, хотя этот фактор не является столь же критическим, как в тех случаях, когда вы используете идиому EAFP для структурирования обработки специальных случаев, не являющихся в действительности ошибками.

Наконец, полностью непредвиденные ошибки и аномалии указывают на наличие логических ошибок и просчетов, допущенных при проектировании и написании кода программы. Чаще всего наилучшая стратегия заключается в том, чтобы избегать использования конструкций `try/except` в отношении подобных ошибок и допустить преждевременное прекращение работы программы с выводом сообщений об ошибках и трассировочной информации. (Возможно, вы захотите записывать эту информацию в журнал и/или отображать ее более подходящим образом, используя специфический для приложения перехватчик из модуля `sys.excepthook`, о котором вскоре пойдет речь.) В тех маловероятных случаях, когда ваша программа должна сохранять работоспособность даже в самых критических ситуациях, вполне адекватным решением может быть использование достаточно протяженных инструкций `try/except` с предложениями `try`, защищающими вызовы функций, которые обеспечивают значительную часть функциональности программы, и широкими предложениями `except`.

В случае длительно выполняющихся программ обязательно записывайте в сохраняемый журнал подробную информацию о возникающих аномалиях или ошибках, которую впоследствии можно будет внимательно изучить (целесообразно также выводить сообщения о подобных проблемах в качестве уведомлений о существовании дополнительной информации, подготовленной для дальнейшего изучения). При этом ключевую роль играет обеспечение возможности вернуться к перманентному

состоянию программы, соответствующему некоторой неповрежденной, внутренне согласованной точке. Чтобы позволить долго выполняющимся программам благополучно продолжать выполнение после проявления некоторых из их собственных логических ошибок, а также после воздействия неблагоприятных внешних факторов, можно либо использовать метод контрольных точек (в основном сводящийся к сохранению состояния программы и написанию ее таким образом, чтобы она могла загрузить сохраненное состояние и продолжить с него дальнейшее выполнение; `http://www.cism.ucl.ac.be/Services/Formations/checkpointing.pdf`), либо организовать обработку транзакций (`https://ru.wikipedia.org/wiki/Обработка_транзакций`), но в данной книге эти темы подробно рассматриваться не будут.

Журналирование ошибок

В тех случаях, когда в процессе распространения исключения вдоль всего стека вызовов Python не удается найти подходящий обработчик, интерпретатор, как правило, выводит связанную с ошибкой трассировочную информацию в стандартный поток ошибок (`sys.stderr`), прежде чем прервать выполнение программы. Вы можете перенаправить этот поток в другой файловый объект, пригодный для получения вывода, чтобы эта информация выводилась в другое место, более соответствующее вашим целям.

Если вы хотите изменить объем или тип информации, выводимой в таких случаях, то перенаправления стандартного потока ошибок вам будет недостаточно. В подобных случаях вы можете назначить собственную функцию переменной `sys.excepthook`: Python будет вызывать эту функцию, прерывая выполнение программы в ответ на необработанное исключение. В этой функции вы сможете организовать вывод любой информации, которая, по вашему мнению, будет полезна для диагностики и отладки программы, и направить эту информацию в объект, выбираемый по вашему усмотрению. Например, можно использовать модуль `traceback` (рассматривается в разделе "Модуль `traceback`" главы 16) для форматирования выводимой трассировочной информации. Выполнение программы прервется тогда, когда прервется выполнение вашей функции, выводящей информацию об исключении.

Пакет `logging`

Стандартная библиотека Python предлагает пакет `logging`, позволяющий организовать систематическое и гибкое журналирование сообщений, выводимых вашим приложением. Вы можете написать целую иерархию классов и подклассов `Logger` или связать их объекты с объектами класса `Handler` (и его подклассами). Вы также можете включить экземпляры класса `Filter` в настраиваемые критерии, позволяющие определить, какие именно сообщения должны записываться в журнал и каким именно образом это должно делаться. Выдаваемые сообщения форматируются экземплярами класса `Formatter` — в действительности сами сообщения являются экземплярами класса `LogRecord`. Пакет `logging` включает даже средство

динамического конфигурирования, посредством которого можно динамически устанавливать конфигурационные файлы журналирования путем чтения их из дисковых файлов или даже получения их через выделенный сокет в специальном потоке.

В то время как пакет `logging` отличается устрашающе сложной и мощной архитектурой, пригодной для реализации изощренных стратегий и политик, которые могут потребоваться в огромных и сложных программных системах, в большинстве приложений можно обойтись крохотным подмножеством пакета, используя для этого простые функции, предоставляемые самим модулем `logging`. Прежде всего следует импортировать пакет `logging`, а затем организовать выдачу сообщения, передав его в виде строки любой из функций `debug`, `info`, `warning`, `error` или `critical`, перечисленных в порядке возрастания уровня строгости ошибок. Если передаваемая вами строка содержит спецификаторы формата, такие как `%s` (рассматриваются в разделе "Традиционное форматирование строк с помощью оператора `%`" главы 8), то после нее нужно передать в качестве дополнительных аргументов все значения, подлежащие форматированию в данной строке. Например, не используйте показанный ниже вызов, выполняющий операцию форматирования независимо от того, нужна она или не нужна.

```
logging.debug('foo is %r' % foo)
```

Вместо этого лучше использовать такой вызов:

```
logging.debug('foo is %r', foo)
```

Он выполняет форматирование, только если это необходимо (т.е. тогда и только тогда, когда вызов `debug` должен обеспечивать запись выводимой информации в журнал в зависимости от текущего уровня строгости).

К сожалению, модуль `logging` не поддерживает подход, обеспечивающий более удобочитаемое форматирование (рассматривается в разделе "Форматирование строк" главы 8), а поддерживает лишь устаревший подход, рассмотренный в разделе "Традиционное форматирование строк с помощью оператора `%`" главы 8. К счастью, любые другие спецификаторы форматирования, кроме простых спецификаторов `%s` и `%r`, могут потребоваться вам для целей журналирования сообщений лишь в редких случаях.

По умолчанию для уровня строгости установлено значение `WARNING`, т.е. каждая из функций `warning`, `error` и `critical` выводит диагностическую информацию в журнал, тогда как функции `debug` и `info` этого не делают. Вы можете в любое время изменить это значение, вызвав функцию `logging.getLogger().setLevel` с передачей ей в качестве единственного аргумента одной из соответствующих констант, предоставляемых модулем `logging`: `DEBUG`, `INFO`, `WARNING`, `ERROR` и `CRITICAL`. Например, как только вы выполните следующий вызов, все функции журналирования, от `debug` до `critical`, будут выводить диагностическую информацию в журнал до тех пор, пока вы вновь не измените уровень строгости:

```
logging.getLogger().setLevel(logging.DEBUG)
```

Если вы впоследствии выполните такой вызов:

```
logging.getLogger().setLevel(logging.ERROR)
```

то лишь функции `error` и `critical` будут направлять свой вывод в журнал (вывод в журнал из функций `debug`, `info` и `warning` будет отсутствовать). Это условие также существует до тех пор, пока вы вновь не измените его, и т.д.

По умолчанию журнальный вывод направляется в стандартный поток ошибок вашего процесса (`sys.stderr`, см. описание в табл. 7.3) и использует упрощенный формат (например, он не включает временную метку в каждую выводимую строку). Вы можете управлять этими настройками, инстанциализируя подходящие экземпляры обработчиков, используя подходящий экземпляр форматировщика или создавая и настраивая новый экземпляр регистрирующего объекта для их сохранения. В типичном простом случае, когда вы хотите всего лишь установить эти параметры раз и навсегда, после чего они продолжают действовать на протяжении всего времени выполнения программы, для этого достаточно вызвать функцию `logging.basicConfig`, обеспечивающую довольно простой способ настройки посредством именованных параметров. Функция `logging.basicConfig` выполняет необходимые действия лишь при первом вызове, причем только в том случае, если вы вызываете ее до вызова любой из функций, выполняющих операции журналирования (`debug`, `info` и др.). Поэтому наиболее распространенный способ заключается в том, чтобы вызывать функцию `logging.basicConfig` в самом начале программы. Например, широко используемая идиома использования этой функции в начале программы выглядит примерно так.

```
import logging
logging.basicConfig(
    format='%(asctime)s %(levelname)8s %(message)s',
    filename='/tmp/logfile.txt', filemode='w')
```

Данные настройки обеспечивают отправку всех журналируемых сообщений в файл, снабжая каждое из них временной меткой в удобном для чтения человеком формате, за которой следует уровень строгости, выводимый в поле шириной восемь символов с выравниванием вправо, за которым следует текст самого сообщения.

Несмотря на очень большой объем подробной информации, касающейся пакета `logging` и всех его удивительных возможностей, вы обязательно должны ознакомиться с онлайн-документацией, содержащей его описание (`https://docs.python.org/3/library/logging.html?module-logging`).

Инструкция `assert`

Инструкция `assert` позволяет добавлять в программу отладочные проверки. Эта простая инструкция имеет следующий синтаксис:

```
assert условие[,выражение]
```

Если вы запускаете Python с флагом оптимизации (`-O`, см. раздел "Синтаксис и параметры командной строки" главы 2), инструкция `assert` заменяется нулевой операцией: компилятор не генерирует для нее код. В противном случае инструкция `assert` вычисляет условие. Если условие удовлетворяется, инструкция ничего не делает. Если же условие не удовлетворяется, инструкция `assert` инстанциализирует класс `AssertionError`, используя выражение в качестве аргумента (или, в отсутствие выражения, делает это без использования аргументов), и возбуждает результирующий экземпляр[2].

Инструкции `assert` можно использовать в качестве эффективного средства документирования программ. Если вы хотите констатировать, что в определенной точке программы во время ее выполнения должно удовлетворяться некоторое важное неочевидное условие `C` (такое условие называют *инвариантом* программы), то зачастую для этого лучше использовать инструкцию `assert C`, а не просто пояснительный комментарий.

Преимуществом использования инструкции `assert` является то, что в случае невыполнения условия `C` она немедленно известит вас об этом, возбудив исключение `AssertionError`, если программа была запущена без флага `-O`. После тщательной отладки программы ее следует запускать с флагом `-O`. При этом инструкция `assert` будет заменяться нулевой операцией, а накладные расходы снизятся (сама инструкция `assert` остается в исходном коде, документируя инвариант).

Не переусердствуйте с использованием инструкции `assert`

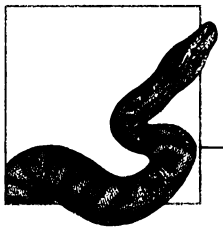
Никогда не используйте инструкцию `assert` для каких-либо других целей, кроме как для отладочной проверки инвариантов программы. Очень распространенной серьезной ошибкой является применение инструкции `assert` в отношении входных значений или аргументов функций: контроль значений аргументов или входных данных лучше всего осуществлять более явным способом, и, в частности, его не должен отключать флаг командной строки.

Встроенная переменная `__debug__`

Когда вы запускаете Python с отключенной оптимизацией (без флага `-O`), встроенная переменная `__debug__` получает значение `True`. При запуске Python с включенной оптимизацией эта переменная получает значение `False`. Кроме того, в этом случае компилятор не генерирует никакого кода для инструкций `if __debug__`.

Чтобы воспользоваться описанной оптимизацией, поместите определение функции, которую вы вызываете только в инструкциях `assert`, в тело инструкции `if __debug__`. Этот прием уменьшает объем скомпилированного кода и ускоряет его работу, если Python запускается с опцией `-O`, а кроме того, улучшает читаемость программы, ясно показывая, что подобные функции существуют лишь для отладки кода.

[2] Некоторые фреймворки сторонних производителей, такие как `pytest`, существенно повышают полезность инструкции `assert`.



6
Модули

Типичная программа на Python состоит из нескольких файлов, в которых содержится исходный код. Каждый такой файл — это *модуль*, группирующий код и данные для повторного использования. Как правило, модули не зависят друг от друга и потому могут повторно использоваться другими программами. Иногда, для того чтобы было легче управлять многочисленными модулями, родственные модули группируют в *пакет* — иерархическую древовидную структуру.

Модуль явно устанавливает зависимости от других модулей посредством инструкций `import` и `from`. В некоторых других языках программирования для организации скрытых каналов связи между модулями используются глобальные переменные. В Python глобальные переменные не являются глобальными для всех модулей и выступают в качестве атрибутов одиночного объекта модуля. Таким образом, связь между модулями в Python всегда устанавливается явно и требует поддержки.

Кроме того, Python предоставляет *модули расширения* — это модули, написанные на других языках программирования, таких как C, C++, Java или C#, но предназначенные для использования в Python. Для кода Python, импортирующего модуль, не имеет значения, является ли данный модуль расширением или собственно модулем Python. Вы всегда можете начать с написания кода модуля на языке Python. Если впоследствии вам понадобится повысить быстродействие программы, вы сможете выполнить рефакторинг модулей и переписать некоторые их части с привлечением низкоуровневых языков, не внося никаких изменений в клиентский код, который использует эти модули. О том, как писать расширения на языках C и Cython, рассказано в главе 24.

В этой главе рассмотрены вопросы, относящиеся к созданию и загрузке модулей. Мы также обсудим группирование модулей в пакеты с помощью служебных утилит Python (`distutils` и `setuptools`), предназначенных для установки распространяемых пакетов и подготовки пакетов к распространению; эта тема более подробно раскрыта в главе 25. Данная глава завершается обсуждением наиболее оптимальных способов управления окружением Python.

Объекты модулей

Модуль — это объект Python с произвольно именованными атрибутами, которые можно связывать и на которые можно ссылаться. Обычно код модуля, носящего имя *name*, хранится в файле *name.py* (раздел "Загрузка модуля").

В Python модули являются объектами (значениями), с которыми можно обращаться как с любыми другими объектами. Таким образом, модули можно передавать в качестве аргументов функциям при их вызове. Точно так же функция может возвращать модуль в качестве результата вызова. Как и любой другой объект, модуль можно связать с переменной, элементом контейнера или атрибутом объекта. Модули могут быть ключами или значениями словаря, а также элементами множества. Например, в словаре `sys.modules` (раздел "Загрузка модуля") объекты модулей хранятся в качестве его значений. Тот факт, что модули можно рассматривать как любые другие значения в Python, часто формулируют в виде утверждения о том, что модули являются объектами *первого класса*.

Инструкция `import`

Чтобы использовать любой исходный файл Python в качестве модуля, другой исходный файл должен импортировать его с помощью инструкции `import`, которая имеет следующий синтаксис:

```
import имя_модуля [as имя_переменной][,...]
```

Вслед за ключевым словом `import` указываются спецификаторы одного или нескольких модулей, разделенные запятыми. В простейшем и наиболее распространенном случае спецификатор *имя_модуля* — это идентификатор, т.е. имя переменной, которую Python связывает с объектом модуля, имеющим то же имя, по завершении выполнения инструкции `import`. В этом случае Python ищет модуль с данным именем. Например, в случае следующей инструкции Python пытается найти модуль с именем `mymodule` и связать переменную `mymodule` с объектом данного модуля в текущей области видимости.

```
import mymodule
```

Спецификатор *имя_модуля* также может быть последовательностью идентификаторов, разделенных точками (`.`). Это позволяет ссылаться на модуль, хранящийся в пакете (раздел "Пакеты").

Если спецификатор модуля включает квалификатор `as`, то Python ищет модуль с именем *имя_модуля* и связывает объект модуля с переменной *имя_переменной*. Например, в случае инструкции

```
import mymodule as alias
```

Python ищет модуль с именем `mymodule` и связывает объект модуля с переменной `alias` в текущей области видимости. Переменная *имя_переменной* всегда должна быть простым идентификатором.

Тело модуля

Тело модуля — это последовательность инструкций, содержащихся в исходном файле модуля. Для указания того, что данный файл является модулем, не требуется использовать какой-либо специальный синтаксис: любой допустимый исходный файл Python может быть использован в качестве модуля. Инструкции, образующие тело модуля, немедленно выполняются, когда модуль впервые импортируется в выполняющуюся программу. В процессе выполнения тела модуля объект модуля уже создан и с ним уже связана соответствующая запись в словаре `sys.modules`. По мере выполнения тела модуля пространство имен (глобальное) модуля постепенно заполняется.

Атрибуты объектов модулей

Инструкция `import` создает новое пространство имен, содержащее все атрибуты данного модуля. Для доступа к атрибутам в этом пространстве имен следует использовать имя или псевдоним модуля в качестве префикса:

```
import mymodule
a = mymodule.f()
```

или

```
import mymodule as alias
a = alias.f()
```

Обычно атрибуты объекта модуля связываются с помощью инструкций в теле модуля. Если инструкция в теле модуля связывает (глобальную) переменную, то связываемый объект является атрибутом модуля.

Тело модуля существует для связывания атрибутов модуля

Обычно назначением тела модуля является создание атрибутов данного модуля: инструкции `def` создают и связывают функции, инструкции `class` создают и связывают классы, а инструкции присваивания могут связывать атрибуты любого типа. Ради ясности и чистоты кода следует избегать делать что-либо другое на верхнем логическом уровне в теле кода, кроме связывания атрибутов модуля.

Атрибуты модуля можно также связывать вне тела (например, в других модулях), обычно с использованием синтаксиса ссылок на атрибут вида `M.`*`имя`* (где *`M`* — любое выражение, значением которого является модуль, а идентификатор *`имя`* — имя атрибута). Однако для ясности обычно лучше все же связывать атрибуты модуля в теле собственного модуля.

Инструкция import устанавливает некоторые атрибуты модуля сразу после создания объекта модуля и до выполнения тела данного модуля. Атрибут __dict__ — это объект словаря, который используется модулем в качестве пространства имен для своих атрибутов. В отличие от других атрибутов модуля, словарь __dict__ не доступен коду внутри модуля в качестве глобальной переменной. Все остальные атрибуты в модуле являются элементами словаря __dict__ и доступны коду внутри модуля в качестве глобальных переменных. Атрибут __name__ — это имя модуля; атрибут __file__ — имя файла, из которого загружен модуль. В других атрибутах, имена которых начинаются и заканчиваются двойными символами подчеркивания, хранятся другие метаданные модуля (в версии v3 количество таких атрибутов возросло во всех модулях).

Для любого объекта модуля *M*, любого объекта *x* и любого идентификатора строки *S* (за исключением __dict__) связывание *M*.*S* = *x* эквивалентно связыванию *M*.__dict__['*S*'] = *x*. Ссылка на атрибут вида *M*.*S* также в основном эквивалентна ссылке *M*.__dict__['*S*']. Единственным отличием является то, что в тех случаях, когда *S* не является ключом в словаре *M*.__dict__, ссылка *M*.__dict__['*S*'] возбуждает исключение KeyError, тогда как ссылка *M*.*S* — исключение AttributeError. Кроме того, атрибуты модуля доступны во всем коде модуля в качестве глобальных переменных. Иными словами, использование *S* в качестве глобальной переменной в теле модуля эквивалентно использованию *M*.*S* (т.е. *M*.__dict__['*S*']) как в отношении связывания, так и ссылки (однако, если *S* не является ключом в словаре *M*.__dict__, то ссылка на *S* как на глобальную переменную возбуждает исключение NameError).

Встроенные объекты Python

Python предоставляет целый ряд встроенных объектов (рассматриваются в главе 7). Все встроенные объекты являются атрибутами модуля builtins (в версии v2 этот модуль называется __builtin__). Когда Python загружает какой-либо модуль, этот модуль автоматически получает дополнительный атрибут __builtins__, ссылающийся либо на модуль builtins (в версии v2 — __builtin__), либо на его словарь. Python может выбирать, какой из этих вариантов использовать, поэтому полагаться на данный атрибут не стоит. Если вам действительно необходимо получить непосредственный доступ к модулю builtins (а необходимость в этом может возникать лишь в редких случаях), то используйте инструкцию import builtins (в версии v2 — инструкцию import __builtin__ as builtins). В случае обращения к переменной, которую не удается найти ни в локальном, ни в глобальном пространстве имен текущего модуля, Python ищет этот идентификатор в словаре __builtins__ текущего модуля, прежде чем возбудить исключение NameError.

Процедура поиска — единственный механизм, который Python предоставляет вам для того, чтобы ваш код мог получить доступ к встроенным объектам. Имена встроенных объектов не зарезервированы и не закодированы жестко в самом Python. Описанный механизм доступа прост и задокументирован, поэтому ваш код

может использовать его непосредственно (однако не переборщите, иначе от этого пострадают простота и ясность вашей программы). Поскольку Python обращается к встроенным объектам лишь в тех случаях, когда не может разрешить имя в локальной или глобальной области видимости, обычно достаточно определить подмену в одном из этих пространств имен. При этом вы можете добавлять собственные встроенные объекты или подменять обычные встроенные функции собственными, и тогда добавленные или подмененные объекты будут видны всем модулям. В следующем иллюстративном примере, ориентированном на версию v3, показано, как упаковать встроенную функцию в собственную функцию, чтобы разрешить функции `abs()` принимать строковый аргумент (и возвращать измененную строку).

```
# abs принимает числовой аргумент; позволим этой функции
# принимать также строку
import builtins
_abs = builtins.abs      # сохранить исходную встроенную функцию
def abs(str_or_num):
    if isinstance(str_or_num, str):    # если arg - строка
        return ''.join(sorted(set(str_or_num)))  # получить взамен
    return _abs(str_or_num)  # вызвать реальную встроенную функцию
builtins.abs = abs     # переопределить встроенную функцию
                       # с помощью функции-обертки
```

Единственное, что необходимо изменить в этом примере для того, чтобы он мог работать в версии v2, — это заменить инструкцию `import builtins` инструкцией `import __builtin__ as builtins`.

Строки документирования модулей

Если первой строкой в теле модуля является строковый литерал, то Python связывает эту строку с атрибутом строки документирования модуля `__doc__`. Строки документирования рассматривались в разделе "Другие атрибуты объектов функций" главы 3.

Закрытые переменные модулей

Ни одна переменная модуля не является истинно закрытой (частной). Однако в соответствии с принятым соглашением любой идентификатор, имя которого начинается с одиночного символа подчеркивания (_), такой как `_secret`, считается закрытым. Другими словами, ведущий символ подчеркивания сообщает программистам клиентского кода о том, что они не должны пытаться получать непосредственный доступ к такому идентификатору.

Среды разработки и другие инструменты следуют этому соглашению для того, чтобы отличать общедоступные атрибуты модуля (т.е. часть интерфейса модуля) от закрытых (т.е. тех, которые должны использоваться только в модуле).

Придерживайтесь соглашения об использовании ведущего символа подчеркивания в именах закрытых атрибутов

Очень важно, чтобы вы соблюдали соглашение об использовании ведущего символа подчеркивания в именах переменных, имитирующих закрытые переменные, особенно если вы пишете клиентский код, который использует модули, написанные другими людьми. Избегайте использования любых переменных, имена которых начинаются с символа _. Можно предположить, что будущие выпуски этих модулей будут поддерживать их общедоступные интерфейсы, но частные детали их реализации, скорее всего, изменятся, и закрытые атрибуты предназначены для реализации именно таких деталей.

Инструкция `from`

Инструкция `from` позволяет импортировать конкретные атрибуты из модуля в текущее пространство имен. Инструкция `from` имеет два варианта синтаксиса:

```
from имя_модуля import имя_атрибута [as имя_переменной][,...]
from имя_модуля import *
```

Инструкция `from` определяет имя модуля, за которым следует один или несколько спецификаторов, разделенных запятыми. В простейшем и наиболее распространенном случае спецификатор атрибута — это идентификатор *имя_атрибута*, т.е. имя переменной, которую Python связывает с одноименным атрибутом, принадлежащим модулю *имя_модуля*, например:

```
from mymodule import f
```

Кроме того, *имя_модуля* может также быть последовательностью идентификаторов, разделенных точками (`.`), что позволяет ссылаться на модуль, хранящийся в пакете (раздел "Пакеты").

Если спецификатор атрибута включает квалификатор `as`, то Python получает из модуля значение атрибута *имя_атрибута* и связывает его с переменной *имя_переменной*, например:

```
from mymodule import f as foo
```

Учтите, что *имя_атрибута* и *имя_переменной* всегда должны быть простыми идентификаторами.

При желании вы можете заключить в круглые скобки всю группу спецификаторов атрибутов, следующих за ключевым словом `import` в инструкции `from`. Иногда это может быть полезным, если у вас есть несколько спецификаторов атрибутов и вы хотите разбить одну логическую строку инструкции `from` на несколько логических строк более элегантным способом, чем с помощью символов обратной косой черты (\).

```
from some_module_with_a_long_name import (
  another_name, and_another as x, one_more, and_yet_another as y)
```

Инструкция `from ... import *`

Код, содержащийся непосредственно в теле модуля (а не в теле функции или класса), может использовать символ "звездочка" (`*`) в инструкции `from`:

```
from имя_модуля import *
```

Символ `*` запрашивает связывание всех атрибутов модуля *имя_модуля* в качестве глобальных переменных. Если модуль *имя_модуля* имеет атрибут `__all__`, то значением этого атрибута является список имен атрибутов, связанных этим типом инструкции `from`. В противном случае этот тип инструкции `from` связывает все атрибуты модуля *имя_модуля*, за исключением тех, имена которых начинаются с символов подчеркивания.

Остерегайтесь использовать инструкцию `from M import *` в своем коде

Поскольку инструкция `from` *M* `import *` может связывать произвольный набор глобальных переменных, в ней заложен риск нежелательных побочных эффектов, таких как сокрытие встроенных функций и изменение связывания нужных вам переменных. Если вы используете инструкцию `from` в такой форме, то делать это следует очень расчетливо и лишь в отношении тех модулей, в документации которых явно указано, что они поддерживают подобный способ их использования. Вероятно, лучше всего воздержаться от использования указанной формы инструкции в коде и рассматривать ее лишь в качестве вспомогательного средства, которое в отдельных случаях удобно использовать в интерактивных сеансах работы с Python.

Сравнение инструкций `from` и `import`

Чаще всего инструкция `import` является более предпочтительной по сравнению с инструкцией `from`. Рассматривайте инструкцию `from`, особенно в форме `from` *M* `import *`, как удобное средство, предназначенное лишь для эпизодического использования в интерактивных сеансах работы с Python. Если вы всегда будете получать доступ к модулю *M* с помощью инструкции `import` *M* и обращаться к его атрибутам с помощью явного синтаксиса *M.A*, то ваш код лишь немного потеряет в лаконичности, но при этом значительно выиграет в ясности и читаемости. К числу случаев, в которых использование инструкции может быть оправданным, относится импорт отдельных модулей пакета (раздел "Пакеты"). Но в большинстве случаев целесообразнее использовать инструкцию `import`, а не `from`.

Загрузка модуля

Операции по загрузке модуля реализованы во встроенной функции `__import__` и в процессе своего выполнения используют атрибуты встроенного модуля `sys`

(раздел "Модуль sys" в главе 7). Ваш код может вызывать функцию __import__ непосредственно, но в современном Python поступать так категорически не рекомендуется; вместо этого следует выполнить инструкцию import importlib и вызвать функцию importlib.import_module, передав ей строку с именем модуля в качестве единственного аргумента. Функция import_module возвращает объект модуля или, если операцию импорта не удается выполнить, возбуждает исключение ImportError. В то же время имеет смысл глубже разобраться в семантике функции __import__, поскольку как функция import_module, так и инструкция import зависят от нее.

Чтобы импортировать модуль *M*, функция __import__ сначала просматривает словарь sys.modules, используя строку *M* в качестве ключа. Если ключ *M* имеется в словаре, функция __import__ возвращает соответствующее значение в качестве запрошенного объекта модуля. В противном случае функция __import__ связывает значение sys.modules[*M*] с новым пустым объектом модуля, атрибут __name__ которого содержит строку *M*, а затем ищет подходящий способ инициализации (загрузки) модуля (раздел "Поиск модуля в файловой системе").

Благодаря этому механизму относительно медленная операция загрузки выполняется лишь тогда, когда данный модуль впервые импортируется в текущую выполняющуюся программу. В случае повторного импортирования модуля он не загружается заново, поскольку функция __import__ быстро находит и возвращает из словаря sys.modules запись, соответствующую данному модулю. Таким образом, все операции импортирования модуля, кроме первой, выполняются очень быстро и сводятся лишь к просмотру словаря. (Относительно возможности *принудительной* перезагрузки модуля см. в разделе "Перезагрузка модулей".)

Встроенные модули

Когда загружается модуль, функция __import__ сначала проверяет, является ли он встроенным. Список всех встроенных модулей содержится в кортеже sys.builtin_module_names, но повторное связывание этого кортежа не влияет на загрузку модулей. Когда Python загружает встроенный модуль, он вызывает его функцию инициализации, как и при загрузке любого другого расширения. Кроме того, поиск модулей осуществляется в расположениях, зависящих от платформы, таких как реестр Windows.

Поиск модуля в файловой системе

Если модуль *M* не является встроенным, то функция __import__ ищет файл, содержащий его код, в файловой системе. Функция __import__ поочередно проверяет строки, являющиеся элементами списка sys.path. Каждый элемент списка представляет путь к каталогу или к архивному файлу, упакованному в популярном ZIP-формате. Список sys.path инициализируется при запуске программы с использованием переменной среды PYTHONPATH (рассматривалась в разделе "Переменные среды" главы 2), если она

имеется. Первым в `sys.path` всегда указывается каталог, из которого загружена основная программа. Пустая строка в `sys.path` соответствует текущему каталогу.

Ваш код может изменить или повторно связать `sys.path`, и такие изменения влияют на то, в каких каталогах и ZIP-архивах функция `__import__` осуществляет поиск загружаемых модулей. Однако эти изменения не влияют на те модули, которые к моменту внесения изменений в `sys.path` уже были загружены (а значит, уже записаны в словарь `sys.modules`).

Если при запуске программы в каталоге *PYTHONHOME* обнаружен файл с расширением *.pth*, то содержимое этого файла, каждый элемент которого должен располагаться в отдельной строке, добавляется в список путей, хранящихся в `sys.path`. Файлы с расширением *.pth* могут содержать пустые строки и комментарии, начинающиеся с символа `#`; Python игнорирует такие строки. В этих файлах также могут содержаться инструкции `import` (которые выполняются прежде, чем начнет выполняться ваша программа), но никакие другие инструкции не допускаются.

В процессе просмотра каждого из каталогов и ZIP-архивов, упомянутых в списке `sys.path`, Python выполняет поиск следующих файлов модуля *M*, перечисленных ниже в том порядке, в каком осуществляется поиск.

Файлы с расширениями *.pyd* и *.dll* (Windows) или *.so* (большинство Unix-подобных платформ), соответствующие модулям расширения Python. (В некоторых диалектах Unix используются другие расширения, например *.sl* в HP-UX.) На большинстве платформ расширения не могут загружаться из ZIP-архива — годятся только модули в виде файлов с исходным кодом или скомпилированным байт-кодом.

Файлы с расширением *.py*, соответствующие исходному коду модулей Python.

Файлы с расширениями *.pyc* (или *.pyo* в версии v2, если Python выполняется с опцией `-O`), соответствующие скомпилированным в байт-код модулям Python.

Если обнаружен файл с расширением *.py*, то (только в версии v3) Python ищет также каталог *__pycache__*. Если такой каталог найден, то Python ищет в нем файлы с расширением *.<tag>.pyc*, где *<tag>* — строка, специфическая для версии Python, которая выполняет поиск модуля.

Последний путь, который Python использует в процессе поиска файла модуля *M* — это *__init__.py*, т.е. ищется файл с именем *__init__.py* в каталоге с именем *M* (раздел "Пакеты").

Найдя исходный файл *M.py*, Python (v3) компилирует его в файл *M.<tag>.pyc*, кроме тех случаев, когда был найден скомпилированный файл, созданный позже файла *M.py*, причем компиляция выполнялась в той же версии Python, в которой был создан исходный файл. Если *M.py* компилировался из каталога, разрешающего запись, то Python создает по необходимости каталог `__pycache__` и сохраняет скомпилированный файл в этом подкаталоге файловой системы, чтобы впоследствии не компилировать его заново. (В версии v2 файл *M.py* компилируется в файл *M.pyc* или *M.pyo*, который Python сохраняет в том же каталоге, в котором находится файл *M.py*.) Если найденный скомпилированный файл создан позже исходного файла (это

определяется на основании внутренней временной метки файла, содержащего байт-код, а не даты, записанной в файловой системе), то Python повторно не компилирует модуль.

Получив файл с байт-кодом, созданный посредством компиляции или прочитанный из файловой системы, Python выполняет тело модуля для инициализации объекта модуля. Если модуль является расширением, Python вызывает функцию инициализации модуля.

Основная программа

Обычно выполнение приложения Python начинается со сценария верхнего уровня, также называемого *основной программой* (см. раздел "Программа `python`" в главе 2). Основная программа выполняется подобно любому другому загружаемому модулю, за исключением того, что Python сохраняет скомпилированный байт-код в памяти, но не сохраняет на диске. Модуль основной программы всегда носит одно и то же имя — `'__main__'`, хранящееся как в глобальной переменной `__name__` (атрибут модуля), так и в виде ключа в словаре `sys.modules`.

Не импортируйте *.py*-файл, который используете в качестве основной программы

Не следует импортировать тот же *.py*-файл, который является основной программой. Если вы сделаете это, то модуль загрузится вновь и его тело выполнится еще раз с самого начала в отдельном объекте модуля с другим значением атрибута `__name__`.

Код модуля Python может протестировать, является ли используемый модуль основной программой, проверив, имеет ли глобальная переменная `__name__` значение `'__main__'`. Показанную ниже идиому часто используют для защиты кода, чтобы он выполнялся только в том случае, если модуль запускается в качестве основной программы.

```
if __name__ == '__main__':
```

Если модуль предназначен только для импортирования, он должен нормально проходить блочные тесты, когда выполняется в качестве основной программы (раздел "Модульное и системное тестирование" в главы 16).

Перезагрузка модулей

Python загружает модуль только тогда, когда он в первый раз импортируется во время выполнения программы. Если разработка ведется в интерактивном режиме, то необходимо явно перезагружать модули после каждого их изменения (некоторые среды разработки обеспечивают автоматическую перезагрузку модулей).

Чтобы перезагрузить модуль в версии v3, передайте объект модуля (а не его имя) в качестве единственного аргумента функции `reload` из модуля `importlib` (в версии v2 вызовите вместо этого встроенную функцию `reload`, которая дает тот же результат). Вызов `importlib.reload(M)` обеспечивает гарантированное использование перезагруженной версии *M* клиентским кодом, полагающимся на инструкцию `import` *M* и получающим доступ к атрибутам с помощью синтаксиса *M.A*. Однако вызов `importlib.reload(M)` не оказывает влияния на другие существующие ссылки, связанные с предыдущими значениями атрибутов *M* (например, с помощью инструкции `from`). Другими словами, вызов функции `reload` не влияет на ранее связанные переменные, связывание которых остается прежним. Неспособность функции `reload` повторно связывать такие переменные служит дополнительным поводом к использованию инструкции `import` вместо инструкции `from`.

Функция `reload` не является рекурсивной: перезагрузка модуля *M* не означает, что при этом перезагружаются также другие модули, импортируемые модулем *M*. Вы должны перезагружать каждый модуль по отдельности путем явных вызовов функции `reload`.

Циклический импорт

Python позволяет определять циклические операции импорта. Например, у вас может быть модуль *a.py*, содержащий инструкцию `import b`, в то время как модуль *b.py* содержит инструкцию `import a`.

Если вы все же используете циклический импорт по каким-либо причинам, то должны хорошо понимать, как он работает, чтобы избежать ошибок в своем коде.

Избегайте циклического импорта

На практике почти всегда лучше избегать операций циклического импорта, поскольку циклические зависимости хрупкие и ими трудно управлять.

Предположим, что основной сценарий выполняет инструкцию `import a`. В соответствии с предыдущим обсуждением эта инструкция создает новый пустой объект модуля в виде значения словаря `sys.modules['a']`, после чего начинает выполняться тело модуля `a`. Выполнение в теле модуля `a` инструкции `import b` создает новый пустой объект модуля в виде значения словаря `sys.modules['b']`, после чего начинает выполняться тело модуля `b`. Выполнение тела модуля `a` не может быть продолжено до тех пор, пока не закончится выполнение тела модуля `b`.

В процессе выполнения модулем `b` инструкции `import a` она выясняет, что значение `sys.modules['a']` уже связано, и поэтому связывает глобальную переменную `a` в модуле `b` с объектом модуля `a`. Поскольку выполнение тела модуля `a` в это время блокировано, то обычно на данном этапе модуль `a` заполняется лишь частично. Тогда

любая попытка кода модуля b получить немедленный доступ к атрибуту модуля a, который к этому времени еще не был связан, приведет к возникновению ошибки.

Если в силу каких-то причин вы все же хотите использовать циклический импорт, то вам следует тщательно продумать, в какой очередности каждый из модулей должен связывать свои глобальные переменные, импортировать другие модули и получать доступ к атрибутам других модулей. Вы можете получить больший контроль над последовательностью развития событий, группируя свои инструкции в функции и вызывая эти функции в контролируемой очередности, а не просто полагаться на последовательность выполнения инструкций верхнего уровня в телах модулей. Обычно проще отказаться от циклического импорта, чем пытаться гарантировать совершенно безопасный порядок выполнения операций в случае циклического импорта.

Изменение записей в словаре `sys.modules`

Функция `__import__` никогда не связывает в качестве значения в словаре `sys.modules` любые другие объекты, отличные от объектов модулей. Однако, если функции `__import__` удается найти в словаре `sys.modules` уже имеющуюся запись, она возвращает соответствующее значение независимо от его типа. Инструкции `import` и `from` внутренне полагаются на результат, возвращаемый функцией `__import__`, и поэтому все может закончиться тем, что они будут использовать объекты, не являющиеся модулями. Это позволяет устанавливать экземпляры классов в качестве записей в словаре `sys.modules`, что предоставляет возможность воспользоваться такими, например, средствами, как специальные методы `__getattr__` и `__setattr__` (см. раздел "Универсальные специальные методы" в главе 4). Этот редко используемый прием, не относящийся к числу простых, позволяет импортировать другие объекты, как если бы они являлись модулями, и вычислять их атрибуты на лету. Рассмотрим соответствующий иллюстративный пример.

```
class TT(object):
    def __getattr__(self, name): return 23
import sys
sys.modules[__name__] = TT()
```

Когда этот код импортируется первый раз в качестве модуля, он переопределяет относящуюся к модулю запись в словаре `sys.modules` экземпляром класса *TT*. При попытке получения значения атрибута с произвольным именем для экземпляра этого класса всегда будет возвращаться значение `23`.

Пользовательские операции импорта

Суть одного из усложненных и редко используемых видов функциональности, предлагаемых Python, заключается в возможности изменения семантики части или всех операций `import` и `from`.

Повторное связывание атрибута __import__

Вы можете изменить связывание атрибута __import__ модуля builtin, повторно связав его с собственной функцией-импортером, например, используя общую методику обертывания встроенных функций (см. раздел "Встроенные объекты Python"). Подобное повторное связывание оказывает влияние на все инструкции import и from, которые выполняются после него, и, таким образом, может иметь нежелательные глобальные эффекты. Пользовательская функция-импортер, созданная путем повторного связывания атрибута __import__, должна реализовывать те же интерфейс и семантику, что и встроенная функция __import__, и, в частности, нести ответственность за корректное использование словаря sys.modules.

Избегайте изменения связывания встроенного атрибута __import__

Несмотря на то что повторное связывание атрибута __import__ поначалу может показаться привлекательным подходом, в большинстве случаев, когда требуются пользовательские функции-импортеры, лучше всего реализовать их посредством функций — перехватчиков операции импорта (рассмотрены ниже).

Перехватчики импорта

Python предлагает расширенную поддержку избирательного изменения деталей выполнения операции импорта. Пользовательские функции-импортеры относятся к числу более сложных и редко используемых приемов, однако некоторые приложения могут нуждаться в них для таких, например, целей, как импортирование кода из архивов, отличных от ZIP-файлов, а также из баз данных, сетевых серверов и т.п.

Наиболее подходящим способом удовлетворения подобных завышенных запросов является запись вызываемых объектов-импортеров в качестве элементов атрибутов meta_path и/или path_hooks модуля sys, о чем подробно говорится в документе **PEP 302** (https://www.python.org/dev/peps/pep-0302/), а также, применительно к версии v3, в документе **PEP 451** (https://www.python.org/dev/peps/pep-0451/). Именно так Python перехватывает обращение к модулю zipimport стандартной библиотеки для обеспечения беспрепятственного импорта модулей из ZIP-файлов, о чем упоминалось ранее. Если вы собираетесь интенсивно использовать перехватчики sys.path_hooks и родственные им, то без тщательного изучения документов **PEP 302** и **PEP 451** вам не обойтись. Однако, для того чтобы вы могли получить некоторое представление об этих возможностях, если потребность в них у вас когда-либо возникнет, ниже приведен небольшой иллюстративный пример.

Предположим, в процессе разработки первого наброска некоторой программы вам было бы удобно использовать инструкции import в отношении модулей, которые еще не написаны, получая при том всего лишь сообщения (и пустые модули). Вы можете получить такую функциональность (оставляя в стороне сложности,

связанные с пакетами, и работая только с простыми модулями) путем создания пользовательского модуля импортера, как показано ниже.

```
import sys, types

class ImporterAndLoader(object):
    '''Импортер и загрузчик часто совмещаются в одном классе'''
    fake_path = '!dummy!'
    def __init__(self, path):
        # Обрабатывать только собственный фиктивный маркер пути
        if path != self.fake_path: raise ImportError
    def find_module(self, fullname):
        # Даже не пытаться обрабатывать любое полное имя модуля
        if '.' in fullname: return None
        return self
    def create_module(self, spec):
        # Выполняется только в v3: создает модуль
        # принятым по умолчанию способом
        return None
    def exec_module(self, mod):
        # Выполняется только в v3: заполняет уже
        # инициализированный модуль;
        # в этом примере просто выводится сообщение
        print('ПРИМЕЧАНИЕ: module {!r} еще не написан'.format(mod))
    def load_module(self, fullname):
        # Выполняется только в v2: создает и заполняет модуль
        if fullname in sys.modules: return sys.modules[fullname]
        # В этом примере просто выводится сообщение
        print('ПРИМЕЧАНИЕ: module {!r} еще не написан'.format(fullname))
        # Создать новый пустой модуль, занести его в sys.modules
        mod = sys.modules[fullname] = types.ModuleType(fullname)
        # Минимальная инициализация нового модуля и его возврат
        mod.__file__ = 'dummy_{}'.format(fullname)
        mod.__loader__ = self
        return mod
# Добавить класс в path_hooks, а фиктивный маркер его пути - в path
sys.path_hooks.append(ImporterAndLoader)
sys.path.append(ImporterAndLoader.fake_path)

if __name__ == '__main__':    # самотестирование при запуске в
                              # качестве основного сценария
    import missing_module     # импортировать простой отсутствующий
                              # модуль
    print(missing_module)     # должно успешно выполниться
    print(sys.modules.get('missing_module'))  # также должно успешно
                                              # выполниться
```

В версии v2 необходимо реализовать метод `load_module`, который помимо всего прочего должен выполнить некоторые стереотипные задачи, такие как обработка словаря `sys.modules` и установка таких атрибутов, имена которых начинаются и заканчиваются двумя символами подчеркивания, как `__file__`, для объекта модуля. В версии v3 система сама выполняет эти стереотипные задачи, поэтому нам достаточно написать тривиальные версии методов `create_module` (который в данном случае всего лишь возвращает значение `None`, запрашивая систему создать объект модуля способом, используемым по умолчанию) и `exec_module` (получает объект модуля с уже инициализированными атрибутами, имена которых содержат двойные символы подчеркивания, и надлежащим образом заполняет его значениями, как обычно).

В версии v3 мы могли бы воспользоваться новым мощным понятием *спецификации модуля*. Однако это потребовало бы использования модуля `importlib` стандартной библиотеки, который в большинстве случаев отсутствует в версии v2; более того, в столь небольшом примере эти мощные возможности нам просто не нужны. Вместо этого мы реализовали метод `find_module`, который в любом случае необходим для версии v2, и, несмотря на то что в настоящее время он является устаревшим, работает и в версии v3 в целях обеспечения обратной совместимости.

Пакеты

Пакет — это модуль, содержащий другие модули. Некоторые или все модули пакета могут быть подпакетами (подчиненными пакетами), что приводит к древовидной иерархической структуре пакета. Пакет *P* создается как одноименный подкаталог некоторого каталога, указанного в `sys.path`. Кроме того, пакеты могут существовать в виде ZIP-файлов. В этом разделе предполагается, что пакет находится в файловой системе, однако то же самое относится и к пакетам в виде ZIP-файлов, поскольку они также используют внутреннюю иерархическую структуру ZIP-файла.

Тело модуля *P* содержится в файле *P/__init__.py*. Этот файл должен существовать (это не относится к пакетам пространств имен в версии v3, рассмотренным в разделе "Пакеты пространств имен (только в версии v3)"), даже если он пустой (что означает пустое тело модуля), тем самым сообщая Python, что *P* действительно является пакетом. Тело модуля пакета загружается, когда вы в первый раз импортируете пакет (или любой из модулей пакета), и ведет себя, как любой другой модуль Python. Другие *.py*-файлы в каталоге *P* — это модули пакета *P*. Подкаталоги *P*, содержащие файлы *__init__.py*, являются *подпакетами P*. Вложение пакетов может распространяться на любую глубину.

Модуль *M*, содержащийся в пакете *P*, можно импортировать как *P.M*. Дополнительные точки обеспечивают навигацию по иерархической структуре пакета. (Тело модуля пакета всегда загружается до загрузки любого из содержащихся в пакете модулей.) Если вы используете синтаксис `import P.M`, то переменная *P* связывается с объектом модуля пакета *P*, а атрибут *M* объекта *P* связывается с модулем *P.M*. Если

вы используете синтаксис `import P.M as V`, то с модулем `P.M` связывается непосредственно переменная *V*.

Использование инструкции `from P import M` для импортирования конкретного модуля *M* из пакета *P* абсолютно допустимо и в действительности настоятельно рекомендуется. Инструкция `from P import M as V` также вполне допустима и полностью эквивалентна инструкции `import P.M as V`.

Если содержащийся в пакете *P* модуль *M* выполняет инструкцию `import X`, то в версии v2 Python по умолчанию осуществляет поиск *X* в *M*, прежде чем искать его в `sys.path`. Однако в версии v3 это не так: чтобы избежать двусмысленности семантик, мы настоятельно рекомендуем использовать в версии v2 инструкцию `from __future__ import absolute_import`, чтобы сделать поведение версии v2 в этом отношении аналогичным поведению версии v3. Тогда модуль *M* содержащийся в пакете *P*, может явно импортировать модуль *X*, относящийся к тому же уровню (т.е. также содержащийся непосредственно в пакете *P*), с помощью инструкции `from . import X`.

Совместное использование объектов модулями одного пакета

Простейший и наиболее чистый способ совместного использования (разделения) объектов (например, функций или констант) модулями, содержащимися в пакете *P*, заключается в группировании разделяемых объектов в модуле, которому обычно присваивают имя *P/common.py*. Благодаря этому вы сможете использовать инструкцию `from . import common` в каждом из модулей пакета, которые нуждаются в доступе к общим объектам, а затем обращаться к этим объектам с помощью ссылок `common.f`, `common.K` и т.п.

Специальные атрибуты объектов пакетов

Атрибут `__file__` пакета *P* — это строка, представляющая путь к телу модуля *P*, т.е. путь к файлу *P/__init__.py*. Атрибут `__package__` пакета *P* — это имя пакета *P*.

Тело модуля пакета *P* — т.е. исходный код на языке Python, содержащийся в файле *P/__init__.py*, — может (как и любой другой модуль) определить глобальную переменную `__all__`, позволяющую контролировать, что будет происходить, если другой код на языке Python выполнит инструкцию `from P import *`. В частности, если переменная `__all__` не установлена, то инструкция `from P import *` импортирует не модули, содержащиеся в пакете *P*, а только имена, которые заданы в теле модуля *P* и не начинаются с символа подчеркивания (_). Как бы то ни было, использовать этот подход *не* рекомендуется.

Атрибут `__path__` пакета *P* — это список строк, представляющих пути к каталогам, из которых загружаются модули и подпакеты *P*. Первоначально Python устанавливает с помощью атрибута `__path__` список, включающий единственный элемент: путь к каталогу, содержащему файл *__init__.py*, который является телом модуля пакета. Ваш код может изменить этот список, чтобы повлиять на поиск модулей и подпакетов данного

пакета, осуществляемый впоследствии. Необходимость в применении такой методики возникает редко, но она может быть полезной, если вы хотите поместить модули пакета в несколько разрозненных каталогов. Только в версии v3: рассмотренные ниже пакеты пространств имен — обычный способ решения указанной задачи.

Пакеты пространств имен (только в версии v3)

Только в версии v3, если при выполнении инструкции `import foo` оказывается, что один или несколько каталогов, являющихся непосредственными подкаталогами элементов `sys.path`, называются *foo* и ни в одном из них не содержится файл *__init__.py*, то Python приходит к заключению, что *foo* — это *пакет пространства имен*. Как результат, Python создает (и присваивает элементу словаря `sys.modules['foo']`) объект пакета `foo` без атрибута `__file__`. Атрибут `foo.__path__` — это список всех каталогов, образующих данный пакет (и, как и в случае обычных пакетов, ваш код может вносить изменения в данный список). Следует отметить, что эта возможность относится к числу редко используемых.

Абсолютный и относительный импорт

Как отмечалось в разделе "Пакеты", обычно инструкция `import` осуществляет поиск целевого объекта в одном из расположений, указанных в `sys.path`, — поведение, описываемое термином *абсолютный импорт* (чтобы гарантировать такое же надежное поведение в версии v2, следует использовать инструкцию `from __future__ import absolute_import`). Кроме того, можно явно использовать *относительный импорт*, соответствующий импорту объекта из текущего пакета. При относительном импорте имена модулей или пакетов начинаются с одной или нескольких точек и доступны лишь в случае инструкции `from`. Инструкция `from . import X` ищет модуль или объект с именем *X* в текущем пакете; инструкция `from .X import y` ищет модуль или объект с именем *y* в модуле или подпакете *X*, находящемся в текущем пакете. Если ваш пакет имеет подпакеты, то их код может получать доступ к объектам, расположенным на более высоких уровнях иерархии пакета, путем использования нескольких точек в начале имени модуля или подпакета, помещаемого между ключевыми словами `from` и `import`. Каждая дополнительная точка соответствует переходу вверх на один уровень иерархии каталогов. Чрезмерное использование этой возможности вредит ясности кода, поэтому пользуйтесь ею с осторожностью и лишь тогда, когда это действительно необходимо.

Утилиты распространения (`distutils`) и установки (`setuptools`)

Модули, расширения и приложения Python можно упаковывать и распространять в нескольких формах.

Сжатые архивные файлы
: Обычно это файлы *.zip* или *.tar.gz* (также известны как файлы *.tgz*), причем обе эти формы переносимы, но существует также много других форм сжатия и архивирования деревьев файлов и каталогов.

Самораспаковывающиеся и самоустанавливающиеся исполняемые файлы
: Как правило, это файлы *.exe* для Windows.

Автономные, готовые к выполнению исполняемые файлы, не требующие установки
: Например, файлы *.exe* для Windows, ZIP-архивы с короткими префиксами сценариев для Unix, файлы *.app* для Mac и т.п.

Платформозависимые установочные пакеты
: Например, пакеты *.msi* для Windows, *.rpm* и *.srpm* для многих дистрибутивов Linux, *.deb* для Debian GNU/Linux и Ubuntu, *.pkg* для macOS.

Расширения Python `Wheels` (и `Eggs`)
: Популярные расширения от сторонних производителей (раздел "Архивные форматы *wheels* и *eggs*").

Чтобы установить пакет, распространяемый в виде самоустанавливающегося исполняемого файла или платформозависимого установщика, пользователю достаточно запустить установщик на выполнение. Способ запуска программы-установщика зависит от платформы, но не зависит от языка, на котором была написана программа. О создании таких самоустанавливающихся исполняемых файлов для различных платформ рассказано в главе 25.

Если же пакет распространяется в виде архивного файла или исполняемого файла, который распаковывается, но не устанавливается самостоятельно, то важно, чтобы пакет был написан на Python. В таком случае пользователь прежде всего должен распаковать файл в подходящий каталог, скажем, в каталог *C:\Temp\MyPack* на компьютере Windows или *~/MyPack* в Unix-подобной системе. Среди извлекаемых файлов обязательно должен быть файл с общепринятым именем *setup.py*, который использует средства Python, известные как *утилиты распространения* (пакет `distutils` стандартной библиотеки) или более популярный и мощный пакет `setuptools` от сторонних производителей (`https://pypi.python.org/pypi/setuptools`). В этом случае распространяемый пакет устанавливается так же просто, как и самоустанавливающийся исполняемый файл. Пользователь открывает окно командной строки и переходит в каталог с распакованным архивом. Затем необходимо выполнить примерно следующую команду:

```
C:\Temp\MyPack> python setup.py install
```

(В настоящее время более предпочтительный способ установки пакетов предлагает система управления пакетами `pip`, кратко рассмотренная в разделе "Окружения Python".) Сценарий *setup.py*, запускаемый этой командой `install`, устанавливает

пакет как часть установки Python данного пользователя в соответствии с опциями, определенными автором пакета в сценарии установки. Разумеется, пользователь должен иметь соответствующие разрешения для записи в каталоги установки Python, и в этом случае может потребоваться использование таких команд, позволяющих повышать привилегии, как *sudo*, но еще лучше использовать для установки пакетов виртуальное окружение (раздел "Окружения Python"). По умолчанию утилиты `distutils` и `setuptools` выводят информацию, когда пользователь запускает сценарий *setup.py*. Опция `---quiet`, указываемая непосредственно перед командой `install`, позволяет скрыть большинство деталей (при этом пользователь по-прежнему видит сообщения об ошибках, если таковые имеются). Следующая команда предоставляет подробную справочную информацию относительно пакета `distutils` или `setuptools`, в зависимости от того, какой из этих инструментов автор пакета использовал в своем файле *setup.py*:

```
C:\Temp\MyPack> python setup.py --help
```

Последние выпуски обеих версий, v2 и v3, поставляются с превосходным установщиком `pip` (рекурсивный акроним от "pip installs packages"), который детально документирован в онлайн-руководстве (`https://pip.pypa.io/en/stable/user_guide/`) и отличается простотой использования в большинстве случаев. Команда `pip install` *пакет* находит онлайн-версию указанного пакета (обычно в огромном репозитории PyPI [`https://pypi.python.org/pypi`], насчитывающем почти 100000 пакетов на момент выхода данной книги), загружает его и устанавливает для вас (в виртуальное окружение, если одно из них активно; см. раздел "Окружения Python"). Именно этот простой, но мощный подход использовался авторами данной книги для установки пакетов в более чем 90% случаев.

Даже если вы загрузили пакет локально (скажем, в каталог */tmp/mypack*) по каким-либо причинам (возможно, он отсутствует в PyPI, или вы пытаетесь экспериментировать с версией, которая там еще не сохранена), `pip` по-прежнему сможет установить его: для этого достаточно выполнить команду `pip install --no-index --findlinks=/tmp/mypack`, и `pip` сделает все остальное.

Архивные форматы *wheels* и *eggs*

Формат *wheels* (как и его предшественник *eggs*, все еще поддерживаемый, но не рекомендуемый для будущих разработок) — это архивный формат Python, включающий структурированные метаданные, а также код на Python. Оба формата, особенно *wheels*, предлагают отличные возможности для упаковки и распространения пакетов Python, и средство `setuptools` (вместе с расширением `wheel`, которое, конечно же, легко установить с помощью команды `pip install wheel`) без проблем работает совместно с ними. Более подробно об этих форматах можно прочитать на сайте `pythonwheels.com`, а также в главе 25.

Окружения Python

Типичный программист на Python работает одновременно над несколькими проектами, каждый из которых имеет собственный список зависимостей (как правило, это сторонние библиотеки и файлы данных). Если зависимости для всех проектов установлены в одном и том же интерпретаторе Python, то очень трудно определить, какие зависимости используются теми или иными проектами, и вовсе невозможно управлять проектами, где некоторые зависимости конфликтуют из-за различий в версиях.

Ранние интерпретаторы Python создавались в предположении, что каждая компьютерная система будет иметь установленный в ней "интерпретатор Python", который будет использоваться для обработки любого кода Python, выполняющегося в этой системе. Дистрибутивы операционных систем стали включать Python в свою базовую установку, но поскольку Python активно развивался, от пользователей начали поступать жалобы на то, что они хотели бы использовать более современную версию языка, чем та, которая предоставляется операционной системой.

Возникли методики, позволяющие устанавливать в системе несколько версий языка, но сам процесс установки оставался нестандартным и требовал вмешательства в систему. Эта проблема была частично разрешена введением каталога *site-packages* в качестве репозитория модулей, добавляемых в установку Python, но поддержка использования проектов с конфликтующими требованиями в одном и том же интерпретаторе по-прежнему оставалась невозможной.

Программистам, привыкшим работать с командной строкой, знакомо понятие *окружение оболочки*. Программа оболочки, выполняющаяся в процессе, имеет текущий каталог, переменные, которые могут устанавливаться командами оболочки (что похоже на пространство имен Python), и другие компоненты специфических для процесса данных. Программы на Python получают доступ к окружению оболочки посредством модуля `os.environ`.

Как отмечалось в разделе "Переменные среды" главы 2, на работу Python оказывают влияние различные аспекты окружения оболочки. Например, то, какой интерпретатор будет выполнен в ответ на команду `python` и другие команды, определяется переменной среды `PATH`. Вы вправе рассматривать эти аспекты окружения оболочки, влияющие на выполнение операций в Python, как ваше *окружение Python*. Изменяя его, вы сможете устанавливать, какой интерпретатор Python будет запускаться в ответ на команду `python`, какие пакеты и модули доступны под определенными именами и пр.

Оставьте установленный системой Python для самой системы

Мы рекомендуем вам управлять своим окружением Python. В частности, не создавайте приложения поверх Python, распространяемого вместе с системой. Вместо этого установите другой независимый дистрибутив Python и настройте свое окружение оболочки, чтобы команда `python` запускала вашу локальную установку Python, а не системную.

Используйте виртуальное окружение

Утилита `pip` обеспечила простой способ установки (и впервые — отмены установки) пакетов и модулей в окружении Python. Изменение каталога *site-packages* системного варианта Python по-прежнему требует использования административных привилегий, и того же самого требует выполнение утилиты `pip` (хотя при желании вы сможете выполнить установку в каталог, отличный от *site-packages*), а установленные модули видны всем программам.

Недостающим элементом является возможность внесения контролируемых изменений в окружение Python, обеспечивающих использование конкретного интерпретатора и конкретного набора библиотек Python. Именно эту возможность предоставляют вам *виртуальные окружения*. Виртуальное окружение, создаваемое на основе конкретного интерпретатора Python, копирует компоненты из установки данного интерпретатора или создает ссылки на них. Критично, однако, то, что каждое окружение должно иметь собственный каталог *site-packages*, в который вы сможете устанавливать выбранные вами ресурсы.

Процедура создания виртуального окружения намного проще процедуры установки Python и требует гораздо меньше системных ресурсов (типичное вновь создаваемое виртуальное окружение занимает менее 20 Мбайт). Вы сможете легко создавать и активизировать их по мере необходимости и столь же легко деактивизировать и уничтожать. Виртуальное окружение можно активизировать и деактивизировать сколько угодно раз на протяжении времени его жизни, а если необходимо обновить установленные ресурсы, то можно использовать утилиту `pip`. Удаление дерева каталогов виртуального окружения, когда в нем больше нет необходимости, полностью освобождает занимаемое им дисковое пространство. Время жизни виртуального окружения может исчисляться как минутами, так и месяцами.

Что такое виртуальное окружение

По сути, виртуальное окружение — это автономное подмножество вашего окружения Python, которое можно активизировать и деактивизировать по мере необходимости. Для интерпретатора Python X.Y оно включает, среди всего прочего, каталог *bin*, содержащий интерпретатор Python X.Y, и каталог *lib/pythonX.Y/site-packages*, содержащий предустановленные версии `easyinstall`, `pip`, `pkg_resources` и `setuptools`. Сопровождение отдельных копий этих важных ресурсов, имеющих отношение к распространению пакетов, позволяет обновлять их по мере необходимости, а не вынужденно привязываться к базовому дистрибутиву Python.

Виртуальное окружение располагает собственными копиями (на платформе Windows) дистрибутивных файлов Python или символическими ссылками на них (на других платформах). Оно настраивает значения `sys.prefix` и `sys.exec_prefix`, по которым интерпретатор и различные установочные утилиты определяют местоположение некоторых библиотек. Это означает, что утилита `pip` может устанавливать

зависимости в каталогах *site-packages* виртуального окружения, изолируя их от других окружений. В результате виртуальное окружение переопределяет, какой интерпретатор должен запускаться, когда вы выполняете команду `python`, и какие библиотеки ему доступны, но оставляет нетронутыми большинство других аспектов вашего окружения Python (таких, как переменные `PYTHONPATH` и `PYTHONHOME`). Поскольку его изменение влияет на ваше окружение оболочки, оно также оказывает влияние на любые подоболочки, в которых вы выполняете команды.

Имея разные окружения, вы сможете, например, тестировать в проекте две различные версии одной и той же библиотеки или тестировать свой проект в различных версиях Python (весьма полезно для проверки совместимости вашего кода с версиями v2/v3). Кроме того, вы можете добавлять зависимости в свои проекты Python, даже не обладая особыми привилегиями, поскольку обычно будете создавать свои виртуальные окружения в тех местах, в которые вам разрешена запись.

В течение длительного времени единственным способом создания виртуальных окружений было использование стороннего пакета `virtualenv` (`https://virtualenv.pypa.io/en/stable/`) с поддержкой или без поддержки со стороны пакета `virtualenvwrapper` (`https://virtualenvwrapper.readthedocs.io/en/latest/`), причем оба пакета все еще доступны для версии v2. Подробнее об этих инструментах можно прочитать в руководстве пользователя по созданию пакетов (`packaging.python.org`). Эти инструменты работают также в версии v3, но в выпуске 3.3 добавлен модуль `venv`, впервые делающий виртуальные окружения собственным средством Python.

Новое в Python 3.6: используйте команду **`python -m venv envpath`** вместо команды **`pyvenv`**, которая в настоящее время признана устаревшей.

Создание и удаление виртуальных окружений

В версии v3 команда **`python -m venv envpath`** создает виртуальное окружение (в каталоге *envpath*, который создается в случае необходимости) на основе интерпретатора Python, использованного при выполнении команды. Указав несколько каталогов в виде аргументов, можно создать с помощью одной команды несколько виртуальных окружений, в которые вы затем установите различные наборы зависимостей. Команда `venv` принимает ряд опций, перечисленных в табл. 6.1.

Таблица 6.1. Опции команды `venv`

Опция	Назначение
`--clear`	Удаляет содержимое каталога, прежде чем установить виртуальное окружение
`--copies`	Устанавливает файлы посредством копирования на Unix-подобных платформах, на которых по умолчанию используются символические ссылки

Опция	Назначение
`-h` или `--help`	Выводит краткую информацию о синтаксисе использования команды и список доступных опций
`--system-site-packages`	Добавляет в список путей поиска виртуального окружения стандартный системный каталог *site-packages*, тем самым делая модули базовой установки Python доступными в виртуальном окружении
`--symlinks`	Устанавливает файлы путем использования символических ссылок на платформах, на которых по умолчанию используется копирование файлов
`--upgrade`	Устанавливает выполняющуюся версию Python в виртуальном окружении, заменяя версию окружения, с которой оно создавалось
`--without-pip`	Отменяет обычное поведение, заключающееся в вызове команды `ensurepip` для загрузки установочной утилиты `pip` в виртуальное окружение

Пользователи версии v2 должны использовать команду **`python -m virtualenv`**, которая не допускает указания нескольких каталогов в качестве аргументов.

В качестве примера ниже показан сеанс работы за терминалом, демонстрирующий процесс создания виртуального окружения и структуру созданного дерева каталогов. Как следует из листинга подкаталогов каталога *bin*, данный конкретный пользователь по умолчанию использует интерпретатор версии v3, установленный в каталоге */usr/local/bin*.

```
machine:~ user$ python3 -m venv /tmp/tempenv
machine:~ user$ tree -dL 4 /tmp/tempenv
/tmp/tempenv
├── bin
├── include
└── lib
    └── python3.5
        └── site-packages
            ├── __pycache__
            ├── pip
            ├── pip-8.1.1.dist-info
            ├── pkg_resources
            ├── setuptools
            └── setuptools-20.10.1.dist-info

11 directories
machine:~ user$ ls -l /tmp/tempenv/bin/
total 80
-rw-r--r-- 1 sh wheel 2134 Oct 24 15:26 activate
```

```
-rw-r--r-- 1 sh wheel 1250 Oct 24 15:26 activate.csh
-rw-r--r-- 1 sh wheel 2388 Oct 24 15:26 activate.fish
-rwxr-xr-x 1 sh wheel  249 Oct 24 15:26 easy_install
-rwxr-xr-x 1 sh wheel  249 Oct 24 15:26 easy_install-3.5
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip3
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip3.5
lrwxr-xr-x 1 sh wheel    7 Oct 24 15:26 python->python3
lrwxr-xr-x 1 sh wheel   22 Oct 24 15:26 python3->/usr/local/bin/python3
```

Удаление виртуального окружения сводится к удалению каталога, в котором оно находится (а также всех подкаталогов и файлов в дереве: **`rm -rf envpath`** в случае Unix-подобных систем). Простота удаления — полезное свойство виртуальных окружений.

Модуль `venv` включает средства, облегчающие программное создание настраиваемых виртуальных окружений (например, путем предварительной установки определенных модулей в окружение или выполнения других необходимых операций после того, как окружение было установлено). Все эти возможности самым исчерпывающим образом описаны в онлайн-документации (`https://docs.python.org/3/library/venv.html`), а потому рассмотрение данного API в нашей книге ограничивается вышеприведенным обсуждением.

Работа с виртуальными окружениями

Чтобы использовать виртуальное окружение, вы должны *активизировать* его из обычного окружения своей оболочки. В любой момент времени может существовать только одно виртуальное окружение — операции активизации не укладываются в "стек", подобно вызовам функций. Активизация подготавливает ваше окружение Python к использованию интерпретатора Python виртуального окружения и каталога *site-packages* (вместе с полной стандартной библиотекой). Если вы хотите прекратить использование этих зависимостей, деактивизируйте виртуальное окружение, что сделает вновь доступным ваше стандартное окружение Python. Дерево каталогов виртуального окружения продолжает существовать до тех пор, пока вы не удалите его, поэтому вы вольны активизировать и деактивизировать виртуальное окружение по своему усмотрению.

Активизация виртуального окружения на Unix-подобных платформах требует использования команды оболочки `source`, чтобы команды скрипта активизации могли внести изменения в текущее окружение оболочки. Простой запуск скрипта означал бы, что его команды будут выполнены в подоболочке, а после выхода из подоболочки все изменения будут утеряны. В случае `bash` и других аналогичных оболочек для активизации окружения в каталоге *envpath* следует выполнить такую команду:

```
source envpath/bin/activate
```

Пользователи других оболочек могут воспользоваться скриптами *activate.csh* и *activate.fish*, находящимися в том же каталоге. В случае Windows-систем используйте такую команду:

```
envpath/Scripts/activate.bat
```

В процессе активизации выполняется ряд операций, наиболее важными из которых являются следующие:

- добавление каталога `bin` виртуального окружения в начало переменной среды `PATH` оболочки, чтобы его команды имели приоритет относительно других команд с тем же именем, уже имеющимся в списке `PATH`;
- определение команды деактивизации для удаления всех последствий активизации и возврата окружения Python в первоначальное состояние;
- изменение приглашения оболочки и включение в него имени виртуального окружения;
- определение переменной среды `VIRTUAL_ENV` в качестве пути доступа к корневому каталогу виртуального окружения (сценарии могут использовать эту информацию для интроспекции виртуального окружения).

В результате выполнения этих действий, как только активизируется виртуальное окружение, команда `python` будет запускать ассоциированный с ним интерпретатор. Интерпретатор видит библиотеки (модули и пакеты), которые были установлены в этом виртуальном окружении, а утилита `pip`, которая теперь берется из виртуального окружения, поскольку в процессе установки модуля она также устанавливается в каталоге `bin` виртуального окружения, по умолчанию устанавливает новые пакеты и модули в каталоге *site-packages* виртуального окружения.

Те, кто впервые сталкивается с виртуальным окружением, должны понимать, что оно не связано с каталогом какого-либо проекта. Вполне допускается работать сразу с несколькими проектами, каждый из которых имеет собственное дерево исходных файлов, используя одно и то же виртуальное окружение. Активизируйте его, а затем свободно перемещайтесь между различными местами хранения файлов, и при этом вам будут доступны одни и те же библиотеки (поскольку окружение Python определяется виртуальным окружением).

Когда вы захотите отключить виртуальное окружение и перестать использовать данный набор ресурсов, выполните команду `deactivate`.

Этого будет достаточно для того, чтобы отменить изменения, внесенные командой активизации, путем удаления каталога `bin` виртуального окружения из переменной `PATH`, после чего команда `python` будет вновь запускать ваш обычный интерпретатор. Коль скоро вы не удаляете виртуальное окружение, оно остается доступным для будущего использования путем повторения действий, необходимых для его активизации.

Управление требованиями зависимостей

Ввиду того что виртуальные окружения изначально предназначались для установки с помощью pip, вы не будете удивлены тем, что использование утилиты pip является предпочтительным способом обслуживания зависимостей в виртуальном окружении. Поскольку утилита pip уже имеет обширную документацию, мы ограничиваемся лишь упоминанием о том, чего достаточно для того, чтобы продемонстрировать преимущества ее использования в виртуальных окружениях. Создав виртуальное окружение, активизировав его и установив зависимости, вы сможете определить точные версии этих зависимостей с помощью команды pip freeze.

```
(tempenv) machine:~ user$ pip freeze
appnope==0.1.0
decorator==4.0.10
ipython==5.1.0
ipython-genutils==0.1.0
pexpect==4.2.1
pickleshare==0.7.4
prompt-toolkit==1.0.8
ptyprocess==0.5.1
Pygments==2.1.3
requests==2.11.1
simplegeneric==0.8.1
six==1.10.0
traitlets==4.3.1
wcwidth==0.1.7
```

Перенаправив вывод этой команды в файл *имя_файла*, вы сможете воссоздать тот же набор зависимостей в другом виртуальном окружении с помощью команды **pip install -r *имя_файла***.

Распространяя код, предназначенный для использования другими людьми, разработчики на Python обычно включают в него файл *requirements.txt*, в котором перечисляют необходимые зависимости. Когда вы устанавливаете программное обеспечение из репозитория Python Package Index, pip устанавливает затребованные вами пакеты вместе со всеми указанными зависимостями. В процессе разработки программ удобно иметь файл требований, который можно использовать для добавления необходимых зависимостей в активное виртуальное окружение (если только они уже не установлены) с помощью простой команды **pip install -r requirements.txt**.

Для поддержания одного и того же набора зависимостей в нескольких виртуальных окружениях добавляйте зависимости в каждое из них, используя один и тот же файл требований. Это общепринятый способ разработки проектов, предназначенных для выполнения в нескольких версиях Python: создайте виртуальное окружение на основе одной из требуемых версий, а затем установите в каждое из них зависимости из одного и того же файла требований. В то время как в предыдущем примере использовались точно версионированные спецификации зависимостей, полученные

с помощью команды `pip freeze`, на практике для определения требований к зависимостям и ограничениям можно использовать довольно сложные способы.

Лучшие практики использования виртуальных окружений

Несмотря на то что в Интернете без труда можно найти массу полезных рекомендаций по этому поводу, можно дать один небольшой, но тем не менее замечательный совет относительно того, как лучше всего организовать работу с виртуальными окружениями.

Работая с одними и теми же зависимостями в нескольких версиях Python, полезно указывать номер версии в имени окружения и использовать общий префикс. Таким образом, для проекта *mutex* можно было бы поддерживать окружения *mutex_35* и *mutex_27* для разработки в версиях v3 и v2 соответственно. Когда вам совершенно ясно, для какой версии Python предназначен проект (вспомните, что имя окружения отображается в приглашении оболочки), вероятность тестирования приложения в неподходящей версии значительно снижается. Для управления установкой ресурсов в обоих окружениях используйте общий файл требований.

Регистрируйте только файл (файлы) требований в системе управления версиями (Source Code Control System, SCCS), а не все окружение. Имея файл требований, можно легко воссоздать виртуальное окружение, зависящее только от выпуска Python и требований. Вы распространяете свой проект и предоставляете пользователям самостоятельно решать, в какой версии (версиях) Python его выполнять, создавая подходящее (предпочтительно виртуальное) окружение.

Храните свои виртуальные окружения вне каталогов проектов. Это позволит избежать необходимости явным образом вынуждать систему управления версиями игнорировать их. В действительности не имеет значения, где хранятся окружения, — система `virtualenvwrapper` сохранит их в централизованном расположении.

Ваше окружение Python не зависит от местонахождения вашего проекта в файловой системе. Вы можете активизировать виртуальное окружение, а затем переключаться между ветвями и перемещаться по дереву изменений в SCCS, чтобы использовать окружение там, где вам нужно.

Чтобы исследовать новый модуль или пакет, создайте и активизируйте новое виртуальное окружение, а затем установите интересующие вас ресурсы с помощью команды `pip install`. Вы вольны экспериментировать с этим новым окружением, как вам заблагорассудится, будучи уверенным в том, что не установите неподходящие зависимости в другие проекты.

Может оказаться так, что для экспериментов в виртуальном окружении приходится устанавливать ресурсы, которые не требуются в текущем проекте. Вместо того чтобы засорять свою среду разработки, разветвите ее: создайте новое виртуальное окружение на основе тех же требований плюс тестируемая функциональность.

Впоследствии, для того чтобы сделать эти изменения постоянными, используйте SCCS для обратного слияния изменений исходного кода и требований из ветви.

По желанию можно создавать виртуальные окружения на основе отладочных сборок Python, что предоставит вам доступ к обильной инструментальной информации о производительности вашего кода на Python (и, разумеется, интерпретатора).

Сама разработка собственных виртуальных окружений требует контроля изменений, и простота их создания способствует этому. Предположим, вы недавно выпустили версию 4.3 модуля и хотите протестировать его код с новыми версиями двух его зависимостей. Обладая достаточными навыками, вы *могли бы* заставить `pip` заменить имеющиеся зависимости в вашем существующем виртуальном окружении.

Однако гораздо легче разветвить проект, обновить требования и создать совершенно новое виртуальное окружение на основе обновленных требований. Вы по-прежнему имеете нетронутое исходное виртуальное окружение и можете переключаться между окружениями для исследования специфических аспектов любых аспектов миграции, которые могут возникать. Как только вы адаптируете код таким образом, чтобы все тесты проходили с обновленными зависимостями, вы фиксируете изменения своего кода и требований и выполняете слияние в версию 4.4, чтобы завершить обновление, известив своих коллег о том, что теперь ваш код готов для работы с обновленными версиями зависимостей.

Виртуальные окружения не решают всех проблем, возникающих перед программистом на языке Python. Всегда можно создать более сложные или более общие инструменты. Но, слава Богу, виртуальные окружения работают, и мы должны как можно полнее использовать все предлагаемые ими преимущества.

7

Встроенные объекты и модули стандартной библиотеки

Термин *встроенный* имеет в Python несколько значений. В большинстве контекстов это означает объект, к которому в коде на языке Python возможен непосредственный доступ без использования инструкции `import`. Механизм, используемый Python для того, чтобы обеспечить такой непосредственный доступ, был описан в разделе "Встроенные объекты Python" главы 6. К встроенным типам Python относятся числа, последовательности, словари, множества, функции (все они были рассмотрены в главе 3), классы (см. раздел "Классы Python" главы 4), классы стандартных исключений (см. раздел "Объекты исключений" главы 5) и модули (см. раздел "Объекты модулей" главы 6). В разделе "Модуль `io`" главы 10 рассматривается тип `file` (встроенный в версии v2), тогда как в разделе "Внутренние типы" главы 13 обсуждаются другие встроенные типы, которые обычно используются во внутренних операциях Python. В этой главе предоставлено дополнительное обсуждение основных встроенных типов (раздел "Встроенные типы"), а также встроенных функций, доступных в модуле `builtins` (`__builtins__` в версии v2; раздел "Встроенные функции").

Как отмечалось в разделе "Встроенные объекты Python" главы 6, некоторые модули называют "встроенными", поскольку они являются неотъемлемой частью стандартной библиотеки Python (хотя для доступа к ним необходимо использовать инструкцию `import`), чтобы отличить их от необязательных дополнительных модулей-надстроек, также называемых *расширениями* Python. В этой главе рассматриваются некоторые встроенные модули ядра языка, а именно: модули `sys` (раздел "Модуль `sys`"), `copy` (раздел "Модуль `copy`"), `collections` (раздел "Модуль `collections`"), `functools` (раздел "Модуль `functools`"), `heapq` (раздел "Модуль `heapq`"), `argparse` (раздел "Модуль `argparse`") и `itertools` (раздел "Модуль `itertools`"). В главе 8 обсуждаются встроенные модули ядра, предназначенные

для работы со строками (string — раздел "Модуль string", codecs — раздел "Модуль codecs" и unicodedata — раздел "Модуль unicodedata"), а в главе 9 — модуль re (раздел "Регулярные выражения и модуль re"). Другие модули стандартной библиотеки Python рассмотрены в частях III и IV.

Встроенные типы

В табл. 7.1 приведены основные встроенные типы Python, такие как int, float, dict и др. Более подробную информацию о многих из этих типов и операциях, выполняемых над их экземплярами, см. в главе 3. Кстати, когда в этом разделе термин встречается термин "число", имеется в виду "число, не являющееся комплексным".

Таблица 7.1. Основные встроенные типы

Тип	Описание
bool	`bool(x=False)` Возвращает `False`, если *x* имеет ложное значение, и `True`, если *x* имеет истинное значение. (См. раздел "Булевы значения" в главе 3.) Тип `bool` расширяет тип `int`: встроенные имена `False` и `True` ссылаются на два единственных экземпляра `bool`. Эти экземпляры являются целыми числами, равными `0` и `1` соответственно, но `str(True)` равно `'True'`, а `str(False)` равно `'False'`
bytearray	`bytearray(x=b''[, codec[, errors]])` Изменяемая последовательность байтов (целые числа в диапазоне от `0` до `255`), поддерживающая обычные методы изменяемых последовательностей плюс методы типа `str` (см. ниже). Если *x* — строка `str` в v3 или экземпляр `unicode` в v2, то вы также должны предоставить аргумент *codec* и можете предоставить аргумент *errors*; результат эквивалентен результату вызова `bytearray(x.encode(codec, errors))`. Если *x* — целое число, то оно должно быть большим или равным нулю, и тогда результирующий экземпляр является байтовым массивом длиной *x*, каждый элемент которого инициализируется значением `0`. Если *x* — объект, соответствующий интерфейсу `buffer`, то для инициализации экземпляра используется доступный только для чтения буфер байтов из *x*. В противном случае *x* должен быть итерируемым объектом, возвращающим целые числа в диапазоне от `0` до `255`, которыми инициализируется экземпляр, например `bytearray([1,2,3,4])==bytearray(b'\x01\x02\x03\x04')`
bytes	`bytes(x=b''[, codec[, errors]])` В версии v2 — синоним `str`. В версии v3 — неизменяемая последовательность байтов с теми же немутирующими методами и тем же поведением в отношении инициализации, что и тип `bytearray`. *Внимание:* `bytes(2)` — это `b'\x00\x00'` в версии v3, но `'2'` в версии v2!
complex	`complex(real=0, imag=0)` Преобразует любое число или подходящую строку в комплексное число. Аргумент *imag* может присутствовать только тогда, когда *real* — число: в таком случае он представляет мнимую часть результирующего комплексного числа. См. также раздел "Комплексные числа" в главе 3

Тип	Описание

`dict`	`dict(x={})`
	Возвращает новый словарь с теми же элементами, которые содержит *x*. (Словари рассматривались в разделе "Словари" в главе 3.) Если *x* — словарь, то `dict(x)` возвращает мелкую (поверхностную) копию *x* аналогично вызову `x.copy()`. *x* также может быть итерируемым объектом, элементы которого являются парами (итерируемые элементы, каждый из которых содержит два элемента). В таком случае вызов `dict(x)` возвращает словарь, ключами которого являются первые элементы каждой пары в *x*, а значениями — соответствующие вторые элементы. Другими словами, если *x* — последовательность, то инструкция `c = dict(x)` эквивалентна следующему коду:
	`c = {}`
	`for key, value in x: c[key] = value`
	В дополнение к позиционному аргументу *x* или вместо него можно задать именованные аргументы. Каждый именованный аргумент становится элементом словаря, в котором имя является ключом: это имя может переопределять элемент, существующий в *x*

`float`	`float(x=0.0)` Преобразует любое число или подходящую строку в число с плавающей точкой. См. раздел "Числа с плавающей точкой" в главе 3
`frozenset`	`frozenset(seq=())` Возвращает новый неизменяемый объект множества с теми же элементами, что и в итерируемом объекте *seq*. Если *seq* — неизменяемое множество, то вызов `frozenset(seq)` возвращает саму последовательность *seq*, аналогично вызову `seq.copy()`. См. раздел "Операции над множествами" в главе 3
`int`	`int(x=0, radix=10)` Преобразует любое число или подходящую строку в целое число. Если *x* — число, то выполняется усечение этого числа до целого в направлении 0 путем отбрасывания дробной части. Аргумент *radix* может присутствовать только тогда, когда *x* — строка: в этом случае *radix* — основание системы счисления, которое может иметь значение в диапазоне от 2 до 36, по умолчанию равное 10. Значение *radix* можно явно задать равным 0, тогда в качестве основания системы счисления принимается одно из значений 2, 8, 10 или 16, в зависимости от формы строки *x*, как в случае целочисленных литералов (см. "Целые числа" в главе 3)
`list`	`list(seq=())` Возвращает новый список объектов с теми же элементами, что и в итерируемом объекте *seq*, расположенными в том же порядке. Если *seq* — список, то вызов `list(seq)` возвращает поверхностную копию *seq*, подобно вызову *seq*`[:]`. См. раздел "Списки" в главе 3

Тип	Описание
memoryview	`memoryview(`*x*`)` Здесь *x* должен быть объектом, поддерживающим интерфейс `buffer` (например, объектом `bytes` или `bytearray`; только в версии v3 так ведут себя экземпляры массивов, которые рассматриваются в разделе "Модуль `array`" главы 15).
memoryview	Вызов `memoryview` возвращает объект *m*, который "видит" точное внутреннее представление участка памяти, занимаемого объектом *x*, и способен работать с этим представлением, причем размер каждого элемента *m* в байтах равен *m*`.itemsize` (в версии v2 — всегда 1); `len(`*m*`)` — количество элементов. Объект *m* поддерживает индексирование (в версии v2 возвращается экземпляр `str` длиной 1; в версии v3 — целое число) и выделение срезов (возвращается другой экземпляр `memoryview`, который "видит" соответствующее подмножество участка памяти). *m* — изменяемый объект, если *x* является изменяемым (но размер *m* не может быть изменен, поэтому присваивание срезу должно выполняться из последовательности той же длины, что и срез). Объект *m* предоставляет несколько атрибутов, доступных только для чтения, и несколько методов, доступных только в версии v3; более подробную информацию можно найти в онлайн-документации (`https://docs.python.org/3/library/stdtypes.html#typememoryview`). Двумя полезными методами, доступными как в версии v2, так и в версии v3, являются *m*`.tobytes()` (возвращает данные *m* в виде экземпляра `bytes`) и *m*`.tolist()` (возвращает данные *m* в виде списка целых чисел)
object	`object()` Возвращает новый экземпляр класса `object`, наиболее фундаментального типа в Python. Непосредственные экземпляры типа `object` не имеют функциональности, и их экземпляры используются исключительно в качестве "стражей", т.е. специальных сигнальных объектов, сравнение которых с любым другим объектом всегда дает ложный результат
set	`set(`*seq*`=())` Возвращает новый изменяемый объект множества с теми же элементами, что и в итерируемом объекте *seq*. Если *seq* — множество, то вызов `set(`*seq*`)` возвращает поверхностную копию *seq*, аналогично вызову *seq*`.copy()`. См. раздел "Множества" в главе 3
slice	`slice([`*start*`,] `*stop*`[, `*step*`])` Возвращает объект среза с доступными только для чтения атрибутами `start`, `stop` и `step`, связанными с соответствующими значениями аргументов, каждый из которых, если он не задан, принимает значение `None`. Если индексы имеют положительные значения, то они имеют тот же смысл, что и индексы в вызове `range(`*start*`,` *stop*`,` *step*`)`. Синтаксис выделения среза *obj*`[`*start*`:`*stop*`:`*step*`]` передает объект среза в качестве аргумента методу `__getitem__`, `__setitem__` или `__delitem__` объекта *obj*. Ответственность за интерпретацию объектов среза, которые получают его методы, возлагается на класс объекта *obj*. См. также раздел "Срезы контейнеров" в главе 4

Тип	Описание
str	`str(obj='')` Возвращает компактное строковое представление объекта *obj*. Если *obj* — строка, то вызов `str` возвращает *obj*. См. также описание функции `repr` в табл. 7.2 и описание метода `__str__` в табл. 4.1. В версии v2 — это синоним `bytes`; в версии v3 — это эквивалент `unicode` из версии v2
super	`super(cls, obj)` Возвращает суперобъект объекта *obj* (который должен быть экземпляром класса *cls* или любого его подкласса), пригодный для вызова методов суперкласса. Инстанциализируйте этот встроенный тип только в коде методов. См. раздел "Кооперативный вызов методов суперклассов" в главе 4. В версии v3 достаточно вызвать в методе функцию `super()` без аргументов, и Python автоматически определит *cls* и *obj* посредством интроспекции
tuple	`tuple(seq=())` Возвращает кортеж с теми же элементами, что и в итерируемом объекте *seq*, расположенными в том же порядке. Если *seq* — кортеж, то вызов `tuple` возвращает *seq*, подобно выражению *seq*`[:]`. См. раздел "Кортежи" в главе 3
type	`type(obj)` Возвращает объект типа, который является типом объекта *obj* (т.е. наиболее глубокий производный тип, экземпляром которого является *obj*). Вызов `type(x)` — это то же самое, что вызов метода *x*`.__class__` для любого *x*. Избегайте проверки равенства или идентичности (тождественности) типов (см. приведенную ниже врезку "Избегайте проверки типов")
unicode	`unicode(string[, codec[, errors]])` (Только в версии v2.) Возвращает объект строки Unicode, созданный посредством декодирования байтовой строки *string*, как если бы был вызван метод *string*`.decode(`*codec*`, `*errors*`)`. Здесь *codec* — имя используемого кодека. Если аргумент *codec* отсутствует, то `unicode` по умолчанию использует кодек, заданный по умолчанию (обычно — `'ascii'`). Аргумент *errors*, если он имеется, представляет собой строку, которая определяет способ обработки ошибок декодирования. См. также раздел "Unicode" в главе 8, где, в частности, приведена информация, касающаяся кодеков и ошибок, а также описание метода `__unicode__` в табл. 4.1. В версии v3 строки Unicode принадлежат к типу *str*

Избегайте проверки типов

Чтобы обеспечить надлежащую поддержку наследования, используйте для проверки принадлежности экземпляра к определенному классу не операцию сравнения типов, а функцию `isinstance` (табл. 7.2). Проверка равенства или идентичности типа объекта некоторому другому типу объектов с помощью функции `type(x)` называется *проверкой типа*. Проверка типа — не совсем подходящее средство для разработки

производственных версий кода на Python, поскольку она противоречит принципу полиморфизма. Просто используйте *x* так, как если бы он имел тип, на который вы рассчитываете, и разрешайте возможные проблемы с помощью инструкции `try/except` (см. раздел "Стратегии контроля ошибок" в главе 5). Это так называемая *утиная типизация* (`https://ru.wikipedia.org/wiki/Утиная_типизация`).

Если вам необходимо всего лишь проверить принадлежность к определенному типу, обычно в целях отладки, используйте функцию `isinstance`. Несмотря на то что вызов функции `isinstance(`*x*`,` *atype*`)` также можно назвать проверкой типа в более широком смысле, этот вызов более предпочтителен по сравнению с инструкцией `type(`*x*`) is` *atype*, поскольку он допускает использование объекта *x*, который является экземпляром любого подкласса *atype*, а не только непосредственного экземпляра самого класса *atype*. В частности, функция `isinstance` отлично подходит для проверки принадлежности к определенному абстрактному базовому классу (см. раздел "Абстрактные базовые классы" в главе 4). Эта новейшая идиома известна как *гусиная типизация*.

Встроенные функции

В табл. 7.2 приведены содержащиеся в модуле `builtins` (в версии v2 — `__builtins__`) функции Python (и некоторые типы, которые на практике используются почти так же, как если бы они были функциями), перечисленные в алфавитном порядке. Встроенные имена не являются зарезервированными словами. Идентификатор, являющийся встроенным именем, можно связывать как в локальной, так и в глобальной области видимости (хотя мы рекомендуем избегать этого; см. предостережение ниже). Имена, связанные в локальной или глобальной области видимости, переопределяют имена, связанные во встроенной области видимости: локальные и глобальные имена скрывают встроенные. Также допускается повторное связывание имен, определенных во встроенной области видимости (см. раздел "Встроенные объекты Python" в главе 6).

Не скрывайте встроенные имена

Избегайте сокрытия встроенных имен: впоследствии они могут понадобиться вашему коду. Весьма соблазнительно использовать для собственных переменных естественные имена, такие как `input`, `list` или `filter`, но поступать так категорически не рекомендуется: все они являются именами встроенных типов или функций Python. Если вы не выработаете устойчивую привычку никогда не скрывать встроенные имена собственными, то рано или поздно столкнетесь с таинственными ошибками в своем коде, вызванными исключительно тем, что такое сокрытие произошло случайно.

Несколько встроенных функций работают по-разному в версиях v3 и v2. Чтобы устранить часть этих различий, включайте в модули v2 инструкцию импорта `from future_builtins import *`: благодаря этому встроенные функции `ascii`, `filter`, `hex`, `map`, `oct` и `zip` будут работать так же, как аналогичные им функции в версии v3. (Однако, чтобы использовать в версии v2 функцию `print`, следует использовать инструкцию `from __future__ import print_function`.)

Большинство встроенных функций не поддерживает именованных аргументов

Большинство встроенных функций и типов нельзя вызывать с именованными аргументами, и они допускают только позиционные аргументы. В приведенном ниже списке мы отдельно оговариваем случаи, когда это ограничение не действует.

Таблица 7.2. Встроенные функции

Функция	Описание
`__import__`	`__import__(`*`module_name`*`[, `*`globals`*`[, `*`locals`*`[, `*`fromlist`*`]]])` В современных версиях Python эта функция признана устаревшей; вместо нее используйте функцию `importlib.import_module` (см. раздел "Загрузка модуля" в главе 6)
`abs`	`abs(`*`x`*`)` Возвращает абсолютное значение числа *x*. Если *x* — комплексное число, `abs` возвращает значение квадратного корня из `x.imag**2+x.real**2` (также известное как *модуль комплексного числа*). В противном случае `abs` возвращает `-`*x*, если *x* `< 0`, и *x*, если *x* `>= 0`. См. также описание функций `__abs__`, `__invert__`, `__neg__` и `__pos__` в табл. 4.4

`all`	`all(`*`seq`*`)`
	Здесь *seq* — это любой итерируемый объект (часто — выражение-генератор; см. раздел "Выражения-генераторы" в главе 3). Функция `all` возвращает значение `False`, если любой элемент *seq* имеет ложное значение; в противном случае (включая пустую последовательность *seq*) функция `all` возвращает значение `True`. Подобно операторам `and` и `or`, рассмотренным в разделе "Закорачивание операторов" главы 3, функция `all` прекращает вычисления и возвращает результат сразу же, как только ответ становится известным. Применительно к функции `all` это означает, что вычисления прекращаются сразу после того, как достигнут элемент с ложным значением, но продолжаются для всей последовательности *seq*, если все ее элементы имеют истинное значение. Ниже приведен иллюстративный пример кода, в котором используется функция `all`.
	`if all(x>0 for x in the_numbers):`
	`  print('все числа положительные')`
	`else:`
	`  print('некоторые из чисел не являются положительными')`

Функция	Описание

any	`any(seq)`
	Здесь *seq* — это любой итерируемый объект (часто — выражение-генератор; см. раздел "Выражения-генераторы" в главе 3). Функция `any` возвращает значение `True`, если любой элемент *seq* имеет истинное значение; в противном случае (включая пустую последовательность *seq*) функция `any` возвращает значение `False`. Подобно операторам `and` и `or`, рассмотренным в разделе "Закорачивание операторов" главы 5, функция `any` прекращает вычисления и возвращает результат сразу же, как только ответ становится известным. Применительно к функции `any` это означает, что вычисления прекращаются сразу после того, как достигнут элемент с истинным значением, но продолжаются для всей последовательности *seq*, если все ее элементы имеют ложное значение. Ниже приведен иллюстративный пример кода, в котором используется функция `any`.
	`if any(x<0 for x in the_numbers):`
	`  print('некоторые числа отрицательные')`
	`else:`
	`  print('отрицательные числа отсутствуют')`

ascii	`ascii(x)` Используется только в версии v3; также может использоваться в модуле версии v2, если он начинается с инструкции `from future_builtins import *`. Аналогична функции `repr`, но экранирует все символы, не являющиеся символами ASCII, в возвращаемой строке, поэтому результат весьма схож с тем, который возвращает функция `repr` в версии v2
bin	`bin(x)` Возвращает двоичное строковое представление целого числа *x*
callable	`callable(obj)` Возвращает значение `True`, если объект *obj* может быть вызванным, и `False` — в противном случае. Объект может быть вызванным, если он является функцией, методом, классом, типом или экземпляром класса, имеющего метод `__call__`. См. также описание метода `__call__` в табл. 4.1
chr	`chr(code)` Возвращает строку длиной `1`, которая представляет одиночный символ, соответствующий целочисленному значению *code* в кодировке Unicode (в версии v2 функция `chr` работает только для значений *code* < `256`; чтобы избежать этого ограничения и получить экземпляр типа `unicode` длиной `1`, используйте функцию `unichr`). См. также описание функций `ord` и `unichr` в этой таблице
compile	`compile(string, filename, kind)` Компилирует строку *string* и возвращает объект кода, пригодный для выполнения с помощью функции `exec` или `eval`. Если *string* не

Функция	Описание
	представляет синтаксически корректный код на Python, то функция `compile` возбуждает исключение `SyntaxError`. Если *`string`* представляет составную инструкцию, располагающуюся на нескольких строках, то последним символом должен быть символ `'\n'`. Аргумент *`kind`*, определяющий режим компиляции, должен иметь значение `'eval'`, если *`string`* — выражение, результат вычисления которого предназначается для функции `eval`; в противном случае значением *`kind`* должно быть `'exec'`. Аргумент *`filename`* должен быть строкой, используемой только в сообщениях об ошибке (если таковые возникают). См. также описание функции `eval` в этой таблице и раздел "Функция `compile` и объекты `code`" в главе 13
`delattr`	`delattr(`*`obj`*`, `*`name`*`)` Удаляет атрибут *`name`* из объекта *`obj`*. Вызов `delattr(`*`obj`*`,'`*`ident`*`')` аналогичен инструкции `del` *`obj.ident`*. Если объект *`obj`* имеет атрибут *`name`* лишь потому, что его имеет класс данного объекта (как это обычно бывает, например, с методами *`obj`*), то удалить этот атрибут из самого объекта нельзя. Вы сможете удалить этот атрибут из класса, если метакласс позволит вам это сделать. Если вам удастся удалить атрибут класса, то его не будет иметь ни объект *`obj`*, ни любой другой экземпляр данного класса
`dir`	`dir([`*`obj`*`])` Вызов функции `dir` без аргументов возвращает отсортированный список всех имен переменных, связанных в текущей области видимости. Вызов `dir(`*`obj`*`)` возвращает отсортированный список имен атрибутов объекта *`obj`*, включая те, которые происходят от типа объекта *`obj`* или наследуются. См. также описание функции `vars` в этой таблице
`divmod`	`divmod(`*`dividend`*`, `*`divisor`*`)` Делит одно число на другое и возвращает пару, элементами которой являются частное от деления и остаток. См. также описание метода `__divmod__` в табл. 4.4

`enumerate`	`enumerate(`*`iterable`*`, `*`start`*`=0)`
	Возвращает новый объект итератора, элементами которого являются пары. В каждой такой паре второй элемент — это соответствующий элемент итерируемого объекта, тогда как первый элемент — целое число: *`start`*, *`start`*+1, *`start`*+2... Ниже приведен иллюстративный пример кода, в котором элементы списка `L` целых чисел перебираются в цикле, и список изменяется на месте путем деления на 2 всех элементов с четными значениями.
	`for i, num in enumerate(L):`
	`  if num % 2 == 0:`
	`  L[i] = num // 2`

`eval`	`eval(`*`expr`*`,[ `*`globals`*`[, `*`locals`*`]])` Возвращает результат вычисления выражения *`expr`*. Выражение может быть объектом кода, готового к выполнению вычислений, или строкой; в последнем

Функция	Описание
	случае функция `eval` получает объект кода посредством внутреннего вызова `compile(`*`expr`*`, '<string>', 'eval')`. Функция `eval` вычисляет объект кода, имеющего вид выражения, используя словари *`globals`* и *`locals`* в качестве пространств имен. В случае отсутствия обоих этих аргументов функция `eval` использует текущее пространство имен. Функция `eval` не может выполнять инструкции, она лишь вычисляет выражения. Тем не менее использование функции `eval` может считаться безопасным только в том случае, если источник кода вам известен и вы доверяете ему. См. также раздел "Выражения" в главе 13 и замечания, касающиеся функции `ast.literal_eval`, в разделе "Стандартный ввод" главы 10
`exec`	`exec(`*`statement`*`,[ `*`globals`*`[, `*`locals`*`]])`
	В версии v3 аналогична функции `eval`, но может применяться к любой инструкции и возвращает значение `None`. В версии v2 `exec` работает аналогичным образом, но является инструкцией, а не функцией. В обоих случаях использование функции `exec` может считаться безопасным только в том случае, если источник кода вам известен и вы доверяете ему. См. также раздел "Инструкции" в главе 3
`filter`	`filter(`*`func`*`, `*`seq`*`)`
	В версии v2 возвращает список тех элементов *`seq`*, для которых *`func`* возвращает истинное значение. В качестве *`func`* можно использовать любой вызываемый объект, принимающий единственный аргумент, или значение `None`. Объектом *`seq`* может быть любой итерируемый объект. Если *`func`* — вызываемый объект, то функция `filter` вызывает *`func`* для каждого элемента *`seq`*, подобно следующему генератору списка:
	`[`*`item`* `for` *`item`* `in` *`seq`* `if` *`func(item)`*`]`
	В версии v2, если *`seq`* — строка или кортеж, то результат, возвращаемый функцией `filter`, также является строкой или кортежем, а не списком. Если аргумент *`func`* имеет значение `None`, то функция `filter` проверяет истинность элементов подобно следующему генератору списка:
	`[`*`item`* `for` *`item`* `in` *`seq`* `if` *`item`*`]`
	В версии v3, независимо от типа *`seq`*, функция `filter` возвращает итератор, а не список (или другой тип последовательности), как в версии v2; поэтому в версии v3 функция `filter` эквивалентна выражению-генератору, а не генератору списка
`format`	`format(`*`x`*`, `*`format_spec`*`='')`
	Возвращает `x.__format__(`*`format_spec`*`)` (см. табл. 4.1)
`getattr`	`getattr(`*`obj`*`, `*`name`*`[, `*`default`*`])`
	Возвращает для объекта *`obj`* атрибут, имя которого определяется строкой *`name`*. Вызов `getattr(`*`obj`*`, 'ident')` аналогичен вызову *`obj.ident`*. Если аргумент *`default`* имеется, а имя *`name`* не найдено в *`obj`*, то функция `getattr` возвращает *`default`* вместо возбуждения исключения

Функция	Описание
	`AttributeError`. См. также разделы "Атрибуты и элементы объектов" в главе 3 и "Основные сведения о ссылках на атрибуты" в главе 4
globals	`globals()` Возвращает атрибут `__dict__` вызывающего модуля (т.е. словарь, используемый в качестве глобального пространства имен в точке вызова). См. также описание функции `locals` в данной таблице
hasattr	`hasattr(`*obj*`, `*name*`)` Возвращает значение `False`, если объект *obj* не имеет атрибута с именем *name* (т.е. если вызов `getattr(`*obj*`,`*name*`)` возбуждает исключение `AttributeError`). В противном случае `hasattr` возвращает значение `True`. См. также раздел "Основные сведения о ссылках на атрибуты" в главе 4
hash	`hash(`*obj*`)` Возвращает хеш-код для объекта *obj*. Ключ словаря или элемент множества может выступать в качестве объекта *obj* только в том случае, если *obj* допускает хеширование. Все объекты, проверка равенства которых путем сравнения дает истинный результат, должны иметь одно и то же значение хеш-кода, даже если они принадлежат разным типам. Если тип объекта *obj* не определяет проверку равенства объектов путем их сравнения, то обычно вызов `hash(`*obj*`)` возвращает `id(`*obj*`)`. См. также описание метода `__hash__` в табл. 4.1
hex	`hex(`*x*`)` Возвращает шестнадцатеричное строковое представление целого числа *x*. См. также описание метода `__hex__` в табл. 4.4
id	`id(`*obj*`)` Возвращает целочисленное значение, идентифицирующее объект *obj*. Идентификатор `id` объекта *obj* гарантированно является уникальным постоянным значением на протяжении всего времени жизни данного объекта (но если впоследствии объект *obj* будет удален механизмом сборки мусора, то это значение может быть повторно использовано для идентификации какого-то другого объекта, поэтому полагаться на сохранение и проверку значений `id` не следует). Если для типа или класса операция проверки равенства не определена, то для сравнения и хеширования экземпляров Python использует значение `id`. Для любых объектов *x* и *y* проверка идентичности *x* `is` *y* равносильна проверке `id(`*x*`)==id(`*y*`)`, но ее легче читать и она выполняется быстрее
input	`input(`*prompt*`='')` В версии v2 вызов `input(`*prompt*`)` — это сокращенная форма вызова `eval(raw_input(`*prompt*`))`. Другими словами, в версии v2 функция `input` приглашает пользователя ввести строку, вычисляет результирующую строку в качестве выражения и возвращает полученный результат. Неявный вызов функции `eval` может приводить к возникновению исключения `SyntaxError`

Функция	Описание
	и других исключений. Функция не особенно дружественна к пользователю и не подходит для большинства программ, но иногда ее удобно использовать для проведения небольших экспериментов и включения в экспериментальные сценарии. См. также описание функций `eval` и `raw_input` в данной таблице. В версии v3 функция `input` эквивалентна функции `raw_input` из версии v2, т.е. она всегда возвращает строку типа `str` и не вызывает функцию `eval`
intern	`intern(`*`string`*`)` Гарантирует сохранение строки в таблице интернированных строк и возвращает саму строку или ее копию. Сравнение интернированных строк с целью проверки их равенства выполняется быстрее, чем для других строк, поскольку для такого сравнения вместо оператора `==` можно использовать оператор `is`. Однако сборщик мусора никогда не восстанавливает память, используемую интернированными строками, поэтому они могут замедлять работу программы, занимая слишком много места в памяти. В этой книге интернированные строки не рассматриваются. В версии v3 функция `intern` не является встроенной: вместо этого она хранится в более подходящем месте — в модуле `sys`
isinstance	`isinstance(`*`obj, cls`*`)` Возвращает значение `True`, если *`obj`* — экземпляр класса *`cls`* (или любого его подкласса); в противном случае возвращается значение `False`. Аргумент *`cls`* может быть кортежем, элементы которого являются классами. В этом случае вызов `isinstance` возвращает значение `True`, если *`obj`* — экземпляр любого из элементов *`cls`*; иначе возвращается значение `False`. См. также раздел "Абстрактные базовые классы" в главе 4
issubclass	`issubclass(`*`cls1, cls2`*`)` Возвращает значение `True`, если *`cls1`* — прямой или косвенный подкласс *`cls2`*; в противном случае возвращается значение `False`. Аргументы *`cls1`* и *`cls2`* должны быть классами. Аргумент *`cls2`* также может быть кортежем, элементы которого являются классами. В этом случае функция `issubclass` возвращает значение `True`, если *`cls1`* — прямой или косвенный подкласс любого из элементов аргумента *`cls2`*; иначе возвращается значение `False`. Для любого класса *`C`* вызов `issubclass(`*`C, C`*`)` возвращает значение `True`

iter	`iter(`*`obj`*`) iter(`*`func, sentinel`*`)`
	
	Создает и возвращает *итератор* — объект, который можно повторно передавать встроенной функции `next` для получения по одному элементу за раз (см. раздел "Итераторы" главы 3). Обычно вызов этой функции с одним аргументом `iter(`*`obj`*`)` возвращает результат вызова *`obj`*`.__iter__()`. Если *`obj`* — последовательность, не реализующая специальный метод `__iter__`, то функция `iter(`*`obj`*`)` эквивалентна следующему генератору.
	
	`def iter_sequence(obj):`
	`  i = 0`

Функция	Описание

```
    while True:
        try: yield obj[i]
        except IndexError: raise StopIteration
        i += 1
```

См. также раздел "Последовательности" в главе 3 и описание метода `__iter__` в табл. 4.2.

Если данная функция вызывается с двумя аргументами, то первый аргумент должен быть вызываемым объектом, не принимающим аргументы, а функция `iter(`*`func`*`, `*`sentinel`*`)` эквивалентна следующему генератору.

```
def iter_sentinel(func, sentinel):
    while True:
        item = func()
        if item == sentinel: raise StopIteration
        yield item
```

Не вызывайте функцию `iter` в предложениях `for`

Как обсуждалось в разделе "Инструкция `for`" в главе 3, инструкция `for` *`x`* `in` *`obj`* полностью эквивалентна инструкции `for` *`x`* `in iter(`*`obj`*`)`. В связи с этим не следует вызывать функцию `iter` в подобных инструкциях `for`: это замедляет работу кода, ухудшает его читаемость и потому не соответствует духу Python.

Функция `iter` *идемпотентна*. Другими словами, если *`x`* — итератор, то вызов `iter(`*`x`*`)` возвращает *`x`*, при условии, что класс объекта *`x`* предоставляет метод `__iter__`, тело которого состоит всего лишь из одной инструкции `return self`

`len` `len(`*`container`*`)`

Возвращает количество элементов в контейнере, который может быть последовательностью, отображением или множеством. См. также описание метода `__len__` в разделе "Методы контейнеров" главы 4

`locals` `locals()`

Возвращает словарь, представляющий текущее локальное пространство имен. Обрабатывайте возвращенный словарь как доступный только для чтения. Попытка изменить его чревата изменением значений локальных переменных и может привести к возбуждению исключения. См. также описание функций `globals` и `vars` в данной таблице

`map` `map(`*`func`*`, `*`seq`*`, *`seqs`*`)`

Функция `map` вызывает объект `func` для каждого элемента итерируемого объекта *`seq`* и возвращает последовательность результатов. Если `map` вызывается с *`n+1`* аргументом, то первый из них, *`func`*, может быть любым вызываемым объектом, который принимает *`n`* аргументов. Все остальные аргументы, передаваемые `map`, должны быть итерируемыми объектами.

Функция	Описание
	Функция `map` повторно вызывает объект *`func`* с *`n`* аргументами (каждый из которых соответствует одному элементу из каждого итерируемого объекта). В версии v3 функция `map` возвращает итератор, выдающий результаты. Например, вызов `map(`*`func, seq`*`)` действует аналогично выражению-генератору `(`*`func(item) for item in seq`*`)`. Если итерируемые аргументы, передаваемые функции `map`, имеют разную длину, то в версии v3 `map` действует так, как если бы наиболее длинные из них были усечены. В версии v2 функция `map` возвращает список результатов. Например, вызов `map(`*`func, seq`*`)` действует подобно генератору списка `[`*`func(item) for item in seq`*`]`. Если итерируемые аргументы, передаваемые функции `map`, имеют разную длину, то в версии v2 `map` действует так, как если бы наиболее короткие из них были дополнены значениями `None`. Кроме того, только в версии v2 аргумент *`func`* может быть равен `None`: в этом случае каждый результат представляет собой кортеж с *`n`* элементами (по одному элементу из каждого итерируемого объекта)
`max`	`max(`*`s, *args, key`*`=None[, `*`default`*`=...])` Возвращает элемент, являющийся наибольшим в единственном позиционном аргументе *`s`* (объект *`s`* должен быть итерируемым) или наибольшим среди нескольких аргументов. Функция `max` — одна из встроенных функций, которые можно вызывать с именованными аргументами: в частности, ей можно передать аргумент `key=` с той же самой семантикой, которая обсуждалась в разделе "Сортировка списка" главы 3. Только в версии v3 допускается передача дополнительного аргумента `default=` — значение, которое должно быть возвращено, если единственный позиционный аргумент — пустой. Если аргумент `default` не передан, а аргумент *`s`* — пустой, то функция `max` возбуждает исключение `ValueError`
`min`	`min(`*`s, *args, key`*`=None[, `*`default`*`=...])` Возвращает элемент, являющийся наименьшим в единственном позиционном аргументе *`s`* (объект *`s`* должен быть итерируемым) или наименьшим среди нескольких аргументов. Функция `min` — одна из встроенных функций, которые можно вызывать с именованными аргументами. Ы частности, ей можно передать аргумент `key=` с той же самой семантикой, которая обсуждалась в разделе "Сортировка списка" главы 3. Только в версии v3 допускается передача дополнительного аргумента `default=` — значение, которое должно быть возвращено, если единственный позиционный аргумент — пустой. Если аргумент `default` не передан, а аргумент *`s`* — пустой, то функция `min` возбуждает исключение `ValueError`
`next`	`next(`*`it[, default`*`])` Возвращает следующий элемент из итератора *`it`*, который продвигается до следующего элемента. Если все элементы исчерпаны, то функция `next` возвращает значение *`default`* или, если аргумент *`default`* не был передан, возбуждает исключение `StopIteration`

Функция	Описание
oct	`oct(x)` Преобразует целое число *x* в восьмеричное строковое представление. См. также описание метода `__oct__` в табл. 4.4
open	`open(filename, mode='r', bufsize=-1)` Открывает или создает файл и возвращает новый объект файла. В версии v3 функция `open` принимает много необязательных параметров, а в версии v2 с помощью инструкции `from io import open` можно перекрыть встроенную функцию `open` функцией, очень похожей на одноименную функцию версии v3. См. раздел "Модуль `io`" в главе 10
ord	`ord(ch)` В версии v2 возвращает целочисленное значение кода одиночного символа *ch* в кодировке ASCII/ISO, которое может находиться в диапазоне от `0` до `255` (включая граничные значения). В версии v2, если типом *ch* является `unicode` (и всегда в версии v3), функция `ord` возвращает целочисленное значение кода символа *ch*, которое может находиться в диапазоне от `0` до `sys.maxunicode` (включая граничные значения). См. также описание функций `chr` и `unichr` в данной таблице
pow	`pow(x, y[, z])` Если аргумент *z* имеется, то функция `pow(x, y, z)` возвращает значение выражения `x**y%z`. Если *z* отсутствует, то функция `pow(x, y)` возвращает значение выражения `x**y`. См. также описание метода `__pow__` в табл. 4.4
print	`print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)` В версии v3 форматирует с использованием типа `str` и направляет в поток *file* каждое из значений *value*, разделенных символами *sep*, со вставкой символа *end* после всех значений (а затем выталкивает оставшиеся данные из буфера, если аргумент *flush* имеет значение `True`). В версии v2 `print` — не функция, а инструкция, кроме тех случаев, когда в начале модуля находится инструкция `from __future__ import print_function`, что настоятельно рекомендуется делать (и предполагается в каждом из примеров, приведенных в книге), и в этом случае `print` работает так же, как в версии v3
range	`range([start,] stop[, step=1])` В версии v2 возвращает список целых чисел, образующих арифметическую прогрессию: `[start, start+step, start+2*step, ...]`. Значение аргумента *start*, если оно отсутствует, по умолчанию равно `0`. Значение аргумента *start*, если оно отсутствует, по умолчанию равно `1`. Если аргумент *step* задан равным `0`, функция `range` возбуждает `ValueError`. Если значение *step* больше `0`, то последним элементом является наибольшее значение `start+i*step`, строго меньшее значения *stop*. Если значение *step* меньше `0`, то последним элементом является наименьшее значение `start+i*step`, строго большее значения *stop*. Если значение *start* равно

Функция	Описание
	значению *stop* или превышает его, а значение *step* больше 0, или если значение *start* равно значению *stop* или меньше его, а значение *step* меньше 0, то результат представляет собой пустой список. В противном случае первым элементом результирующего списка всегда является *start*. См. также описание функции xrange в данной таблице. В версии v3 функция range возвращает встроенный тип, обеспечивающий эффективное компактное представление эквивалентного, *доступного только для чтения* списка целых чисел, образующих арифметическую прогрессию. В большинстве практических применений эта функция эквивалентна функциям range и xrange версии v2, но более эффективна. Если требуется именно список элементов арифметической прогрессии, то в версии v3 для этого следует выполнить вызов list(range(...))
raw_input	raw_input(*prompt*='') Только в версии v2: записывает строку-приглашение *prompt* в стандартное устройство вывода, читает строку из стандартного устройства ввода и возвращает ее (без завершающего символа \n). В случае считывания признака конца файла функция raw_input возбуждает исключение EOFError. См. также описание функции input в данной таблице. В версии v3 эта функция называется input

reduce	reduce(*func*, *seq*[,*init*])
	
	(В версии v3 функция reduce не является встроенной: она хранится в модуле functools.) Функция reduce поочередно применяет объект *func* к элементам последовательности *seq* в направлении слева направо, сводя ее к единственному значению. Объект *func* должен быть вызываемым объектом с двумя аргументами. Функция reduce вызывает *func* для первого из двух элементов *seq*, затем для результата первого вызова и третьего элемента и т.д. Функция reduce возвращает результат последнего из таких вызовов. Если задан аргумент init, то он используется до того, как будет использован первый элемент *seq*, если таковой имеется. В случае отсутствия *init* последовательность *seq* не должна быть пустой. Если *init* отсутствует, а *seq* имеет только один элемент, то функция reduce возвращает элемент *seq*[0]. Если *init* имеется, а последовательность *seq* — пустая, функция reduce возвращает *init*. Таким образом, функция reduce в целом эквивалентна следующему коду:
	
	`def reduce_equivalent(func,seq,init=None):`
	`  seq = iter(seq)`
	`  if init is None: init = next(seq)`
	`  for item in seq: init = func(init,item)`
	`  return init`
	
	В качестве примера можно привести вычисление произведения элементов последовательности:
	
	`theprod = reduce(operator.mul, seq, 1)`

Функция	Описание
reload	`reload(module)` Перезагружает и заново инициализирует объект модуля *module* и возвращает *module*. В версии v3 функция `reload` не является встроенной: в ранних версиях Python 3 она хранится в модуле `imp`, в текущих версиях (3.4, 3.5 и т.д.) — в модуле `importlib`
repr	`repr(obj)` Возвращает полное описательное строковое представление объекта *obj*. Если это возможно, функция `repr` возвращает строку, которую можно передать функции `eval` для создания нового объекта с тем же значением, что и *obj*. См. также описание типа `str` в табл. 7.1 и метода `__repr__` в табл. 4.1
reversed	`reversed(seq)` Возвращает новый объект итератора, выдающий элементы последовательности *seq* (которая должна быть именно последовательностью, а не просто итерируемым объектом) в обратном порядке
round	`round(x, n=0)` Возвращает число с плавающей точкой, равное значению *x*, округленному до указанного количества цифр (*n*) после десятичной точки (т.е. равное ближайшему к *x* числу, кратному `10**-n`). Если имеются два таких кратных числа, равноудаленных от *x*, то в версии v2 функция `round` возвращает то из них, которое находится дальше от 0; в версии v3 функция `round` возвращает четное кратное. Поскольку в современных компьютерах числа с плавающей точкой представляются не в десятичном формате, а в двоичном, то в большинстве случаев возвращаемые функцией `round` результаты не являются точными, о чем можно подробнее прочитать в онлайновом руководстве (`https://docs.python.org/3/tutorial/floatingpoint.html`). См. также раздел "Модуль `decimal`" в главе 15
setattr	`setattr(obj, name, value)` Связывает атрибут `name` объекта *obj* со значением *value*. Вызов `setattr(obj, 'ident', val)` аналогичен вызову *obj.ident=val*. См. описание встроенной функции `getattr` в данной таблице, а также разделы "Атрибуты и элементы объектов" в главе 3 и "Связанные и несвязанные методы" в главе 4

sorted	`sorted(seq, cmp=None, key=None, reverse=False)`
	Возвращает список с теми же элементами, что и в итерируемом объекте *seq*, расположенными в отсортированном порядке. Эквивалентна следующему коду:
	`def sorted(seq,cmp=None,key=None,reverse=False):`
	`  result = list(seq)`
	`  result.sort(cmp,key,reverse)`
	`  return result`
	Аргумент *cmp* предусмотрен только в версии v2, но не в версии v3; см. описание метода `cmp_to_key` в табл. 7.4. О назначении аргументов см. в разделе

Функция	Описание
	"Сортировка списка" главы 3. Функция `sorted` — одна из встроенных функций, допускающих именованные аргументы, в частности, для того, чтобы можно было передавать необязательные аргументы *key* и/или *reverse*
sum	`sum(`*seq*`, `*start*`=0)` Возвращает сумму элементов итерируемого объекта *seq* (которые должны быть числами и, в частности, не могут быть строками) плюс начальное значение *start*. Чтобы "просуммировать" (конкатенировать) итерируемый объект, состоящий из строк, используйте метод `''.join(`*iterofstrs*`)`, описанный в табл 8.1, а также обратитесь к разделу "Сборка строк из отдельных элементов" в главе 16
unichr	`unichr(`*code*`)` Только в версии v2: возвращает строку Unicode, единственный символ которой соответствует аргументу *code*, где *code* — целое число, значения которого могут находиться в пределах от `0` до `sys.maxunicode` (включая граничные значения). См. также описание функций `str` и `ord` в табл. 7.1. В версии v3 для этих целей следует использовать функцию `chr`
vars	`vars([`*obj*`])` Будучи вызванной без аргументов, функция `vars` возвращает словарь, содержащий все связанные переменные в текущей области видимости (подобно функции `locals`, описанной в данной таблице). Обращайтесь с этим словарем как с доступным только для чтения. Вызов `vars(`*obj*`)` возвращает словарь, содержащий все атрибуты, которые в данный момент связаны в объекте *obj*. См. также описание функции `dir` в данной таблице. Этот словарь может разрешать внесение изменений, в зависимости от типа *obj*
xrange	`xrange([`*start*`,]`*stop*`[,`*step*`=1])` Только в версии v2: возвращает итерируемый объект, который содержит целые числа, образующие арифметическую прогрессию, во всех остальных отношениях аналогичный объекту, возвращаемому функцией `range`. В версии v3 эту роль играет функция `range`. См. также описание функции `range` в этой таблице
zip	`zip(`*seq*`,*`*seqs*`)` В версии v2 возвращает список кортежей, в котором *n*-й кортеж содержит *n*-й элемент каждой из последовательностей, заданных аргументами. Функция `zip` должна вызываться по крайней мере с одним аргументом, причем все аргументы должны быть итерируемыми объектами. Если итерируемые объекты имеют разную длину, то функция `zip` возвращает список, длина которого определяется длиной самого короткого итерируемого объекта, тогда как остающиеся хвостовые элементы в других итерируемых объектах игнорируются. См. также описания функции `map` в данной таблице и функции `izip_longest` в табл. 7.5. В версии v3 функция `zip` возвращает итератор, а не список, и потому эквивалентна выражению-генератору, а не генератору списка

Модуль sys

Атрибуты модуля sys связаны с данными и функциями, которые предоставляют информацию о состоянии интерпретатора Python или непосредственно влияют на интерпретатор. В табл. 7.3 приведены наиболее часто используемые атрибуты sys, перечисленные в алфавитном порядке. Большинство атрибутов модуля sys, которые не включены в данную таблицу, имеют специальное назначение и предназначены для использования в отладчиках, профилировщиках и интегрированных средах разработки; более подробную информацию о них можно получить в онлайн-документации (https://docs.python.org/3/library/sys.html). Для получения платформозависимой информации лучше всего воспользоваться модулем platform (https://docs.python.org/3/library/platform.html?highlight=platform%20module#module-platform), который в этой книге не рассматривается.

Таблица 7.3. Атрибуты модуля sys

Атрибут	Описание
argv	Список аргументов командной строки, переданных основному сценарию. argv[0] — имя или полный путь к файлу основного сценария или же '-c', если в командной строке использована опция -c. Полезный пример применения атрибута sys.argv приведен в разделе "Модуль argparse"
byteorder	Имеет значение 'little' в случае платформ, на которых для представления информации используется порядок байтов *от младшего к старшему* (little-endian), и 'big' в случае платформ, на которых используется порядок байтов *от старшего к младшему* (big-endian). Более подробно об этом можно прочитать в Википедии (https://ru.wikipedia.org/wiki/Порядок_байтов)
builtin_module_names	Кортеж строк, которые представляют имена всех модулей, скомпилированных в данном интерпретаторе

displayhook	displayhook(*value*)
	Python вызывает функцию displayhook в интерактивных сеансах, передавая ей результат вычисления каждого введенного выражения. По умолчанию функция displayhook не выполняет никаких действий, если *value* равно None; в противном случае она сохраняет значение *value* (во встроенной переменной _) и отображает его посредством функции repr.
	`def _default_sys_displayhook(value):`
	`  if value is not None:`
	`  __builtins__._ = value`
	`  print(repr(value))`
	Можно повторно связать атрибут sys.displayhook для изменения описанного интерактивного поведения. Исходное значение остается доступным через атрибут sys.__displayhook__

Атрибут	Описание
`dont_write_bytecode`	Если значение этого атрибута `True`, Python не записывает файл байт-кода (с расширением *.pyc* или, в версии v2, *.pyo*) на диск в ходе импортирования файла с исходным кодом (имеющего расширение *.py*). В частности, эта возможность оказывается удобной при импорте из файловой системы, доступной только для чтения
`excepthook`	`excepthook(type, value, traceback)` В тех случаях, когда исключение не перехватывается обработчиками и распространяется вверх вдоль всего стека вызовов, Python вызывает функцию `excepthook`, передавая ей класс исключения, объект и трассировочную информацию (см. раздел "Распространение исключений" в главе 5). Заданная по умолчанию функция `excepthook` отображает сообщение об ошибке и трассировочную информацию. Заданный по умолчанию способ отображения и/или протоколирования информации о не перехваченных исключениях можно изменить, повторно связав атрибут `sys.excepthook`. Копия ссылки на встроенную функцию, заданную по умолчанию, хранится в атрибуте `sys.__excepthook__`
`exc_info`	`exc_info()` Если текущий поток обрабатывает исключение, функция `exc_info` возвращает кортеж, содержащий три элемента: класс, объект и объект трассировки исключения. Если текущий поток не обрабатывает исключение, то функция `exc_info` возвращает кортеж (`None`, `None`, `None`). Относительно отображения трассировочной информации обратитесь к разделу "Модуль `traceback`" главы 16.

Хранение ссылки на объект трассировки может препятствовать удалению ненужных объектов механизмом сборки мусора

Объект трассировки косвенно сохраняет ссылки на все переменные стека вызовов. Если вы сохраняете ссылку на объект трассировки (например, косвенно, посредством связывания переменной с кортежем, возвращаемым функцией `exc_info`), то Python должен хранить в памяти данные, которые иначе были бы удалены механизмом сборки мусора. Всегда убеждайтесь в том, что любые связывания объекта трассировки являются лишь кратковременными, используя, например, инструкцию `try/finally` (обсуждается в разделе "Инструкция `try/finally`" главы 5)

Атрибут	Описание
exit	`exit(`*`arg`*`=0)` Возбуждает исключение `SystemExit`, которое обычно прекращает выполнение программы после того, как с помощью обработчиков, установленных с помощью инструкций `try/finally`, `with` и средств модуля `atexit`, будут выполнены завершающие операции по очистке ресурсов. Если аргумент `arg` является целым числом, Python использует его в качестве кода завершения работы программы: `0` — успешное завершение, тогда как все остальные значения указывают на преждевременное прекращение выполнения программы по тем или иным причинам. Для большинства платформ код завершения имеет значения в диапазоне от `0` до `127`. Если `arg` не является целым числом, то Python выводит значение `arg` в стандартный поток вывода ошибок `sys.stderr` и устанавливает код завершения равным `1` (типовой код "неудачного завершения")
float_info	Доступный только для чтения объект, в атрибутах которого хранится подробная низкоуровневая информация, касающаяся реализации типа `float` в данном интерпретаторе Python. Для получения более подробных сведений обратитесь к онлайн-документации (`https://docs.python.org/3/library/sys.html#sys.float_info`)
getrefcount	`getrefcount(`*`object`*`)` Возвращает счетчик ссылок объекта *`object`*. Счетчики ссылок рассматриваются в разделе "Сборка мусора" главы 13
getrecursionlimit	`getrecursionlimit()` Возвращает текущий предел глубины стека вызовов Python. См. также раздел "Рекурсия" в главе 3 и описание функции `setrecursionlimit` в данной таблице
getsizeof	`getsizeof(`*`obj`*`, [`*`default`*`])` Возвращает размер (в байтах) объекта *`obj`* (не считая элементов и атрибутов, на которые может ссылаться данный объект) или значение *`default`*, если объект *`obj`* не предоставляет способа для определения его размера (в последнем случае, если аргумент *`default`* отсутствует, возбуждается исключение `TypeError`)
maxint	(Только в версии v2.) Наибольшее возможное значение типа `int` в данной версии Python (равное по крайней мере `2**31-1`, т.е. `2147483647`). Отрицательные значения могут простираться до значения `-maxint-1` в дополнительном представлении. В версии v3 размер типа `int` не имеет ограничений (подобно типу `long` в версии v2), поэтому понятие "наибольшее возможное целое значение" в этой версии отсутствует
maxsize	Наибольшее возможное количество байтов в объекте в данной версии Python (равное по крайней мере `2**31-1`, т.е. `2147483647`)

Атрибут	Описание
`maxunicode`	Наибольшее значение кодовой точки для символа Unicode в данной версии (равное по крайней мере `2**16-1`, т.е. `65535`). В версии v3 всегда равно `1114111` (`0x10FFFF`)
`modules`	Словарь, элементами которого являются имена и объекты всех загруженных модулей. Более подробная информация о словаре `sys.modules` приведена в разделе "Загрузка модуля" главы 6
`path`	Список строк, определяющих каталоги и ZIP-файлы, в которых Python выполняет поиск загружаемых модулей. Более подробная информация о списке `sys.path` приведена в разделе "Поиск модуля в файловой системе" главы 6
`platform`	Строка, содержащая название платформы, на которой выполняется программа. Типичными значениями являются краткие названия операционных систем, такие как `'darwin'`, `'linux2'` и `'win32'`. В частности, чтобы обеспечить переносимость кода между версиями Linux и версиями v2/v3, выполняйте проверку того, что текущей операционной системой является Linux, с помощью выражения `sys.platform.startswith('linux')`
`ps1`, `ps2`	`ps1` и `ps2` определяют строки основного и вспомогательного приглашений к вводу, которыми первоначально являются строки `'>>> '` и `'... '` соответственно. Эти атрибуты существуют лишь в интерактивных сеансах работы с Python. Если вы свяжете любой из этих атрибутов с объектом *x*, не являющимся строкой, то Python будет вызывать функцию `str(x)` для данного объекта всякий раз, когда требуется вывести приглашение к вводу. Это обеспечивает возможность динамического формирования приглашений: напишите класс, который определяет метод `__str__`, свяжите экземпляр этого класса с атрибутом `sys.ps1` и/или `sys.ps2`. Например, для вывода нумерованных приглашений можно использовать следующий код.

```
>>> import sys
>>> class Ps1(object):
... def __init__(self):
... self.p = 0
... def __str__(self):
... self.p += 1
... return '[{}]>>> '.format(self.p)
...
>>> class Ps2(object):
... def __str__(self):
... return '[{}]... '.format(sys.ps1.p)
...
>>> sys.ps1 = Ps1(); sys.ps2 = Ps2()
[1]>>> (2 +
```

Атрибут	Описание

	`[1]... 2)`
	`4`
	`[2]>>>`

setrecursionlimit	`setrecursionlimit(`*`limit`*`)` Устанавливает предельное значение глубины стека вызовов Python (по умолчанию — `1000`). Этот лимит предотвращает крах Python из-за слишком большого количества рекурсивных вызовов. В случае программ с глубокими рекурсиями может потребоваться повышение данного лимита, но большинство платформ не в состоянии поддерживать слишком большие значения глубины стека вызовов. Более полезной областью применения функции является *снижение* этого лимита в проверочных целях во время тестирования и отладки программы для организации корректного снижения возможностей программы вместо возбуждения исключения `RuntimeError` в ситуациях, когда количество рекурсивных вызовов способно переполнить стек вызовов. См. также раздел "Рекурсия" в главе 3 и описание функции `getrecursionlimit` в данной таблице
stdin, stdout, stderr	`stdin`, `stdout` и `stderr` являются предопределенными файловыми объектами, которые соответствуют стандартным потокам ввода, вывода и ошибок Python. Повторно связав атрибуты `stdout` и `stderr` с файловыми объектами, открытыми для записи (объектами, которые предоставляют метод `write`, принимающий строковый аргумент), можно перенаправить вывод полезной информации и сообщений об ошибках в другое место. Повторно связав атрибут `stdin` с файловым объектом, открытым для чтения (объектом, который предоставляет метод `readline`, возвращающий строку), можно назначить другой источник ввода, из которого будут выполнять чтение встроенные функции `raw_input` (только версия v2) и `input`. Первоначальные значения этих атрибутов остаются доступными в атрибутах `__stdin__`, `__stdout__` и `__stderr__`. Файловые объекты рассматриваются в разделе "Модуль `io`" главы 10
tracebacklimit	Максимальное количество уровней стека вызовов, отображаемых для необработанных исключений. По умолчанию этот атрибут не установлен (т.е. ограничений не существует). При значениях атрибута `sys.tracebacklimit<=0` Python выводит лишь тип исключения и значение без трассировочной информации
version	Строка, описывающая версию Python, номер сборки и дату ее создания, а также версию использованного компилятора. Значением `sys.version[:3]` является `'2.7'` для Python 2.7, `'3.5'` для Python 3.5 и т.д.

Модуль copy

Как обсуждалось в разделе "Операции присваивания" главы 3, при выполнении инструкции присваивания Python не копирует присваиваемый объект, указанный справа от знака равенства. Вместо этого инструкция присваивания создает ссылку на объект. Если вы хотите создать копию *x*, можете запросить создание копии самим объектом *x* или запросить тип объекта *x* создать новый экземпляр объекта, являющийся копией *x*. Если *x* — список, то копию *x* возвращает вызов `list(x)`, равно как и выражение `x[:]`. Если *x* — словарь, то копию *x* возвращают вызовы `dict(x)` и `x.copy()`. Если *x* — множество, то копию *x* возвращают вызовы `set(x)` и `x.copy()`. Мы считаем, что во всех этих случаях лучше всего последовательно использовать хорошо читаемую идиому вызова типа, но среди сообщества пользователей Python единого мнения по этому поводу не существует.

Модуль `copy` предоставляет функцию `copy`, способную создавать и возвращать копии многих объектов. Обычные копии, например, создаваемые с помощью вызовов `list(x)` для списка *x* и `copy.copy(x)`, называются *мелкими* (поверхностными): если *x* содержит ссылки на другие объекты (элементы или атрибуты), то обычная (мелкая) копия *x* содержит отдельные ссылки на те же самые объекты. Однако иногда вам может понадобиться *глубокая* копия, использующая рекурсивное копирование вложенных объектов. К счастью, необходимость в этом возникает лишь в редких случаях, поскольку глубокая копия занимает больше места в памяти и требует больше времени для своего создания. С целью создания глубоких копий модуль `copy` предоставляет функцию `deepcopy`.

copy `copy(x)`

Создает и возвращает мелкую копию *x* для объектов *x* многих типов (однако копирование нескольких типов, таких как модули, классы, файлы, фреймы и другие внутренние типы, не поддерживается). Если *x* — неизменяемый объект, то вызов `copy.copy(x)` может вернуть сам объект *x* в качестве оптимизации. Класс может изменить способ копирования своих экземпляров посредством вызовов `copy.copy`, определив специальный метод `__copy__(self)`, который возвращает новый объект — мелкую копию `self`.

deepcopy `deepcopy(x, [memo])`

Создает глубокую копию *x* и возвращает ее. Глубокое копирование предполагает рекурсивный обход направленного (не обязательно ациклического) графа ссылок (`https://ru.wikipedia.org/wiki/Глоссарий_теории_графов`). Воспроизведение точной формы графа требует соблюдения некоторых мер предосторожности: если в процессе обхода графа некоторые ссылки на один и тот же объект встречаются более одного раза, то отдельные ссылки не должны создаваться. Вместо этого должны использоваться ссылки на тот же самый копируемый объект. Рассмотрим следующий простой пример.

```
sublist = [1,2]
original = [sublist, sublist]
thecopy = copy.deepcopy(original)
```

Значением выражения `original[0] is original[1]` является `True` (т.е. оба элемента списка `original` ссылаются на один и тот же объект). Это важное свойство списка `original`, и любой объект, предполагаемый в качестве "копии", должен следовать данному правилу. Семантика функции `copy.deepcopy` гарантирует, что значением выражения `thecopy[0] isthecopy[1]` также является `True`: графы ссылок списков `original` и `thecopy` имеют одну и ту же форму. Устранение повторного копирования имеет важный положительный побочный эффект: оно позволяет избежать бесконечных циклов, которые в противном случае могли бы возникать, если в графе ссылок имеются циклы.

Вызов `copy.deepcopy` принимает второй, необязательный аргумент *memo* — словарь, который сопоставляет идентификаторы уже скопированных объектов с новыми объектами, являющимися их копиями. Аргумент *memo* передается всеми рекурсивными вызовами функции `deepcopy` самой себя; вы также можете явно передавать его (обычно в виде первоначально пустого словаря), если вам нужно поддерживать карту соответствия между идентичностями оригиналов и копий.

Класс может изменить способ копирования своих экземпляров посредством вызовов `copy.deepcopy`, определив специальный метод `__deepcopy__(self, memo)`, который возвращает новый объект — глубокую копию `self`. Если методу `__deepcopy__` требуется создать глубокую копию некоторого вложенного объекта `subobject`, он должен сделать это путем вызова `copy.deepcopy(subobject, memo)`. Если класс не имеет специального метода `__deepcopy__`, то вызов `copy.deepcopy` для экземпляра этого класса пытается вызвать также специальные методы `__getinitargs__`, `__getnewargs__`, `__getstate__` и `__setstate__`, рассмотренные в разделе "Сериализация и десериализация экземпляров с помощью модуля `pickle`" главы 11.

Модуль `collections`

Модуль `collections` предоставляет полезные типы, среди которых есть как коллекции (т.е. контейнеры), так и абстрактные базовые классы (см. раздел "Абстрактные базовые классы" в главе 4). Начиная с Python 3.4 абстрактные базовые классы содержатся в модуле `collections.abc` но для обеспечения обратной совместимости они все еще остаются непосредственно доступными в самом модуле `collections`. Можно, однако, ожидать, что в одном из будущих выпусков версии v3 этой возможности уже не будет.

Класс `ChainMap` (только в версии v3)

Класс `ChainMap` "связывает в цепочку" несколько отображений. Пусть *c* — экземпляр класса `ChainMap`, тогда выражение `c[key]` возвращает значение первого из отображений, которое имеет ключ *key*, в то время как любые изменения *c* воздействуют лишь на самое первое отображение в *c*. В версии v2 это поведение можно аппроксимировать следующим кодом.

```
class ChainMap(collections.MutableMapping):
    def __init__(self, *maps):
        self.maps = list(maps)
        self._keys = set()
        for m in self.maps: self._keys.update(m)
    def __len__(self): return len(self._keys)
    def __iter__(self): return iter(self._keys)
    def __getitem__(self, key):
        if key not in self._keys: raise KeyError(key)
        for m in self.maps:
            try: return m[key]
            except KeyError: pass
    def __setitem__(self, key, value)
        self.maps[0][key] = value
        self._keys.add(key)
    def __delitem__(self, key)
        del self.maps[0][key]
        self._keys = set()
        for m in self.maps: self._keys.update(m)
```

Для повышения эффективности класса можно определить и другие методы, но приведенный выше их набор является тем минимумом, которого требует класс `MutableMapping`. Стабильная на производственном уровне ретроподдержка `ChainMap` для версии v2 (и ранних версий Python 3) доступна в репозитории PyPI (`https://pypi.python.org/pypi/chainmap/`), и вы можете установить данный модуль подобно любому другому модулю PyPI — например, выполнив команду **`pip install chainmap`**.

Для получения более подробной информации о классе `ChainMap` и ознакомления с "рецептами" его применения обратитесь к онлайн-документации Python (`https://docs.python.org/3/library/collections.html?highlight=collections#collections.ChainMap`).

Класс `Counter`

`Counter` — это подкласс `dict` с целочисленными значениями, предназначенными для подсчета количества вхождений каждого ключа (хотя допустимы также значения `<=0`). В общих чертах он соответствует типам, которые в других языках программирования называются *мультимножествами* (multi-set). Обычно экземпляры `Counter` создаются на основе итерируемых объектов с хешируемыми элементами: `c = collections.Counter(iterable)`. После этого вы сможете определять количество вхождений элемента итерируемого объекта, используя его в качестве индекса. При обращении к `c` по отсутствующему ключу возвращается `0` (для удаления записи в `c` используйте инструкцию `del c[entry]`; инструкция `c[entry] = 0` оставляет запись в `c`, просто соответствующее значение становится равным `0`).

Экземпляр *c* поддерживает все методы dict. В частности, вызов *c*.update(*otheriterable*) обновляет все счетчики, инкрементируя их в соответствии с количеством вхождений элементов в объекте *otheriterable*. Поэтому, например, после выполнения следующего кода выражение *c*['o'] возвращает значение 4, а каждое из выражений *c*['f'] и *c*['m'] — значение 1.

```
c = collections.Counter('moo')
c.update('foo')
```

В дополнение к методам dict экземпляр *c* поддерживает еще три метода, которые описаны ниже.

elements *c*.elements()

Возвращает итератор по элементам, повторяющий ключи из *c*, для которых *c*[*key*]>0, причем количество повторений ключа определяется значением его счетчика.

most_common *c*.most_common([*n*])

Возвращает список пар для *n* ключей *c*, имеющих наибольшие значения счетчика (или все ключи, если *n* опущено), в порядке уменьшения значений счетчика (с произвольным разрешением "связей" между ключами с одинаковыми значениями счетчика). Каждая пара имеет вид (*k*, *c*[*k*]), где *k* — один из *n* наиболее часто встречающихся ключей *c*.

subtract *c*.subtract(*iterable*)

Подобен методу *c*.update (*iterable*), выполняемому "в обратном направлении", т.е. выполняет вычитание значений счетчиков, а не их сложение. Результирующие значения счетчиков *c* могут быть <=0.

Для получения более подробной информации о классе Counter и ознакомления с "рецептами" его применения обратитесь к онлайн-документации Python (https://docs.python.org/3/library/collections.html?highlight=collections#collections.Counter).

Класс OrderedDict

OrderedDict — это подкласс dict, запоминающий порядок, в котором первоначально вставлялись ключи (присваивание нового значения существующему ключу не изменяет этот порядок, но при удалении ключа и повторной его вставке порядок изменяется). Пусть *o* — экземпляр класса OrderedDict, тогда при итерировании по элементам *o* ключи возвращаются в порядке их вставки (от самого старого к самому новому), а вызов o.popitem() удаляет и возвращает последний вставленный ключ. Проверки на равенство двух экземпляров OrderedDict чувствительны к порядку; проверки на равенство экземпляра OrderedDict и экземпляра dict или другого типа отображения нечувствительны к порядку. Для получения более подробной информации о классе OrderedDict и ознакомления с "рецептами" его применения

обратитесь к онлайн-документации Python (https://docs.python.org/3/library/collections.html?highlight=collections#ordereddict-objects).

Класс defaultdict

Класс defaultdict расширяет класс dict и добавляет атрибут экземпляра default_factory. Если в экземпляре *d* класса defaultdict значение атрибута d.default_factory равно None, то *d* ведет себя как словарь. В противном случае атрибут d.default_factory должен быть вызываемым объектом без аргументов, и тогда *d* ведет себя как словарь, за исключением случаев, когда вы пытаетесь получить доступ к *d* по ключу *k*, который отсутствует в *d*. В этом конкретном случае при обращении к *d*[*k*] выполняется вызов *d*.default_factory(), и возвращаемый результат присваивается *d*[*k*] в качестве значения. Другими словами, тип defaultdict ведет себя в значительной мере подобно классу, представленному следующим кодом.

```
class defaultdict(dict):
    def __init__(self, default_factory, *a, **k):
        dict.__init__(self, *a, **k)
        self.default_factory = default_factory
    def __getitem__(self, key):
        if key not in self and self.default_factory is not None:
        self[key] = self.default_factory()
    return dict.__getitem__(self, key)
```

Как следует из приведенного эквивалентного кода, инстанциализация класса defaultdict требует передачи ему дополнительного аргумента (предшествующего любым другим возможным позиционным и/или именованным аргументам, передаваемым простому словарю типа dict). Этот дополнительный аргумент становится начальным значением default_factory; вы сможете обращаться к default_factory и повторно связывать данный атрибут.

Общее поведение defaultdict в основном совпадает с поведением, следующим из приведенного выше эквивалентного кода (за исключением методов str и repr, возвращающих строки, отличные от тех, которые были бы возвращены для словаря типа dict). Такие методы, как get и pop, никаких изменений не испытывают. Все поведение, связанное с ключами (метод keys, итерирование, проверка принадлежности посредством оператора in и т.п.), в точности соответствует тем ключам, которые в данный момент содержатся в контейнере (независимо от того, поместили ли вы их явно или неявно посредством индексирования, вызвавшего default_factory).

Типичным применением defaultdict является установка list в качестве default_factory для создания отображения, ассоциирующего ключи со списками значений.

```
def make_multi_dict(items):
    d = collections.defaultdict(list)
```

```
    for key, value in items: d[key].append(value)
return d
```

Будучи вызванной с любым итерируемым объектом, элементами которого являются пары вида `(key, value)`, где все ключи хешируемые, функция `make_multi_dict` возвращает отображение, ассоциирующее каждый ключ со списками, которые состоят из одного или нескольких значений, соответствующими ему в итерируемом объекте (если вы хотите получить результат в виде словаря типа `dict`, замените последнюю инструкцию инструкцией `return dict(d)`, но это может потребоваться лишь в редких случаях).

Если наличие дубликатов в результате для вас нежелательно, а каждое значение является хешируемым, используйте вызов `collections.defaultdict(set)` и метод `add` вместо метода `append` в цикле.

Класс deque

deque `deque(`*seq*`=(), `*maxlen*`=None)`

`deque` — это тип последовательности, экземпляры которого являются *двухсторонними очередями* (операции добавления и удаления элемента на любом из концов очереди выполняются очень быстро). Экземпляр *d* класса `deque` — это изменяемая последовательность, допускающая индексирование и итерирование (однако *d* не допускает извлечения срезов; можно индексировать только по одному элементу за раз, причем как для ссылки, так и для удаления или изменения связывания). Начальными элементами *d* являются элементы последовательности *seq*, расположенные в том же порядке. Атрибут *d*`.maxlen` доступен только для чтения. Если он имеет значение `None`, то длина *d* не ограничивается; если атрибут — целое число, то оно не должно быть отрицательным, и тогда длина *d* не может превышать *d*`.maxlen` (при добавлении слишком большого количества элементов предупреждающие сообщения не выводятся, однако это сопровождается удалением соответствующего количества элементов на другом конце очереди). Это полезный тип очереди с дисциплиной обслуживания "видны последние N объектов", также известный в других языках программирования как *кольцевой буфер*.

Перечень методов, поддерживаемых экземпляром *d*, приводится ниже.

append *d*`.append(`*item*`)`

Присоединяет элемент в конец *d*.

appendleft *d*`.appendleft(`*item*`)`

Присоединяет элемент в начало *d*.

clear *d*`.clear()`

Удаляет все элементы *d*, оставляя его пустым.

extend *d*`.extend(`*iterable*`)`

Присоединяет все элементы итерируемого объекта в конец *d*.

extendleft *d*.extendleft(*item*)

Присоединяет все элементы итерируемого объекта в начало *d*.

pop *d*.pop()

Удаляет и возвращает последний (крайний справа) элемент *d*. Если *d* — пустой объект, возбуждается исключение IndexError.

popleft *d*.popleft()

Удаляет и возвращает первый (крайний слева) элемент *d*. Если *d* — пустой объект, возбуждается исключение IndexError.

rotate *d*.rotate(*n*=1)

Прокручивает *d* на *n* шагов вправо (если *n*<0, выполняется прокручивание влево).

Класс namedtuple

Класс namedtuple — это функция-фабрика, создающая и возвращающая подкласс класса tuple, к элементам экземпляра которого можно обращаться не только по индексу, но и по ссылке на атрибут.

namedtuple namedtuple(*typename*, *fieldnames*)

Аргумент *typename* — это строка, являющаяся допустимым идентификатором (начинается с буквы, может быть продолжена буквами, цифрами и символом подчеркивания; не может быть зарезервированным словом, таким как 'class') и именующая новый тип, создаваемый и возвращаемый классом namedtuple. Аргумент *fieldnames* — последовательность строк, являющихся допустимыми идентификаторами и именующих атрибуты нового типа в том порядке, в каком они заданы (для удобства *fieldnames* может также быть одной строкой, в которой идентификаторы разделены пробелами или запятыми). Вызов namedtuple возвращает тип: вы можете связать этот тип с именем, а затем использовать его для создания неизменяемых экземпляров, инициализированных позиционными и именованными аргументами. Вызовы repr или str для этих экземпляров форматируют их в стиле именованных аргументов.

```
point = collections.namedtuple('point', 'x,y,z')
p = point(x=1,y=2,z=3) # может создаваться с
                       # именованными аргументами
x, y, z = p # может распаковываться, как обычный
            # кортеж
if p.x < p.y: # допускает обращение к элементам
              # как к атрибутам
    print(p) # форматирует как именованные аргументы
# вывод: point(x=1, y=2, z=3)
```

Именованный кортеж, такой как point, и его экземпляры, такие как *p*, также предоставляют атрибуты и методы, перечень которых приводится ниже (их имена начинаются с символа подчеркивания лишь для того, чтобы избежать конфликтов с именами полей, а не для того, чтобы обозначить их как "закрытые").

_asdict	*p*._asdict()
	Возвращает экземпляр dict, ключами которого являются имена полей *p*, а значениями — соответствующие элементы p.
_fields	*point*._fields
	Возвращает кортеж строк с именами полей, в данном случае ('x', 'y', 'z').
_make	*point*._make(*seq*)
	Возвращает экземпляр *point*, элементы которого инициализируются значениями из итерируемого объекта *seq* в том порядке, в котором они указаны (значение len(*seq*) должно быть равно значению len(*point*._fields)).
_replace	*p*._replace(**_kwargs_*)
	Возвращает копию *p*, в которой 0 или более элементов заменены в соответствии с переданными именованными аргументами.

Более подробную информацию о классе namedtuple и рекомендации по его применению можно найти в онлайн-документации (https://docs.python.org/3.3/library/collections.html#namedtuple-factory-function-for-tuples-with-named-fields).

Модуль functools

Модуль functools предоставляет функции и типы, поддерживающие функциональное программирование в Python (табл. 7.4).

Таблица 7.4. Функции модуля functools

Функция	Описание
cmp_to_key	cmp_to_key(*func*)
	Аргумент *func* должен быть вызываемым объектом с двумя аргументами, возвращающим число: <0, если первый аргумент следует считать "меньшим, чем" второй, >0, если справедливо обратное, и 0, если оба аргумента следует считать равными (аналогично использованию прежней встроенной функции cmp, признанной устаревшей в версии v2 и исключенной из версии v3, и прежнего именованного аргумента cmp функций sort и sorted). Функция cmp_to_key возвращает вызываемый объект *k*, пригодный для использования в качестве именованного аргумента key таких функций и методов, как sort, sorted, min, max и др. Это может быть полезным для преобразования программ, использующих аргумент старого стиля cmp, в программы, использующие аргумент нового стиля key, который требуется в версии v3 и настоятельно рекомендуется к использованию в версии v2

Функция	Описание
lru_cache	lru_cache(*max_size*=128, *typed*=False) (Только в версии v3. Чтобы использовать эту функцию в версии v2, установите с помощью команды pip install ретроподдержку, разработанную Джейсоном Кумбсом (https://pypi.python.org/pypi/backports.functools_lru_cache).) Данная функция представляет собой декоратор *мемоизации*, пригодный для декорирования функций, все аргументы которых хешируемые, и добавляющий в функцию кеш, который сохраняет последние max_size результатов (чтобы в кеше были сохранены все предыдущие результаты, значение max_size должно быть равным некоторой степени 2 или значению None). В случае повторного вызова декорированной функции с теми же значениями аргументов, которые уже находятся в кеше, она немедленно возвращает ранее кешированный результат, пропуская базовый код тела функции. Если аргумент *typed* имеет значение True, то аргументы, которые при сравнении оказываются равными, но относятся к различным типам, например 23 и 23.0, кешируются по отдельности. Более подробное описание вместе с примерами применения можно найти в онлайн-документации (https://docs.python.org/3/library/functools.html#functools.lru_cache)

partial	partial(*func*, **a*, ***k*)
	Аргументом *func* может быть любой вызываемый объект. Функция partial возвращает новый вызываемый объект *p*, который ведет себя в точности подобно объекту *func*, вызываемому с некоторыми позиционными и/или именованными аргументами, которые уже связанны со значениями, предоставляемыми *a* и *k*. Другими словами, partial создает новую функцию путем *фиксирования части аргументов func* — процесс, известный под названием *каррирование* (это название было присвоено данному преобразованию в честь математика Хаскелла Карри). Допустим, у вас имеется числовой список L и вы хотите усечь все отрицательные числа в этом списке до 0. Вот один из возможных способов сделать это:
	`L = map(functools.partial(max, 0), L)`
	То же самое можно сделать с помощью лямбда-выражения:
	`L = map(lambda x: max(0, x), L)`
	или (наиболее компактный способ) с помощью генератора списка:
	`L = [max(0, x) for x in L]`
	Функция functools.partial оказывается удобной в ситуациях, требующих использования обратных вызовов, как, например, в случае управляемого событиями кода для GUI или сетевых приложений (глава 18).

Функция	Описание
	Функция `partial` возвращает вызываемый объект с атрибутами `func` (обернутая функция), `args` (кортеж, состоящий из предварительно связанных позиционных параметров) и `keywords` (словарь, состоящий из предварительно связанных аргументов, или `None`)
`reduce`	`reduce(`*`func`*`, `*`seq`*`[, `*`init`*`])`
	Подобна встроенной функции `reduce` (см. табл. 7.2). В версии v3 `reduce` не является встроенной функцией, но по-прежнему доступна в модуле `functools`
`total_ordering`	Класс декоратора, пригодный для декорирования классов, которые предоставляют по крайней мере один метод, выполняющий проверку на неравенство, например `__lt__`, а также метод `__eq__`. Исходя из существующих методов класса класс декоратора `total_ordering` добавляет в него остальные методы, выполняющие проверку на неравенство, что избавляет вас от необходимости самостоятельно добавлять шаблонный код
`wraps`	`wraps(`*`wrapped`*`)`
	Декоратор, пригодный для декорирования функций, которые обертывают другую функцию, *`wrapped`* (часто — функций, вложенных в другие декораторы). Функция `wraps` копирует атрибуты `__name__`, `__doc__` и `__module__` функции *`wrapped`* в функцию-декоратор, тем самым улучшая поведение встроенной функции `help`, а также модуля `doctests` (раздел "Модуль `doctest`" в главе 16)

Модуль heapq

Модуль `heapq` использует алгоритмы *min heap* для поддержания списка в "почти отсортированном состоянии" по мере вставки и извлечения его элементов. Выполняемые модулем `heapq` операции работают быстрее, чем вызов метода `sort` списка после каждой вставки элемента, и намного быстрее, чем метод *бисекции* (половинного деления; `https://docs.python.org/3/library/bisect.html`). Для многих целей, таких как реализация "приоритетных очередей", почти отсортированный порядок, поддерживаемый модулем `heapq`, ни в чем не уступает полностью отсортированному порядку, но работает быстрее и проще в обслуживании.

`heapify` `heapify(`*`alist`*`)`

Выполняет перестановку элементов для преобразования списка в кучу, удовлетворяющую *условию min heap*:

- для любого `i>=0` выполняются соотношения
  `alist``[i]<=`*`alist`*`[2*i+1]` и
  `alist``[i]<=`*`alist`*`[2*i+2]`
- при условии что все индексы `<` `len(`*`alist`*`)`

Если список удовлетворяет условию кучи типа *min heap*, то первый элемент списка является наименьшим (или равным наименьшему). Отсортированный список удовлетворяет условию кучи, но этому условию удовлетворяют также многие другие перестановки элементов списка, не представляющие полностью отсортированное состояние. Быстродействие функции `heapify` характеризуется линейной зависимостью от размера списка: `O(len(alist))`.

heappop `heappop(alist)`

Удаляет и возвращает наименьший (первый) элемент `alist` — списка, удовлетворяющего условию кучи, и переставляет оставшиеся элементы таким образом, чтобы условие кучи удовлетворялось и после удаления элемента. Быстродействие функции `heapify` характеризуется логарифмической зависимостью от размера списка: `O(log(len(alist)))`.

heappush `heappush(alist, item)`

Вставляет элемент в `alist` — список, удовлетворяющий условию кучи, и переставляет оставшиеся элементы таким образом, чтобы условие кучи удовлетворялось и после вставки элемента. Быстродействие функции `heappush` характеризуется логарифмической зависимостью от размера списка: `O(log(len(alist)))`.

heappushpop `heappushpop(alist, item)`

Логический эквивалент вызова `heappush`, за которым следует вызов `heappop`, аналогичный следующему коду:

```
def heappushpop(alist, item):
    heappush(alist, item)
    return heappop(alist)
```

Быстродействие функции `heappushpop` характеризуется логарифмической зависимостью от размера списка — `O(log(len(alist)))`, и обычно эта функция работает быстрее, чем логически эквивалентная ей функция, приведенная выше. Функция `heappushpop` может быть вызвана с пустым списком `alist`: в этом случае она возвращает аргумент `item`, как и в том случае, если `item` меньше любого из существующих элементов `alist`.

heapreplace `heapreplace(alist, item)`

Логический эквивалент вызова функции `heappop`, за которым следует вызов функции `heappush`, аналогичный следующему коду:

```
def heapreplace(alist, item):
    try: return heappop(alist)
    finally: heappush(alist, item)
```

Быстродействие функции `heapreplace` характеризуется логарифмической зависимостью от размера списка: `O(log(len(alist)))`, и обычно эта функция работает быстрее, чем логически эквивалентная ей функция, приведенная выше. Функция `heapreplace` не может быть вызвана с пустым списком `alist`: она всегда возвращает элемент, уже имеющийся в `alist`, а не вставляемый элемент.

merge `merge(*iterables)`

Возвращает итератор, выдающий отсортированные (в порядке возрастания) элементы совокупности итерируемых объектов, каждый из которых должен быть отсортированным в порядке возрастания.

nlargest `nlargest(n, seq, key=None)`

Возвращает отсортированный в порядке убывания список *n* наибольших элементов итерируемого объекта *seq* (менее чем *n*, если количество элементов в *seq* меньше *n*). Подобна функции `sorted(seq, reverse=True)[:n]`, но работает быстрее для небольших значений *n*. Также допускается указание именованного аргумента `key`, как это допускается и для функции `sorted`.

nsmallest `nsmallest(n, seq, key=None)`

Возвращает отсортированный в порядке возрастания список *n* наименьших элементов итерируемого объекта *seq* (менее чем *n*, если количество элементов в *seq* меньше *n*). Подобна функции `sorted(seq)[:n]`, но работает быстрее для небольших значений *n*. Также допускается указание именованного аргумента `key`, как это допускается и для функции `sorted`.

Идиома "декорирование — сортировка — отмена декорирования"

Некоторые функции модуля `heapq`, хотя и используют операции сравнения, не поддерживают именованный аргумент `key`, обеспечивающий возможность настройки процесса сортировки. Иначе и не могло быть, поскольку эти функции выполняют операции над простым списком элементов на месте: им просто негде "припрятать" пользовательские ключи сравнения, вычисленные раз и навсегда.

Если вам одновременно нужны и функциональность кучи, и специализированные операции сравнения, можете воспользоваться доброй старой идиомой "декорирование — сортировка — отмена декорирования" (decorate-sort-undecorate — DSU), которую еще называют преобразованием Шварца (`https://ru.wikipedia.org/wiki/Преобразование_Шварца`). Эта идиома имела решающее значение для оптимизации сортировки в старых версиях Python, еще до введения функциональности `key`.

Применительно к модулю `heapq` идиома DSU выглядит следующим образом.

Декорирование. Создайте вспомогательный список *A*, каждый элемент которого является кортежем, начинающимся с ключа сортировки и заканчивающимся элементом исходного списка *L*.

Вызовите функции модуля `heapq` для *A*, в типичных случаях начиная с вызова `heapq.heapify(A)`.

Отмена декорирования. Извлекая элемент из *A*, в типичных случаях — путем вызова `heapq.heappop(A)`, возвращайте только последний элемент результирующего кортежа (соответствующий элементу исходного списка *L*).

Добавляя элемент в *A* путем вызова функции `heapq.heappush(A, item)`, декорируйте фактический элемент, который вы вставляете в кортеж, начинающийся с ключа сортировки.

Эту последовательность действий лучше всего обернуть в класс, как показано в следующем примере.

```
import heapq

class KeyHeap(object):
    def __init__(self, alist, key):
        self.heap = [
            (key(o), i, o)
            for i, o in enumerate(alist)]
        heapq.heapify(self.heap)
        self.key = key
        if alist:
            self.nexti = self.heap[-1][1] + 1
        else:
            self.nexti = 0

    def __len__(self):
        return len(self.heap)

    def push(self, o):
        heapq.heappush(
            self.heap,
            (self.key(o), self.nexti, o))
        self.nexti += 1

    def pop(self):
        return heapq.heappop(self.heap)[-1]
```

В этом примере мы используем увеличивающееся число посреди декорированного кортежа (после ключа сортировки и перед фактическим элементом), тем самым гарантируя, что фактические элементы никогда не будут сравниваться непосредственно, даже если их ключи сортировки равны (эта семантическая гарантия является важным аспектом функциональности аргумента `key` функции `sort` и других аналогичных функций).

Модуль `argparse`

При написании программ на Python, предназначенных для запуска из командной строки (или посредством "сценария оболочки" в Unix-подобных системах и "пакетного файла" в Windows), часто желательно предоставить пользователю возможность вводить *аргументы командной строки* (включая *опции командной строки*, которые в соответствии с общепринятым соглашением снабжаются префиксом в виде одного

или двух дефисов). В Python эту возможность обеспечивает атрибут `sys.argv` модуля `sys`, в котором аргументы командной строки сохраняются в виде списка строк (`sys.argv[0]` — имя, используемое для запуска программы; собственно аргументы находятся в подсписке `sys.argv[1:]`). Для обработки этих аргументов стандартная библиотека Python предлагает три модуля. Ниже мы рассмотрим лишь самый новый и мощный из них, `argparse`, ограничившись обсуждением лишь небольшого, наиболее важного подмножества богатой функциональности этого модуля. Намного больше информации можно найти в онлайн-документации Python (`https://docs.python.org/3/library/argparse.html`) и практическом руководстве по модулю `argparse` (`https://docs.python.org/3.4/howto/argparse.html`).

`ArgumentParser` `ArgumentParser(**kwargs)`

`ArgumentParser` — это класс, экземпляры которого разбирают аргументы. Он принимает много именованных аргументов, большинство из которых в основном предназначено для улучшения справочных сообщений, отображаемых программой, если она запускается с опциями `-h` или `--help` командной строки. Одним из аргументов, которые вы должны передать, является `description` — строка, содержащая краткое описание назначения программы.

После того как вы получите экземпляр *ap* класса `ArgumentParser`, подготовьте его, вызвав нужное количество раз метод *ap*`.add_argument`, а затем вызовите метод *ap*`.parse_args()` без аргументов (для разбора строки `sys.argv`): этот вызов возвращает экземпляр `argparse.Namespace`, содержащий аргументы и опции вашей программы в виде атрибутов.

Метод `add_argument` имеет обязательный первый аргумент: им может быть либо строка идентификатора для позиционных аргументов командной строки, либо имя флага для опций командной строки. В последнем случае передайте одно или несколько имен флагов. Опции часто имеют как короткое имя (один дефис, затем одиночный символ), так и длинное (два дефиса, затем идентификатор).

Вслед за позиционными аргументами передайте методу `add_argument` нуль или более именованных аргументов, позволяющих управлять его поведением. Ниже приведены наиболее часто используемые из них.

`action` Определяет действия, которые должен выполнить анализатор по отношению к данному аргументу. Значение по умолчанию, `'store'`, означает сохранение значения аргумента в пространстве имен (в имени, определяемом именованным аргументом `dest`). Другими полезными значениями являются `'store_true'` и `'store_false'`, которые сохраняют для опции булевы значения `True` и `False` соответственно (если опция отсутствует, то по умолчанию для нее сохраняются противоположные булевы значения, `False` и `True`), и `'append'`, сохраняющее список и присоединяющее каждое значение аргумента к этому списку (что позволяет указывать данную опцию несколько раз).

choices Набор допустимых значений данного аргумента командной строки (при попытке анализа аргумента, значение которого *отсутствует в этом наборе*, возбуждается исключение). По умолчанию ограничения отсутствуют.

default Значение, которое используется в случае отсутствия данного аргумента командной строки. Значение по умолчанию — `None`.

dest Имя атрибута, используемого для данного аргумента командной строки. По умолчанию определяется строкой первого позиционного аргумента после удаления из нее начальных символов дефиса.

help Строка, содержащая краткое описание аргумента и предназначенная для вывода справочных сообщений.

nargs Количество аргументов командной строки, собираемых в единый список, который ассоциируется с данным логическим аргументом (значение по умолчанию — `1`; в этом случае аргумент сохраняется в пространстве имен как отдельный элемент, а не в виде списка из одного элемента). Поддерживаются следующие значения: целое число `>0` (заданное количество аргументов сохраняется в виде списка), `'?'` (по возможности используется и сохраняется в виде отдельного элемента один аргумент командной строки; в случае отсутствия аргумента используется значение `default`), `'*'` (0 или несколько аргументов сохраняются в виде списка), `'+'` (1 или несколько аргументов сохраняются в виде списка) или `argparse.REMAINDER` (все остальные аргументы сохраняются в виде списка).

type Вызываемый объект, который принимает строку, часто представляющую тип, например `int`; используется для преобразования строковых значений в другие значения. Может быть экземпляром `argparse.FileType`, предназначенным для открытия строки как файла с данным именем (`FileType('r')` — для чтения, `FileType('w')` — для записи и т.д.).

Приведем простой пример использования класса `argparse` (сохраните этот код в файле *greet.py*).

```
import argparse
ap = argparse.ArgumentParser(description='Just an example')
ap.add_argument('who', nargs='?', default='World')
ap.add_argument('--formal', action='store_true')
ns = ap.parse_args()
if ns.formal:
    greet = 'Most felicitous salutations, o {}.'
else:
    greet = 'Hello, {}!'
print(greet.format(ns.who))
```

Программа, запущенная посредством команды `python greet.py`, выведет текст `Hello, World!`, тогда как в случае запуска посредством команды `python greet.py --formal Cornelia` будет выведен текст `Most felicitous salutations, o Cornelia.`

Модуль itertools

Модуль itertools предлагает высокопроизводительные компоненты, предназначенные для создания итераторов и манипулирования ими. При обработке крупных совокупностей элементов во многих случаях следует отдавать предпочтение итераторам, а не, например, спискам. Это объясняется тем, что итераторы применяют подход, в котором используются так называемые "ленивые" (отложенные) вычисления: итератор возвращает с помощью ключевого слова yield ("выдать") по одному элементу за раз по мере необходимости, а не сразу все элементы в виде списка (или другого рода последовательности), который весь должен храниться в памяти. Такой подход позволяет создавать даже неограниченные итераторы, в то время как списки всегда должны включать ограниченное количество элементов (поскольку объем памяти любого компьютера ограничен).

В табл. 7.5 приведены наиболее часто используемые атрибуты модуля itertools. Каждый из них представляет тип итератора, который можно вызывать для получения экземпляра интересующего вас типа, или функцию-фабрику, ведущую себя аналогичным образом. В разделах онлайн-документации Python, посвященных модулю itertools, содержится описание многих других атрибутов, в том числе *комбинаторных* генераторов, предназначенных для генерации перестановок, сочетаний и декартовых произведений, а также полезные сведения о классификации атрибутов itertools (https://docs.python.org/3/library/itertools.html).

В онлайн-документации также приведено множество готовых рецептов сочетания и использования атрибутов itertools. В этих шаблонных примерах предполагается, что код вашего модуля начинается инструкцией from itertools import *. Такой подход не рекомендуется и используется в упомянутых рецептах лишь для представления их кода в более компактной форме. Наилучший подход заключается в применении инструкции import itertools as it и последующем использовании ссылок вида it.*ссылка* вместо более длинной инструкции itertools.*ссылка*.

Таблица 7.5. Атрибуты модуля itertools

Атрибут	Описание
chain	chain(**iterables*) Возвращает элементы из первого аргумента, затем элементы из второго аргумента и т.д. до тех пор, пока не будет достигнут конец последнего аргумента, аналогично следующему выражению-генератору: (item for iterable in iterables for item in iterable)
chain.from_iterable	chain.from_iterable(*iterables*) Возвращает элементы из итерируемых объектов, заданных аргументом, в порядке их следования, аналогично следующему выражению-генератору:

Атрибут	Описание
	`(item for iterable in iterables for itemin iterable)`

compress	`compress(`*`data`*`, `*`conditions`*`)`
	Возвращает каждый элемент *data*, которому в *conditions* соответствует элемент с истинным вычисляемым значением, аналогично следующему выражению-генератору:
	`(item for item, cond in zip(data, conditions) if cond)`

count	`count(`*`start`*`=0, `*`step`*`=1)`
	Возвращает последовательные целые числа, начиная со значения `start`, аналогично следующему генератору:
	`def count(start=0, step=1):`
	`  while True:`
	`  yield start`
	`  start += step`

cycle	`cycle(`*`iterable`*`)`
	Возвращает каждый элемент *iterable*, бесконечно повторяя элементы с самого начала всякий раз, когда достигается последний элемент, аналогично следующему генератору:
	`def cycle(iterable):`
	`  saved = []`
	`  for item in iterable:`
	`  yield item`
	`  saved.append(item)`
	`  while saved:`
	`  for item in saved: yield item`

dropwhile	`dropwhile(`*`func`*`, `*`iterable`*`)`
	Опускает элементы итерируемого объекта, для которых функция *func* возвращает значение `True`, до тех пор, пока *func* не возвратит значение `False`, а затем возвращает каждый элемент аналогично следующему генератору:
	`def dropwhile(func,iterable):`
	`  iterator = iter(iterable)`
	`  for item in iterator:`
	`  if not func(item):`
	`  yield item`
	`  break`
	`  for item in iterator: yield item`

Атрибут	Описание

`groupby`	`groupby(`*`iterable`*`, `*`key`*`=None)`
	
	Как правило, итерируемый объект *`iterable`* должен быть уже отсортированным в соответствии с ключом *`key`* (как обычно, `None` означает функцию идентичности). Функция `groupby` возвращает пары `(`*`k`*`,` *`g`*`)`, каждая из которых представляет *группу* смежных элементов из *`iterable`* с одинаковыми значениями *`k`* для *`key`*`(`*`item`*`)`; каждый объект *`g`* — это итератор, возвращающий элементы группы. При продвижении объекта `groupby` предыдущие итераторы *`g`* становятся недействительными (поэтому, если элементы группы впоследствии должны подвергнуться дополнительной обработке, сохраните где-нибудь "снимок" группы в виде списка с помощью вызова `list(`*`g`*`)`).
	
	Другой способ получения групп, возвращаемых функцией `groupby`, заключается в том, что каждая из них заканчивается, как только изменяется значение *`key`*`(`*`item`*`)` (именно поэтому функция `groupby` обычно вызывается только для итерируемого объекта, который уже отсортирован в соответствии с *`key`*).
	
	Предположим, имеется множество (тип `set`) слов, представленных символами в нижнем регистре, и мы хотим получить словарь, который ассоциирует каждую начальную букву с самым длинным словом, начинающимся с этой буквы.
	
	`import itertools as it, operator`
	`def set2dict(aset):`
	`  first = operator.itemgetter(0)`
	`  words = sorted(aset, key=first)`
	`  adict = {}`
	`  for initial,`
	`  group in it.groupby(words, key=first):`
	`  adict[initial] = max(group, key=len)`
	`  return adict`

`ifilter`	`ifilter(`*`func`*`, `*`iterable`*`)`
	
	Возвращает те элементы итерируемого объекта, для которых *`func`* возвращает значение `True`, аналогично следующему выражению-генератору:
	
	`(item for item in iterable if func(item))`
	
	В качестве *`func`* можно указать любой вызываемый объект, который принимает один аргумент, или `None`. Если в качестве *`func`* указано значение `None`, `ifilter` возвращает элементы, вычисление которых дает истинное значение, аналогично следующему выражению-генератору:
	
	`(item for item in iterable if item)`

Атрибут	Описание

ifilterfalse	`ifilterfalse(`*`func`*`, `*`iterable`*`)`
	Возвращает те элементы итерируемого объекта, для которых *`func`* возвращает значение `False`, аналогично следующему выражению-генератору:
	`(item for item in iterable if not func(item))`
	Если в качестве *`func`* указано значение `None`, `ifilter` возвращает элементы, вычисление которых дает ложное значение, аналогично следующему выражению-генератору:
	`(item for item in iterable if not item)`

imap	`imap(`*`func`*`,*`*`iterables`*`)`
	Возвращает результаты вызова *`func`*, принимающей по одному аргументу из каждого итерируемого объекта. Вычисления прекращаются по исчерпании самого короткого итерируемого объекта. Действует аналогично следующему генератору:
	`def imap(func,*iterables):`
	`  iters = [iter(x) for x in iterables]`
	`  while True: yield func(*(next(x) for x in iters))`

islice	`islice(`*`iterable`*`[, `*`start`*`], `*`stop`*`[, `*`step`*`])`
	Возвращает элементы итерируемого объекта, пропуская первые *`start`* элементов (по умолчанию — `0`), пока не будет достигнут элемент с индексом *`stop`* (который не возвращается), продвигаясь на *`step`* шагов (по умолчанию — `1`) за один раз. Все аргументы должны быть неотрицательными числами (или `None`), а значение *`step`* должно быть `>0`. За исключением проверок и необязательных аргументов, действует аналогично следующему генератору:
	`def islice(iterable,start,stop,step=1):`
	`  en = enumerate(iterable)`
	`  n = stop`
	`  for n, item in en:`
	`  if n>=start: break`
	`  while n<stop:`
	`  yield item`
	`  for x in range(step): n, item = next(en)`

izip	Возвращает кортежи, включающие по одному соответствующему элементу из каждого из объектов *`iterables`*; вычисления прекращаются по исчерпании самого короткого из элементов *`iterables`*. Действует аналогично вызову `imap(`*`tuple`*`, *`*`iterables`*`)`. Используется только в версии v2; в версии v3

Атрибут	Описание
	`izip(*iterables)` эту же функциональность предлагает встроенная функция `zip`, описанная в табл. 7.2.
`izip_longest`	`izip_longest(*iterables, fillvalue=None)` Возвращает кортежи, включающие по одному соответствующему элементу из каждого итерируемого объекта; вычисления прекращаются по исчерпании самого длинного итерируемого объекта. Ведет себя так, как если бы каждый из остальных объектов был "дополнен" до той же длины ссылками на *`fillvalue`*.

`repeat`	`repeat(item[, times])`
	Повторно возвращает элемент *`item`*, аналогично следующему выражению-генератору:
	`(item for x in range(times))`
	Отсутствию *`times`* соответствует неограниченный итератор, возвращающий потенциально бесконечное количество элементов, аналогично следующему генератору:
	`def repeat_unbounded(item):`
	`  while True: yield item`

`starmap`	`starmap(func, iterable)`
	Возвращает значение *`func`*`(*`*`item`*`)` для каждого элемента итерируемого объекта (каждый элемент должен быть итерируемым объектом, обычно кортежем), аналогично следующему генератору:
	`def starmap(func,iterable):`
	`  for item in iterable:`
	`  yield func(*item)`

`takewhile`	`takewhile(func, iterable)`
	Возвращает элементы из итерируемого объекта при условии, что вызов *`func`*`(`*`item`*`)` возвращает истинное значение, аналогично следующему генератору:
	`def takewhile(func,iterable):`
	`  for item in iterable:`
	`  if func(item):`
	`  yield item`
	`  else:`
	`  break`

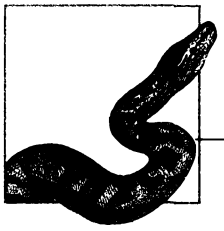
Атрибут	Описание
tee	`tee(`*`iterable`*`, `*`n`*`=2)` Возвращает кортеж из *n* независимых итераторов, каждый из которых возвращает те же элементы, которые содержатся в *iterable*. Возвращаемые итераторы не зависят друг от друга, но они *не являются* независимыми от *iterable*. Избегайте изменения объекта *iterable* на протяжении того времени, в течение которого вы используете любой из возвращаемых итераторов.

Мы привели эквивалентные генераторы и выражения-генераторы для многих атрибутов модуля `itertools`, но очень важно не забывать о преимуществе в скорости, обеспечиваемом типами `itertools`. В качестве примера рассмотрим повторение некоторого действия 10 раз:

```
for _ in itertools.repeat(0, 10): pass
```

В зависимости от используемого выпуска Python и платформы, этот код выполняется на 10–20% быстрее, чем обычный альтернативный код:

```
for _ in range(10): pass
```



8

Строки и байты

В версии v3 текстовые строки Unicode предоставляются типом `str` вместе со всеми его операторами, встроенными функциями, методами и специально предназначенными для этого модулями. В ней также имеется в некотором смысле аналогичный тип `bytes`, представляющий произвольные двоичные данные в виде последовательностей байтов, также известных как *строки байтов* или *байтовые строки*. В этом состоит одно из главных отличий версии v3 от версии v2, в которой тип `str` — это последовательность байтов, тогда как текстовые строки Unicode — это тип `unicode`. В обеих версиях многие операции над текстом можно выполнять с использованием объектов любого типа.

В этой главе рассмотрены такие темы, как методы строковых объектов (раздел "Методы строковых и байтовых объектов"), форматирование строк (раздел "Форматирование строк"), а также модули `string` (раздел "Модуль `string`") и `pprint` (раздел "Модуль `pprint`"). Вопросам, связанным со спецификой текстовых строк Unicode, посвящен раздел "Unicode". Новые форматированные строковые литералы (введены в версии v3.6) рассмотрены в разделе "Новое в версии 3.6: форматированные строковые литералы".

Методы строковых и байтовых объектов

Объекты `str` и `bytes` представляют неизменяемые последовательности, рассмотренные в разделе "Строки" главы 3. К ним применимы все операции, выполняемые над неизменяемыми последовательностями (повторение, конкатенация, индексирование и выделение срезов), которые возвращают объект того же типа. Кроме того, строковый или байтовый объект *s* поддерживает несколько немутирующих методов, описанных в табл. 8.1.

Если отдельно не оговорено иное, то обсуждаемые методы имеются в объектах обоих типов. В версии v3 методы объектов `str` возвращают строку Unicode, тогда как методы объектов `bytes` возвращают байтовую строку (в версии v2 за текстовые строки, т.е. строки Unicode, отвечает тип `unicode`, а за байтовые — тип `str`). Термины "буквы", "пробельные символы" и т.п. относятся к соответствующим атрибутам модуля `string` и обсуждаются в разделе "Модуль `string`".

В табл. 8.1 при указании целочисленных значений, используемых по умолчанию, вместо них используется более лаконичное выражение `sys.maxsize`, на практике означающее "любое число, каким бы большим оно ни было".

Таблица 8.1. Методы строковых и байтовых объектов

Метод	Описание
capitalize	*s*.capitalize()
	Возвращает копию строки *s*, в которой первый символ, если он является буквой, записан в верхнем регистре, а все остальные буквы, если они есть, записаны в нижнем регистре
casefold	*s*.casefold()
	Только для типа `str` в версии v3. Возвращает строку, обработанную в соответствии с алгоритмом, описанным в разделе 3.13 стандарта Unicode. Аналогичен методу *s*.lower (описан далее), но учитывает более широкие эквивалентности, такие, например, как эквивалентность в немецком языке символов 'β' и 'ss', и поэтому лучше приспособлен для сопоставления символов в нижнем регистре
center	*s*.center(*n*, *fillchar*=' ')
	Возвращает строку длиной max(len(*s*), *n*), которая содержит в своей центральной части копию строки *s*, окруженную равным количеством копий символа *fillchar* с обеих сторон (например, результат вызова 'ciao'.center(2) — строка 'ciao', а результат вызова 'x'.center(4,'_') — строка '_x__')
count	*s*.count(*sub*, *start*=0, *end*=sys.maxsize)
	Возвращает количество неперекрывающихся вхождений подстроки *sub* в срезе *s*[*start*:*end*]
decode	s.decode(*encoding*='utf-8', *errors*='strict')
	Только для типа `bytes`. Возвращает объект `str`, декодированный из байтов *s* согласно заданной кодировке. Аргумент *errors* определяет способ обработки ошибок декодирования: 'strict' — возбуждаются исключения UnicodeError, 'ignore' — некорректные данные игнорируются, 'replace' — некорректные данные заменяются вопросительными знаками. Более подробная информация содержится в разделе "Unicode". Другие значения могут быть зарегистрированы посредством вызова codec.register_error() (см. табл. 8.7)
encode	s.encode(*encoding*=None, *errors*='strict')
	Только для типа `str`. Возвращает объект `bytes`, полученный из *s* с использованием заданной кодировки и способа обработки ошибок. Более подробная информация содержится в разделе "Unicode"

Метод	Описание
endswith	*s*.endswith(*suffix*, *start*=0, *end*=sys.maxsize) Возвращает значение True, если *s*[*start*:*end*] заканчивается строкой *suffix*, в противном случае возвращается значение False. Аргумент *suffix* может быть представлен кортежем строк, и в этом случае endswith возвращает значение True, если *s*[*start*:*end*] заканчивается одной из них
expandtabs	*s*.expandtabs(*tabsize*=8) Возвращает копию строки *s*, в которой каждый символ табуляции заменен одним или несколькими пробелами, с позициями табуляции, расположенными через каждые *tabsize* символов
find	*s*.find(*sub*,*start*=0,*end*=sys.maxsize) Возвращает наименьший индекс, с которого начинается вхождение подстроки *sub* в *s*, при условии, что *sub* полностью содержится в срезе *s*[*start*:*end*]. Например, результат вызова 'banana'.find('na'), как и вызова 'banana'.find('na',1), — 2, тогда как результат вызова 'banana'.find('na',3), равно как и вызова 'banana'.find('na',-2), — 4. Если строка *sub* не найдена, метод find возвращает значение -1
format	*s*.format(**args*,***kwargs*) Только для типа str. Форматирует позиционные и именованные аргументы в соответствии с инструкциями форматирования, содержащимися в строке *s*. Более подробная информация приведена в разделе "Форматирование строк"
format_map	*s*.format_map(*mapping*) Только для типа str в версии v3. Форматирует аргумент *mapping* в соответствии с инструкциями форматирования, содержащимися в строке *s*. Эквивалентен методу *s*.format(***mapping*), но использует непосредственно отображение *mapping*
index	*s*.index(*sub*, *start*=0, *end*=sys.maxsize) Подобен методу *find*, но возбуждает исключение ValueError, если подстрока *sub* не найдена
isalnum	*s*.isalnum() Возвращает значение True, если len(*s*) больше 0 и все символы в *s* являются буквами или цифрами. Если строка *s* — пустая или хотя бы один из символов *s* не является ни буквой, ни цифрой, метод isalnum возвращает значение False
isalpha	*s*.isalpha() Возвращает значение True, если len(*s*) больше 0 и все символы в *s* являются буквами. Если строка *s* — пустая или хотя бы один из символов *s* не является буквой, метод isalpha возвращает значение False

Метод	Описание
isdecimal	`s.isdecimal()` Только для типа `str` в версии v3. Возвращает значение `True`, если `len(s)` больше `0` и все символы `s` могут быть использованы для образования чисел в десятичной системе счисления. Это определение включает символы Unicode, определенные как арабские цифры*
isdigit	`s.isdigit()` Возвращает значение `True`, если `len(s)` больше `0` и все символы `s` являются цифрами. Если строка `s` — пустая или хотя бы один символ в `s` не является цифрой, метод `isdigit` возвращает значение `False`
isidentifier	`s.isidentifier()` Только для типа `str` в версии v3. Возвращает значение `True`, если `s` является действительным идентификатором в соответствии с определением в языке Python. Ключевые слова также соответствуют этому определению, и поэтому, например, вызов `'class'.isidentifier()` возвращает значение `True`
islower	`s.islower()` Возвращает значение `True`, если все буквы в `s` являются буквами нижнего регистра. Если `s` не содержит букв или хотя бы одна из них является буквой верхнего регистра, метод `islower` возвращает значение `False`.
isnumeric	`s.isnumeric()` Только для типа `str` в версии v3. Аналогичен методу `s.isdigit()`, но использует более широкое определение цифровых символов, которое включает все символы, определенные как цифровые в стандарте Unicode (например, дроби)
isprintable	`s.isprintable()` Только для типа `str` в версии v3. Возвращает значение `True`, если все символы в `s` являются пробелами (`'\x20'`) или определены в стандарте Unicode как печатаемые символы. В отличие от других методов, названия которых начинаются с "is", вызов метода `'  '.isprintable()` возвращает значение `True`
isspace	`s.isspace()` Возвращает значение `True`, если `len(s)` больше `0` и все символы в `s` являются пробельными. Если строка `s` — пустая или хотя бы один из символов `s` не является пробельным, метод `isspace` возвращает значение `False`
istitle	`s.istitle()` Возвращает значение `True`, если каждая непрерывная последовательность букв в строке начинается с прописной буквы, а все остальные буквы такой последовательности являются буквами нижнего регистра (например, вызов `'King Lear'.istitle()` возвращает значение `True`). Если `s`

Метод	Описание
	не содержит букв или хотя бы одна из букв нарушает указанное условие, метод `istitle` возвращает значение `False` (например, вызовы `'1900'.istitle()` и `'Troilus and Cressida'.istitle()` возвращают значение `False`)
isupper	*s*`.isupper()` Возвращает значение `True`, если все буквы в *s* являются буквами верхнего регистра. Если *s* не содержит букв или хотя бы одна буква в *s* является буквой в верхнем регистре, метод `isupper` возвращает значение `False`
join	*s*`.join(`*seq*`)` Возвращает строку, полученную посредством конкатенации элементов *seq*, которые должны быть итерируемыми объектами, содержащими строки, и вставки копии *s* между каждой парой элементов (например, вызов `''.join(str(x) for x in range(7))` возвращает строку `'0123456'`, а вызов `'x'.join('aeiou')` — строку `'axexixoxu'`)
ljust	*s*`.ljust(`*n*`, `*fillchar*`=' ')` Возвращает строку длиной `max(len(`*s*`),` *n*`)`, начинающуюся со строки *s*, за которой следуют нуль или более завершающих копий символа *fillchar*
lower	*s*`.lower()` Возвращает копию строки *s*, в которой все буквы переведены в нижний регистр
lstrip	*s*`.lstrip(`*x*`=string.whitespace)` Возвращает копию строки *s*, из которой удалены все указанные начальные символы, найденные в строке *x*. Например, вызов `'banana'.lstrip('ab')` возвращает строку `'nana'`
replace	*s*`.replace(`*old*`, `*new*`, `*maxsplit*`=sys.maxsize)` Возвращает копию строки *s*, в которой неперекрывающиеся вхождения подстроки *old* заменены строкой *new*. Аргумент *maxsplit* определяет максимальное количество возможных замен (например, вызов `'banana'.replace('a','e',2)` возвращает строку `'benena'`)
rfind	`s.rfind(`*sub*`, start=0, `*end*`=sys.maxsize)` Возвращает наибольший индекс, с которого начинается вхождение подстроки *sub* в *s*, при условии, что *sub* полностью содержится в срезе *s*`[`*start*`:`*end*`]`. Если подстрока *sub* не найдена, метод `rfind` возвращает значение `-1`
rindex	*s*`.rindex(`*sub*`, `*start*`=0, `*end*`=sys.maxsize)` Подобен методу `rfind`, но возбуждает исключение `ValueError`, если подстрока *sub* не найдена

Метод	Описание
rjust	`s.rjust(n, fillchar=' ')` Возвращает строку длиной `max(len(s), n)`, заканчивающуюся копией строки `s`, которой предшествуют нуль или более начальных копий символа `fillchar`
rstrip	`s.rstrip(x=string.whitespace)` Возвращает копию строки `s`, из которой удалены все указанные завершающие символы, найденные в строке `x`. Например, вызов `'banana'.rstrip('ab')` возвращает строку `'banan'`

split	`s.split(sep=None, maxsplit=sys.maxsize)`
	Возвращает список `L`, содержащий вплоть до `maxsplit+1` строк. Каждый элемент `L` представляет собой "слово" из строки `s`, в которой `sep` служит разделителем слов. Если количество слов в `s` превышает `maxsplit`, то последним элементом `L` является подстрока `s`, которая следует за первыми `maxsplit` слов. Если для `sep` указано значение `None`, то в качестве разделителя используется любая строка, состоящая из пробельных символов. Например, вызов
	`'four score and seven years'.split(None,3)`
	возвращает строку
	`['four','score','and','seven years']`
	Обратите внимание на разницу между разбиением строк с использованием аргумента `None` метода `split` (разделителем служит любая строка, состоящая из пробельных символов) и аргумента `' '` (разделителем служит любой одиночный символ пробела, а не другие пробельные символы, такие как символ табуляции, символ перевода строки или строка пробелов).
	`>>> x = 'a  b' # два пробела между a и b`
	`>>> x.split()  # или, эквивалентно, x.split(None)`
	`['a', 'b']`
	`>>> x.split(' ')`
	`['a', '', 'b']`
	В первом случае в качестве разделителя используется расположенная посередине подстрока, состоящая из двух символов пробела; во втором случае — каждый одиночный пробел, поэтому между двумя пробелами существует пустая строка

splitlines	`s.splitlines(keepends=False)` Подобен методу `s.split('\n')`. Однако, если значением аргумента `keepends` является `True`, то в каждый элемент результирующего списка включается завершающий символ `'\n'`

Метод	Описание
startswith	`s.startswith(prefix, start=0, end=sys.maxsize)` Возвращает значение `True`, если срез `s[start:end]` начинается со строки `prefix`; в противном случае возвращается значение `False`. Аргументом `prefix` может быть кортеж строк, и в этом случае метод `startswith` возвращает значение `True`, если срез `s[start:end]` начинается с любой из этих строк
strip	`s.strip(x=string.whitespace)` Возвращает копию строки `s`, из которой удалены как начальные, так и завершающие символы, найденные в строке `x`. Например, вызов `'banana'.strip('ab')` возвращает строку `'nan'`
swapcase	`s.swapcase()` Возвращает копию строки `s`, в которой все буквы верхнего регистра преобразованы в буквы нижнего регистра и наоборот
title	`s.title()` Возвращает копию строки `s`, преобразованную таким образом, что каждая непрерывная последовательность символов начинается с прописной буквы, а все остальные буквы такой последовательности являются буквами нижнего регистра

translate	`s.translate(table)`
	Возвращает копию строки `s`, преобразованную путем перекодирования или удаления символов, которые удается найти в `table`. В версии v3 (а также в версии v2, если `s` — строка `unicode`) `table` — это словарь типа `dict`, ключами которого являются порядковые числительные Unicode, а значениями — порядковые числительные Unicode, строки Unicode или `None` (для удаления символа). Приведенный ниже код может работать в обеих версиях, v2 и v3, причем в версии v3 префикс `u` перед строками является избыточным и может быть опущен:
	`print(u'banana'.translate({ord('a'):None,ord('n'):u'ze'}))`
	`# вывод: 'bzeze'`
	В версии v2, если `s` — строка байтов, то ее метод `translate` работает несколько иначе (детальнее см. в онлайн-документации). Ниже приведен пример использования метода `translate` для версии v2.
	`import string`
	`identity = string.maketrans('','')`
	`print('some string'.translate(identity,'aeiou'))`
	`# вывод: sm strng`
	Эквивалентный код для Unicode-строк или для версии v3:
	`no_vowels = dict.fromkeys(ord(x) for x in 'aeiou')`
	`print(u'some string'.translate(no_vowels))`
	`# вывод: sm strng`

Метод	Описание

	Ниже приведен пример для версии v2, в котором все гласные заменяются буквой *a*, а все буквы *s* удаляются.
	`intoas = string.maketrans('eiou','aaaa')`
	`print('some string'.translate(intoas))`
	`# вывод: sama strang`
	`print('some string'.translate(intoas,'s'))`
	`# вывод: ama trang`
	Эквивалентный код для Unicode-строк или для версии v3:
	`intoas = dict.fromkeys((ord(x) for x in 'eiou'), 'a')`
	`print(u'some string'.translate(intoas))`
	`# вывод: sama strang`
	`intoas_nos = dict(intoas, s='None')`
	`print(u'some string'.translate(intoas_nos))`
	`# вывод: ama trang`

upper	*s*.upper() Возвращает копию строки *s* с переводом всех встречающихся в ней букв в верхний регистр

* Обратите внимание на то, что этому определению удовлетворяют также строки, представляющие числа с использованием символов точки (.) и запятой (,) в качестве разделителей.

Модуль string

Модуль string предоставляет несколько полезных атрибутов.

ascii_letters
: Строка ascii_lowercase+ascii_uppercase.

ascii_lowercase
: Строка 'abcdefghijklmnopqrstuvwxyz'.

ascii_uppercase
: Строка 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.

digits
: Строка '0123456789'.

hexdigits
: Строка '0123456789abcdefABCDEF'.

octdigits
: Строка '01234567'.

`punctuation`
: Строка `'!"#$%&\'( )*+,-./:;<=>?@[\]^_'{|}~'` (т.е. все символы ASCII, которые считаются знаками пунктуации в локали `'C'`; не зависит от того, какая локаль является активной).

`printable`
: Строка, состоящая из символов ASCII, которые считаются печатаемыми (т.е. цифры, буквы, знаки пунктуации и пробельные символы).

`whitespace`
: Строка, содержащая символы ASCII, которые считаются пробельными символами: по крайней мере, символы пробела, табуляции, перевода строки и возврата каретки, но могут быть и другие символы (например, некоторые управляющие), в зависимости от локали.

Вы не должны повторно связывать эти атрибуты: результат таких действий будет неопределенным, поскольку значения этих атрибутов используются другими компонентами библиотеки Python.

Модуль `string` также предоставляет класс `Formatter`, которому посвящен раздел "Форматирование строк".

Форматирование строк

В версии v3 появился новый мощный способ форматирования строк, также получивший ретроподдержку в версии v2. Строки Unicode (но *не* байтовые строки в версии v3) предоставляют метод `format`, аргументы которого, переданные ему при вызове, вставляются в соответствии с определенными правилами в поля замены, обозначаемые парой фигурных скобок в форматируемой строке-шаблоне, для которой вызывается данный метод. Наиболее общая форма вызова метода `format` имеет следующий вид:

```
'[строка][{поле_замены}]...'.format(*args, **kwargs)
```

Здесь *строка* — это обычный текст, а *поле_замены* — поле, подлежащее замене одним из аргументов, указанных при вызове метода `format`. Если поля замены отсутствуют, то возвращается строковый объект, представляющий исходное содержимое строки-шаблона.

Процесс форматирования лучше всего представлять себе как последовательность операций, каждой из которых управляет соответствующее поле замены. Каждое значение, подлежащее форматированию, прежде всего *выбирается*, затем, если это необходимо, *преобразуется* и, наконец, *форматируется*. Это описание согласуется со следующей наиболее общей структурой поля замены:

```
{[селектор] [!преобразование] [:спецификация_формата]}
```

Выбор значений

В случае простых строк-шаблонов вызов метода `format` имеет следующий вид:

```
'[строка]{[селектор]}...'.format(*args, **kwargs)
```

Здесь *селектор* — выражение, с помощью которого осуществляется выбор форматируемого значения, способное обрабатывать как позиционные (*args*), так и именованные (*kwargs*) аргументы метода `format`. В простейшем случае пустых полей замены (`{}`), когда селекторы значений не указаны явно, соответствие между полями и позиционными аргументами устанавливается автоматически на основании их относительных позиций.

```
>>> 'First: {} second: {}'.format(1, 'two')
'First: 1 second: two'
```

На позиционные аргументы также можно ссылаться по номерам, соответствующим их позициям в списке аргументов, что позволяет использовать значения аргументов повторно или с несоблюдением порядка их следования.

```
>>> 'Second: {1} first: {0}'.format(1, 'two')
'Second: two first: 1'
```

Поля с автоматической и явной нумерацией аргументов нельзя смешивать: необходимо использовать только один из этих способов. В то же время любой из этих способов допускает смешивание с именами именованных аргументов, используемыми в качестве селекторов.

```
>>> 'a: {a}, 1st: {}, 2nd: {}, a again: {a}'.format(1, 'two', a=3)
'a: 3, 1st: 1, 2nd: two, a again: 3'
>>> 'a: {a} first:{0} second: {1} first: {0}'.format(1, 'two', a=3)
'a: 3 first:1 second: two first: 1'
```

Если аргументом является последовательность, то при выборе значения, подлежащего форматированию, в селекторе можно использовать индексирование. Это относится как к позиционным (нумеруемым автоматически или явно), так и именованным аргументам.

```
>>> 'p0[1]: {[1]} p1[0]: {[0]}'.format(('zero', 'one'),
('two', 'three'))
'p0[1]: one p1[0]: two'
>>> 'p1[0]: {1[0]} p0[1]: {0[1]}'.format(('zero', 'one'),
('two', 'three'))
'p1[0]: two p0[1]: one'
>>> '{} {} {a[2]}'.format(1, 2, a=(5, 4, 3))
'1 2 3'
```

Если аргументом является составной объект, то для выбора его атрибутов в качестве форматируемых значений можно использовать точечную нотацию. Ниже приведен пример с комплексными числами, которые имеют атрибуты `real` и `imag`,

предназначенные для хранения соответственно действительной и мнимой частей числа.

```
>>> 'First r: {.real} Second i: {a.imag}'.format(1+2j, a=3+4j)
'First r: 1.0 Second i: 4.0'
```

В случае необходимости операции индексирования и выбора атрибута могут применяться многократно.

Преобразование значений

Вы также можете задать выполняемое по умолчанию преобразование значения с помощью одного из его методов перед его вставкой в выходную строку. Для этого в поле замены вслед за селектором необходимо указать флаг преобразования в виде символа восклицательного знака (`!`), за которым следует буквенное обозначение используемого метода:

```
'{[селектор] [!преобразование]}'.format(*args, **kwargs)
```

Паре символов `!s` соответствует применение метода `__str__`, паре символов `!r` — применение метода `__repr__`, а паре символов `!a` — применение встроенного метода `ascii` (только в версии v3).

При выполнении оставшейся части процесса форматирования вместо значения, первоначально выбранного с помощью селектора, используется преобразованное значение.

Форматирование значений

В тех случаях, когда требуется дополнительное форматирование значения, используется более полная форма поля замены:

```
'{[селектор][!преобразование][:спецификация_формата]}'.format(val)
```

Она включает завершающую (необязательную) часть, известную под названием *спецификатор (описатель) формата*. Отсутствие двоеточия в поле замены говорит о том, что преобразованное значение `val` используется без какого-либо дополнительного форматирования. Спецификатор формата может включать целый ряд необязательных опций (индикаторов), представленных ниже.

```
[заполнитель[выравнивание]][знак][#][ширина][,[.точность][тип]
```

Опции выравнивания и заполнения

Если требуется выравнивание отформатированного значения, то это достигается за счет заполнения соответствующих позиций указанным символом-заполнителем до нужной ширины поля. По умолчанию в качестве символа-заполнителя используется пробел, но вместо него можно использовать другой символ (который не может быть открывающей или закрывающей фигурной скобкой), указав его перед индикатором выравнивания (табл. 8.2).

Таблица 8.2. Индикаторы выравнивания

Индикатор	Описание
`'<'`	Выравнивание по левому краю поля
`'>'`	Выравнивание по правому краю поля
`'^'`	Выравнивание по центру поля
`'='`	Работает только для числовых типов: символы-заполнители добавляются между знаком и первой цифрой числа

Если выравнивание не определено явно, то по умолчанию большинство значений выравнивается по левому краю, за исключением числовых значений, которые выравниваются по правому краю. Если ширина поля не определяется далее в спецификаторе формата, то независимо от параметров заполнения и выравнивания символы-заполнители не добавляются.

Опция знака

Только для числовых типов можно указать способ обозначения положительных и отрицательных чисел, включив в спецификацию формата необязательный индикатор знака (табл. 8.3).

Таблица 8.3. Индикаторы знака

Индикатор	Описание
`'+'`	Вставляет `'+'` в качестве знака для положительных чисел и `'-'` в качестве знака для отрицательных чисел
`'-'`	Вставляет `'-'` в качестве знака для отрицательных чисел; для положительных чисел знак не используется (поведение по умолчанию, если индикатор знака не указан)
`' '`	Вставляет `' '` в качестве знака для положительных чисел и `'-'` в качестве знака для отрицательных чисел

Индикатор системы счисления

Только при форматировании целочисленных значений можно использовать индикатор системы счисления (символ `'#'`). Если он имеется, то отформатированным двоичным числам будут предшествовать символы `'0b'`, восьмеричным — символы `'0o'`, а шестнадцатеричным — символы `'0x'`. Например, выражение `'{:x}'.format(23)` дает строку `'17'`, тогда как выражение `'{:#x}'.format(23)'` — строку `'0x17'`.

Ширина поля

Также существует возможность задать ширину поля, отводимого для форматируемого значения. Если указанная ширина меньше длины значения, то ширина поля автоматически устанавливается такой, чтобы ее было достаточно для охвата значения (значения не обрезаются). Если выравнивание не задано, то значение выравнивается по левому краю (за исключением чисел, которые выравниваются по правому краю).

```
>>> str = 'a string'
>>> '{:^12s}'.format(str)
'  a string  '
>>> '{:.>12s}'.format(str)
'....a string'
```

Запятая в качестве разделителя групп разрядов

Только для числовых значений и только для десятичного (используемого по умолчанию) форматного типа допускается включение в спецификацию символа запятой (`,`), задающего использование запятой в качестве разделителя групп разрядов в результирующем значении. Это поведение игнорирует параметры системной локали. Если требуется, чтобы символы разделителя групп разрядов и десятичной точки брались из системной локали, используйте форматный тип `'n'`, описанный в табл. 8.4.

```
print('{:,}'.format(12345678))
# вывод: 12,345,678
```

Спецификатор точности

Спецификатор точности (например, `.2`) имеет разный смысл для различных форматных типов (см. следующий раздел), причем для большинства числовых форматов в качестве значения по умолчанию используется спецификатор `.6`. Для форматных типов `f` и `F` этот спецификатор определяет количество *десятичных* цифр, до которых должно округляться число в процессе форматирования. Для форматных типов `g` и `G` он определяет количество *значащих* цифр, до которых должно быть округлено значение. Для нечисловых типов он определяет *усечение* значения до заданного количества символов, отсчитываемых от левого края, прежде чем будет применено форматирование.

```
>>> s = 'a string'
>>> x = 1.12345
>>> 'as f: {:.4f}'.format(x)
'as f: 1.1235'
>>> 'as g: {:.4g}'.format(x)
'as g: 1.123'
>>> 'as s: {:.6s}'.format(s)
'as s: a stri'
```

Форматный тип

Спецификация формата заканчивается указанием необязательного *форматного типа* (типа форматирования), который определяет способ представления значения с заданными опциями ширины и точности. Если форматный тип опущен, то к значению применяется форматный тип, заданный по умолчанию.

Для форматирования строк Unicode используется форматный тип `s`.

Допустимые значения форматных типов для целых чисел приведены в табл. 8.4.

Таблица 8.4. Форматные типы для целых чисел

Форматный тип	Описание
`'b'`	Двоичный формат — последовательности единиц и нулей
`'c'`	Символ Unicode, порядковое значение которого является форматируемым значением
`'d'`	Десятичный формат (используется по умолчанию)
`'o'`	Восьмеричный формат — последовательности восьмеричных цифр
`'x'` или `'X'`	Шестнадцатеричный формат — последовательности шестнадцатеричных цифр с буквами в нижнем и верхнем регистре соответственно
`'n'`	Десятичный формат с использованием разделителей из соответствующей локали (запятые в Великобритании и США), если она установлена

Для чисел с плавающей точкой используется другой набор форматных типов, представленный в табл. 8.5.

Таблица 8.5. Форматные типы для чисел с плавающей точкой

Форматный тип	Описание
`'e'` или `'E'`	Экспоненциальный формат — научная нотация, включающая целое число в диапазоне от нуля до девяти и букву `'e'` или `'E'` непосредственно перед экспонентой
`'f'`или `'F'`	Формат чисел с фиксированной точностью, включающий бесконечные значения (`'inf'`) и значение "не число" (`'nan'`) в нижнем или верхнем регистре
`'g'` или `'G'`	Общий числовой формат. По возможности использует формат чисел с фиксированной точностью, а если это невозможно — экспоненциальный формат. Для `'e'`, `'inf'` и `'nan'` используется регистр, в котором указан форматный тип
`'n'`	Подобен общему числовому формату, но использует разделители групп разрядов и символ десятичной точки, специфические для локали, если системная локаль установлена
`'%'`	Процентный формат — умножает значение на `100` и форматирует его как число с фиксированной точностью, за которым следует символ `'%'`

Если форматный тип не задан, то для чисел с плавающей точкой используется тип `'g'` с хотя бы одной цифрой после десятичной точки и точностью по умолчанию, равной `12`.

```
>>> n = [3.1415, -42, 1024.0]
>>> for num in n:
...     '{:>+9,.2f}'.format(num)
...
   ' +3.14'
```

```
'   -42.00'
'+1,024.00'
```

Вложенные спецификации формата

В некоторых случаях вам будет удобно включить в вызов метода `format` аргумент, облегчающий определение точного формата для другого аргумента. Эта цель достигается с помощью вложенного форматирования. Например, для того чтобы форматировать строку в поле, ширина которого на четыре символа превышает длину самой строки, методу `format` можно передать значение ширины в качестве аргумента.

```
>>> s = 'a string'
>>> '{0:>{1}s}'.format(s, len(s)+4)
'    a string'
>>> '{0:_^{1}s}'.format(s, len(s)+4)
'__a string__'
```

Тщательно все продумав, вы можете использовать спецификацию ширины и вложенное форматирование для вывода последовательности кортежей в виде аккуратно выровненных столбцов.

```
def print_strings_in_columns(seq_of_strings, widths):
    for cols in seq_of_strings:
        row = ['{c:{w}.{w}s}'.format(c=c, w=w)
               for c, w in zip(cols, widths)]
        print(' '.join(row))
```

(В версии 3.6 выражение `'{c:{w}.{w}s}'.format(c=c, w=w)` можно упростить до `f'{c:{w}.{w}s}'`, о чем говорится в разделе "Новое в версии 3.6: форматированные строковые литералы".) Тогда код

```
c = [
        'four score and'.split(),
        'seven years ago'.split(),
        'our forefathers brought'.split(),
        'forth on this'.split(),
    ]
print_strings_in_columns(c, (8, 8, 8))
```

в котором используется эта функция, выведет следующую информацию:

```
four     score    and
seven    years    ago
our      forefath brought
forth    on       this
```

Форматирование пользовательских классов

В конечном счете значения форматируются посредством вызова их метода `__format__` с передачей ему спецификации формата в качестве аргумента. Каждый

встроенный тип либо реализует свой собственный метод, либо наследует его от объекта `object`, метод `format` которого принимает в качестве аргумента только пустую строку.

```
>>> object().__format__('')
'<object object at 0x110045070>'
>>> 3.141592653589793.__format__('18.6')
'           3.14159'
```

Используя эти знания, вы, если захотите, сможете реализовать собственный, совершенно другой мини-язык форматирования. Приведенный ниже простой пример демонстрирует передачу спецификаций формата и возврат (постоянного) результата в составе отформатированной строки. Поскольку спецификация формата выбирается произвольно, вы вправе реализовать любую нотацию форматирования, которую захотите.

```
>>> class S(object):
...     def __format__(self, fstr):
...         print('Format string:', fstr)
...         return '42'
...
>>> my_s = S()
>>> '{:format_string}'.format(my_s)
Format string: format string
'42'
>>> 'The formatted value is: {:anything you like}'.format(my_s)
Format string: anything you like
'The formatted value is: 42'
```

Значение, возвращаемое методом `__format__`, вставляется в качестве поля замены в строку, вызывающую метод `format`, тем самым обеспечивая возможность любой интерпретации строки формата.

Чтобы облегчить управление форматированием объектов, модуль `string` предоставляет класс `Formatter` с полезными методами, предназначенными для решения многих задач форматирования. Более подробную информацию о классе `Formatter` можно получить в онлайн-документации (`https://docs.python.org/3.5/library/string.html#string.Formatter`).

Новое в версии 3.6: форматированные строковые литералы

Это новое средство упрощает использование описанных выше возможностей форматирования. В нем применяется тот же синтаксис форматирования, но разрешается указывать значения в виде встроенных выражений, а не через подстановки. Вместо спецификаторов аргументов в *f-строках* используются выражения, вычисляемые и форматируемые в соответствии с определениями. Например, вместо команд

```
>>> name = 'Dawn'
>>> print('{name!r} is {l} characters long'
          .format(name=name, l=len(name)))
'Dawn' is 4 characters long
```

начиная с версии 3.6 и далее можно использовать более лаконичную форму:

```
>>> print(f'{name!r} is {len(name)} characters long')
'Dawn' is 4 characters long
```

Для определения компонентов выражений форматирования можно использовать вложенные фигурные скобки.

```
>>> for width in 8, 11:
...     for precision in 2, 3, 4, 5:
...         print(f'{3.14159:{width}.{precision}}')
...
     3.1
    3.14
   3.142
  3.1416
       3.1
      3.14
     3.142
    3.1416
```

Помните, однако, что эти строковые литералы не являются константами — они вычисляются всякий раз, когда выполняется содержащее их выражение, что может приводить к увеличению накладных расходов времени выполнения.

Традиционное форматирование строк с помощью оператора %

Традиционная форма выражения форматирования строк в Python имеет следующий синтаксис:

формат `%` *значения*

Здесь *формат* — строка, содержащая спецификаторы формата, а *значения* — значения, подлежащие форматированию, обычно в виде кортежа. (В этой книге мы обсуждаем только подмножество традиционных средств форматирования — спецификатор формата, который вы должны знать для того, чтобы использовать модуль `logging`; см. раздел "Пакет `logging`" в главе 5).

В случае использования модуля `logging` эквивалент этого выражения мог бы выглядеть следующим образом:

`logging.info(`*формат*`, *`*значения`*`)`

Значения играют роль позиционных аргументов, следующих за первым аргументом (*формат*).

Традиционный способ форматирования строк предлагает в целом те же возможности, которые предлагаются функцией `printf` в языке C, и работает аналогичным образом. Каждый спецификатор формата представляет собой подстроку, которая начинается со знака процента (`%`) и заканчивается одним из символов преобразования, приведенных в табл. 8.6.

Таблица 8.6. Символы преобразований, используемые при форматировании строк

Символ	Выходной формат	Примечания
`d, i`	Десятичное число со знаком	Значение должно быть числом
`u`	Десятичное число без знака	Значение должно быть числом
`o`	Восьмеричное число без знака	Значение должно быть числом
`x`	Шестнадцатеричное число без знака (буквы в нижнем регистре)	Значение должно быть числом
`X`	Шестнадцатеричное число без знака (буквы в верхнем регистре)	Значение должно быть числом
`e`	Значение с плавающей точкой в экспоненциальной форме (нижний регистр буквы `e`, обозначающей экспоненту)	Значение должно быть числом
`E`	Значение с плавающей точкой в экспоненциальной форме (верхний регистр буквы `e`, обозначающей экспоненту)	Значение должно быть числом
`f, F`	Число с плавающей точкой в десятичном представлении	Значение должно быть числом
`g, G`	Аналогичен форматам `e` и `E`, если *exp* >=4 или меньше точности; в противном случае аналогичен форматам `f` и `F`	*exp* — это показатель степени, в которую возводится преобразуемое число
`c`	Одиночный символ	Значение должно быть целым числом или односимвольной строкой
`r`	Строка	Преобразует любое значение с помощью метода `repr`
`e`	Строка	Преобразует любое значение с помощью метода `str`
`%`	Литеральный символ `%`	Не потребляет значений

Чаще всего с модулем `logging` используются символы преобразования `r`, `s` и `%`. Между знаком `%` и символом преобразования можно указать количество необязательных модификаторов, которые вскоре будут обсуждаться.

Что регистрируется в журнале средствами модуля `logging` — так это строка *формат*, в которой каждый спецификатор формата заменяется соответствующим элементом *значения*, преобразованным в строку в соответствии со спецификатором. Вот несколько простых примеров.

```
import logging
logging.getLogger().setLevel(logging.INFO)
x = 42
y = 3.14
z = 'george'
logging.info('result = %d', x)          # вывод: result = 42
logging.info('answers: %d %f', x, y)    # вывод: answers: 42 3.140000
logging.info('hello %s', z)             # вывод: hello george
```

Синтаксис спецификатора формата

Спецификатор формата может включать модификаторы, управляющие способом преобразования соответствующего элемента *значения* в строку. Спецификатор формата состоит из следующих компонентов, перечисленных ниже в порядке их указания.

Обязательный начальный символ `%`, которым отмечается начало спецификатора.

Нуль или более необязательных флагов преобразования.

`#`
: Преобразование использует альтернативную форму (если она существует для его типа).

`0`
: Преобразованное значение дополняется нулями.

`-`
: Преобразованное значение выравнивается влево.

Пробел
: Перед положительными числами вставляется пробел.

`+`
: Перед любым преобразованным числом вставляется соответствующий знак (`+` или `-`).

Необязательный указатель минимальной ширины: одна или несколько цифр или символ "звездочка" (`*`), который означает, что значение ширины берется из следующего элемента *значения*.

Необязательный указатель точности представления: точка (`.`), за которой следует нуль или более цифр или символ "звездочка" (`*`), который означает, что значение ширины берется из следующего элемента *значения*.

Обязательный тип преобразования (см. табл. 8.6).

Каждый спецификатор формата сопоставляется с отдельным значением в соответствии с их позициями, поэтому количество значений в аргументе *значения* должно в точности совпадать с количеством спецификаторов в аргументе *формат* (плюс по одному значению, плюс по одному дополнительному значению для каждого значения ширины или точности, предоставленного символом `*`). Если ширина или точность задается символом `*`, то этот символ потребляет из аргумента *значения* один

элемент, который должен быть целым числом, и это число должно определять количество символов, задающее ширину или точность для данного преобразования.

Когда следует использовать спецификатор формата %r

В большинстве случаев в ваших строках *формат* будет использоваться только преобразование вида %s. Лишь время от времени, если, скажем, понадобится задать определенный тип выравнивания или определенную точность, в них будут появляться спецификаторы формата, содержащие модификаторы, наподобие %9.6s. Однако существует один важный специальный случай, когда вам стоит серьезно подумать об использовании вместо них спецификаторов %r.

Протоколируя потенциальные сообщения об ошибках, всегда используйте спецификатор формата %r

Если вы ожидаете, что среди протоколируемых строковых значений могут быть сообщения об ошибках (таких, например, как отсутствие файла с указанным именем), то в этих случаях использование спецификатора %s сопряжено с определенными рисками: если ошибка связана с тем, что строка содержит лишние начальные или замыкающие пробелы или непечатаемые символы наподобие \b, то заметить такую ошибку, просматривая журнал сообщений, сформированный с использованием спецификатора %s, будет трудно. В подобных ситуациях целесообразно использовать спецификатор формата %r, обеспечивающий отчетливое отображение всех символов.

Функции wrap и fill модуля textwrap

Модуль textwrap предоставляет класс и несколько функций, предназначенных для форматирования строки посредством ее разбиения на строки, длина которых не превышает заданной величины. Если у вас возникнет потребность в тонкой настройке заполнения строк текстом, можете создать экземпляр класса TextWrapper из модуля textwrap, который предоставит вам все необходимые для этого вспомогательные функции. Однако в большинстве случаев вам будет вполне достаточно одной из двух основных функций, предлагаемых этим классом, описание которых приводится ниже.

wrap wrap(*s*, *width*=70)

Возвращает список строк (без завершающих символов перехода на новую строку), длина каждой из которых не превышает *width* символов и которые (будучи воссоединены с помощью пробелов) вновь дают строку *s*. Функция wrap также поддерживает другие именованные аргументы (эквивалентные атрибутам экземпляров класса TextWrapper). Более сложные примеры применения этих функций вы найдете в онлайн-документации (https://docs.python.org/2/library/textwrap.html).

fill `fill(s, width=70)`

Возвращает одну многострочную строку, эквивалентную `'\n'.join(wrap(s, width))`.

Модуль pprint

Модуль `pprint` обеспечивает красивую печать сложных структур данных, применяя форматирование, направленное на то, чтобы улучшить читаемость текста по сравнению с тем, чего можно добиться, используя встроенную функцию `repr` (см. табл. 7.2). Если у вас возникнет потребность в тонкой настройке параметров форматирования, создайте экземпляр класса `PrettyPrinterclass` из модуля `pprint`, который предоставит вам все необходимые для этого вспомогательные функции. Однако в большинстве случаев вам будет вполне достаточно использовать одну из двух основных функций, предлагаемых этим классом, описание которых приводится ниже.

pformat `pformat(obj)`

Возвращает строку, представляющую красивую печать объекта *obj*.

pprint `pprint(obj, stream=sys.stdout)`

Обеспечивает красивую печать объекта *obj* в поток открытого для записи файла с использованием завершающего символа перевода строки.

То же самое обеспечивают следующие инструкции:

```
print(pprint.pformat(x))
pprint.pprint(x)
```

Во многих случаях, например когда строковое представление *x* умещается в одной строке, любая из этих конструкций в целом эквивалентна функции `print(x)`. Однако в случае таких, например, объектов, как `x=list(range(30))`, вызов `print(x)` отображает *x* на двух строках, разрывая первую строку в произвольной точке, тогда как модуль `pprint` отобразит *x* на 30 строках, по одной на каждый элемент. Вы можете использовать модуль `pprint` во всех случаях, в которых считаете получаемые с его помощью результаты более предпочтительными по сравнению с обычным строковым представлением объектов.

Модуль reprlib

Модуль `reprlib` (`repr` в версии v2) содержит альтернативную версию встроенной функции `repr` (см. табл. 7.2), позволяющую устанавливать ограничения на длину строкового представления объектов. Если у вас возникнет потребность в тонкой настройке длины выводимого строкового представления, можете создать экземпляр класса `Repr`, предоставляемого данным модулем, или его подкласса, который снабдит вас всеми необходимыми для этого вспомогательными функциями. Однако в большинстве случаев вам будет вполне достаточно одной функции `repr`, предлагаемой данным модулем, описание которой приводится ниже.

repr repr(*obj*)

Возвращает строку, представляющую объект *obj*, одновременно налагая разумные ограничения на объем выводимой информации.

Unicode

В версии v2 для преобразования байтовых строк в строки Unicode используется встроенная функция unicode или метод decode байтовых строк. Кроме того, такое преобразование может выполняться неявно при передаче байтовых строк функциям, ожидающим аргументы в виде строк Unicode. В версии v3 это преобразование всегда должно выполняться явным образом с помощью метода decode байтовых строк.

В любом случае такое преобразование выполняет вспомогательный объект, известный под названием *кодек* (сокр. от *кодировщик-декодировщик*). Кодек также преобразует строки Unicode в байтовые строки, как явно с помощью метода encode строк Unicode, так и (только в версии v2) неявно.

Чтобы идентифицировать кодек, следует передать его имя методу unicode, decode или encode. Если имя кодека не передается или, в случае версии v2, преобразование выполняется неявно, Python использует декодирование, заданное по умолчанию, обычно посредством метода 'ascii' в версии v2 или 'utf8' в версии v3.

Каждое преобразование имеет параметр errors — строку, определяющую способ обработки ошибок, которые возникают в процессе преобразования. Значение по умолчанию, 'strict', означает возбуждение исключений для любых ошибок. Значению 'replace' соответствует замена каждого ошибочного символа символом '?' в результирующей байтовой строке или символом u'\ufffd' в результирующей строке Unicode. Значению 'ignore' соответствует пропуск ошибочных символов без возбуждения исключений. Значению 'xmlcharrefreplace' соответствует замена каждого ошибочного символа ссылкой на символ XML. Вы можете написать собственную функцию, реализующую другую стратегию обработки ошибок, и зарегистрировать ее под соответствующим именем, вызвав метод codecs.register_error, описанный в табл. 8.7.

Модуль codecs

Сопоставлением имен кодеков с объектами кодеков управляет модуль codecs, который также предоставляет возможность разрабатывать собственные объекты кодека и регистрировать их, чтобы их можно было находить по имени, как любой встроенный кодек. Кроме того, модуль codecs позволяет вам самостоятельно находить любой кодек, получая функции, которые кодек использует для кодирования и декодирования строк, а также функции-фабрики, играющие роль оберток вокруг файловых объектов. Необходимость в использовании всех этих продвинутых возможностей модуля codecs возникает крайне редко, и в этой книге мы не будем на них останавливаться.

Вместе с пакетом encodings стандартной библиотеки Python модуль codecs предоставляет встроенные кодеки, полезные для разработчиков на языке Python, которым приходится иметь дело с интернационализацией программ. В поставку Python входит свыше 100 кодеков; их список вместе с кратким описанием каждого из них можно найти в онлайн-документации (https://docs.python.org/3/library/codecs.html#standard-encodings). С технической точки зрения модуль sitecustomize позволяет установить любой из поставляемых кодеков в качестве кодека, используемого по умолчанию всем сайтом, но так поступать не рекомендуется, и предпочтительный вариант заключается в том, чтобы указывать кодеки по имени всякий раз, когда выполняется преобразование между байтовыми строками и строками Unicode. В версии v2 в качестве кодека по умолчанию установлен кодек 'ascii', который принимает символы с кодами в диапазоне 0–127, т.е. 7-битовом диапазоне таблицы кодировки ASCII, общем почти для всех кодировок. В западноевропейских странах популярен кодек 'latin-1' — быстрая встроенная реализация стандарта кодирования ISO 8859-1, которая предлагает кодирование в виде одного байта на каждый символ из числа существующих в западноевропейских языках (обратите внимание на то, что символ '€', обозначающий валюту "евро", отсутствует в 'latin-1'; если он вам нужен, используйте кодек 'iso8859-15').

Кроме того, модуль codecs предоставляет кодеки, которые реализованы в Python для большинства кодировок, соответствующих стандарту ISO 8859, с именами от 'iso8859-1' до 'iso8859-15'. Например, если вы применяете кодировку ASCII плюс некоторые буквы греческого алфавита, распространенные в научных статьях, можете воспользоваться кодеком 'iso8859-7'. Если речь идет о системах Windows, то процедуры преобразования многобайтового символьного набора, используемого на этой платформе, упакованы в кодеке 'mbcs'. Многие кодеки специально предназначены для поддержки азиатских языков. Кроме того, модуль codecs предоставляет несколько стандартных кодовых страниц (кодеки с именами от 'cp037' до 'cp1258'), старые кодировки, специфические для Mac (кодеки с именами от 'mac-cyrillic' до 'mac-turkish'), а также стандартные кодировки Unicode 'utf-8' (вероятно, наилучший выбор в большинстве случаев, а потому рекомендуемый и используемый по умолчанию в версии v3) и 'utf-16' (для последнего также предусмотрены варианты с порядком байтов от старшего к младшему и от младшего к старшему: 'utf-16-be' и 'utf-16-le'). Кроме того, для использования с кодировкой UTF-16 кодеки предоставляют атрибуты BOM_BE и BOM_LE, реализующие метки порядка байтов от старшего к младшему и от младшего к старшему соответственно, а также атрибут BOM, реализующий метку порядка байтов для текущей платформы.

Модуль codecs также предоставляет функцию, позволяющую регистрировать собственные функции обработки ошибок (табл. 8.7).

Таблица 8.7. Функция `register_error()`

`register_error`	`register_error(`*`name`*`, `*`func`*`)` Аргумент *`name`* должен быть строкой. Аргумент *`func`* должен быть вызываемым объектом с одним аргументом *`e`*, представляющим экземпляр исключения `UnicodeDecodeError` Указанный объект должен возвращать кортеж, включающий два элемента: строку Unicode, подлежащую вставке в результирующую преобразованную строку, и индекс, начиная с которого следует продолжить преобразование (обычно в качестве последнего используется индекс `e.end`). Тело функции может использовать строку `e.encoding` с именем кодека, выполняющего данное преобразование, и `e.object[e.start:e.end]` — подстроку, ставшую причиной возникновения ошибки

Кроме того, модуль `codecs` предоставляет функцию, облегчающую работу с файлами кодированного текста.

`EncodedFile` `EncodedFile(`*`file`*`, `*`datacodec`*`, `*`filecodec`*`=None, `*`errors`*`='strict')`

Служит оболочкой файлового объекта *`file`*, возвращающего файловый объект *`ef`*, который обеспечивает неявное прозрачное применение заданной кодировки ко всем данным, считываемым или записываемым в файл. Если вы записываете байтовую строку *`s`* в *`ef`*, то *`ef`* сначала декодирует *`s`* с помощью кодека *`datacodec`*, а затем кодирует результат с помощью кодека *`filecodec`* и записывает его в *`file`*. Если вы читаете строку, то *`ef`* применяет сначала *`filecodec`*, а затем *`datacodec`*. Если для *`filecodec`* указано значение `None`, то *`ef`* использует *`datacodec`* для преобразований в обоих направлениях.

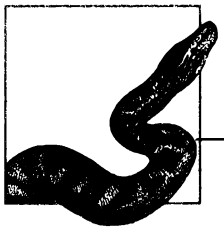
Например, если вы хотите записать байтовые строки с кодировкой `latin-1` в `sys.stdout` и выходные строки должны иметь кодировку `utf-8`, то в версии v2 необходимо использовать следующий код:

```
import sys, codecs
sys.stdout = codecs.EncodedFile(sys.stdout,
'latin-1','utf-8')
```

Обычно вам будет достаточно использовать вместо этого вызов `io.open` (раздел "Создание объекта "файла" с помощью метода `io.open`" в главе 10).

Модуль `unicodedata`

Модуль `unicodedata` обеспечивает легкий доступ к базе данных символов Unicode (`http://unicode.org/ucd/`). Используя предоставляемые модулем `unicodedata` функции, вы сможете не только узнать категорию и официальное название (если таковое имеется) любого символа Unicode, но и получить массу других, более экзотических сведений о нем. Вы также сможете найти символ Unicode, если вам известно его название. Необходимость в подобных средствах возникает лишь в редких случаях, и в этой книге они не рассматриваются.



9 Регулярные выражения

Регулярные выражения позволяют создавать строки-шаблоны, которые можно использовать для поиска и замены текста. Овладеть регулярными выражениями нелегко, но они послужат вам очень мощным подспорьем при обработке текстовых данных. Богатая функциональность регулярных выражений, предлагаемая в Python, реализована во встроенном модуле `re`.

Регулярные выражения и модуль `re`

Регулярное выражение (РВ) конструируется на основе строки, представляющей шаблон. Функциональность регулярных выражений позволяет сравнивать с шаблоном любую строку и находить те ее части, которые соответствуют данному шаблону.

В Python функциональность регулярных выражений предоставляет модуль `re`. Функция `compile` создает объект регулярного выражения на основе строки шаблона и необязательных флагов. Методы этого объекта обеспечивают поиск соответствий данному регулярному выражению в строке или выполнение необходимых подстановок. Кроме того, в модуле `re` содержатся функции, эквивалентные методам объекта регулярного выражения, но использующие строку шаблона РВ в качестве первого аргумента.

Овладеть регулярными выражениями нелегко, и они не являются основной темой данной книги, поэтому ниже мы рассмотрим лишь способы их применения в Python. Для общего ознакомления с регулярными выражениями рекомендуем прочитать книгу Джеффри Фридла *Регулярные выражения*, где регулярные выражения обсуждаются не только на учебном, но и на более продвинутом уровне. Множество ссылок на руководства и справочники по регулярным выражениям можно найти в Интернете. В частности, неплохое подробное руководство по регулярным выражениям содержится в онлайновой документации Python. Сайты наподобие *Pythex* (`pythex.org`) и *regex101* (`regex101.com`) предоставляют возможность тестировать регулярные выражения в интерактивном режиме.

Использование модуля `re` при работе с байтовыми строками и строками Unicode

По умолчанию регулярные выражения в версии v3 работают двояким образом. В случае экземпляров `str` РВ сопоставляются с символами Unicode (например, символ Unicode *c* считается "буквой", если выражение `'LETTER' in unicodedata.name(c)` истинно). В случае экземпляров `bytes` РВ сопоставляются с символами ASCII (например, байтовый символ *c* считается "буквой", если выражение `c in string.ascii_letters` истинно). В версии v2 регулярные выражения по умолчанию сопоставляются с символами ASCII независимо от того, применяются ли они к экземплярам `str` или `unicode`. Это поведение можно изменить с помощью необязательных флагов, таких как `re.U` (раздел "Флаги опций"). Например, в версии v3 это делается так.

```
import re
print(re.findall(r'\w+', 'cittá'))
# вывод: ['cittá']
```

А вот пример в версии v2.

```
import re
print(re.findall(r'\w+', u'cittá'))
# вывод: ['citt']
print(re.findall(r'\w+', u'cittá', re.U))
# вывод: ['citt\xe1']
```

Синтаксис строковых шаблонов

Строковые шаблоны, представляющие регулярные выражения, подчиняются следующему синтаксису.

- Буквенные и цифровые символы означают сами себя. Регулярному выражению, шаблоном которого является строка, состоящая из букв и цифр, соответствует такая же строка.
- Многие алфавитно-цифровые символы приобретают специальный смысл в шаблоне, если им предшествует символ обратной косой черты (`\`).
- Знаки пунктуации работают совершенно иначе: они совпадают сами с собой, если экранированы символом обратной косой черты в шаблоне, и имеют специальный смысл, если не экранированы.
- Символ обратной косой черты соответствует шаблону `\\`.

Поскольку символы обратной косой черты встречаются в шаблонах РВ очень часто, лучше всегда указывать их с использованием синтаксиса "сырых" строк (см. раздел "Строки" в главе 3). Такие элементы шаблона, как, например, `r'\t'` (эквивалент обычного строкового литерала `'\\t'`), совпадают с соответствующими специальными символами (в данном случае с символом табуляции `'\t'`, или `chr(9)`), поэтому

синтаксис "сырых" строк можно использовать даже в тех случаях, когда требуется литеральное сопоставление с такими символами.

Специальные элементы синтаксиса шаблонов РВ приведены в табл. 9.1. Точный смысл некоторых из них может изменяться, если при создании объектов РВ на основе шаблонов используются необязательные флаги. Необязательные флаги рассмотрены в разделе "Флаги опций".

Таблица 9.1. Синтаксис шаблонов регулярных выражений

Элемент	Что означает в регулярном выражении
`.`	Любой символ, за исключением символа `\n` (в режиме `DOTALL` совпадает также с `\n`)
`^`	Начало строки (в режиме `MULTILINE` совпадает также с позицией, непосредственно следующей за символом `\n`)
`$`	Конец строки (в режиме `MULTILINE` совпадает также с позицией, непосредственно предшествующей символу `\n`)
`*`	Повторение предыдущего РВ нуль или более раз; жадный поиск (пытается захватить как можно бо́льшую часть строки)
`+`	Повторение предыдущего РВ один или более раз; жадный поиск (пытается захватить как можно бо́льшую часть строки)
`?`	Повторение предыдущего РВ нуль или один раз; жадный поиск (пытается захватить один символ, если это возможно)
`*?`, `+?`, `??`	Нежадные версии `*`, `+` и `?` соответственно (пытается захватить как можно меньшую часть строки)
`{`*m*`,`*n*`}`	Повторение предыдущего РВ от *m* до *n* раз (жадный поиск)
`{`*m*`,`*n*`}?`	Повторение предыдущего РВ от *m* до *n* раз (нежадный поиск)
`[...]`	Любой символ из набора в скобках
`[^...]`	Любой символ, не встречающийся в наборе в скобках
`|`	РВ слева или РВ справа
`(...)`	РВ в скобках; круглые скобки выделяют группу
`(?iLmsux)`	Альтернативный способ задания необязательных флагов; не оказывает влияния на поиск совпадений*
`(?:...)`	Как `(...)`, но без выделения группы
`(?P<`*id*`>...)`	Как `(...)`, но группа получает имя *id*
`(?P=`*id*`>...)`	Точно соответствует ранее выделенной группе с именем *id*
`(?#...)`	Содержимое скобок является просто комментарием; не оказывает влияния на поиск совпадений
`(?=...)`	*Опережающая проверка*: следующая часть строки должна соответствовать РВ `...`, но дальнейшее сопоставление с шаблоном начинается с того же места

Элемент	Что означает в регулярном выражении
`(!:...)`	*Отрицательная опережающая проверка*: следующая часть строки не должна соответствовать РВ `...`, и дальнейшее сопоставление с шаблоном начинается с того же места
`(?<=...)`	*Ретроспективная проверка*: предыдущая часть строки, заканчивающаяся в текущей позиции, должна соответствовать РВ `...` (РВ `...` должно иметь фиксированную длину, т.е. не должно содержать элементы + и *)
`(?<!...)`	*Отрицательная ретроспективная проверка*: предыдущая часть строки, заканчивающаяся в текущей позиции, не должна соответствовать РВ `...` (РВ `...` должно иметь фиксированную длину, т.е. не должно содержать элементы + и *)
`\`*число*	Совпадает со строкой, соответствующей ранее выделенной группе с указанным номером (группам автоматически присваиваются номера от 1 до 99 в направлении слева направо)
`\A`	Совпадает с пустой строкой, но только в начале всей строки (т.е. с самой позицией, а не символом, перед началом строки)
`\b`	Совпадает с пустой строкой, но только в начале или в конце слова (слитная последовательность алфавитно-цифровых символов максимально возможной длины; см. также `\w`)
`\B`	Совпадает с пустой строкой, но не в начале или в конце слова
`\d`	Одна цифра, аналогично набору `[0-9]` (в режиме Unicode многие другие символы также считаются "цифрами" с точки зрения `\d`, но не `[0-9]`)
`\D`	Один символ, не являющийся цифрой, аналогично набору `[^0-9]` (в режиме Unicode многие другие символы также считаются "цифрами" с точки зрения `\D`, но не `[^0-9]`)
`\s`	Любой пробельный символ; аналогично набору `[\t\n\r\f\v]`
`\S`	Любой символ, не являющийся пробельным; аналогично набору `[\t\n\r\f\v]`
`\w`	Любой алфавитно-цифровой символ; аналогично `[a-zA-Z0-9]` (только если не используется Unicode или не установлен флаг `LOCALE` или `UNICODE`)
`\W`	Любой символ, не являющийся алфавитно-цифровым; обратен `\w`
`\Z`	Пустая строка, но только в начале или в конце всей строки
`\\`	Символ обратной косой черты

* Используя конструкцию `(?...)`, помещайте ее в начале шаблона для улучшения его читаемости. Начиная с версии 3.6, если эта конструкция располагается в другом месте, выводится предупреждающее сообщение `DeprecationWarning`.

Распространенные идиомы регулярных выражений

Всегда используйте синтаксис `r'...'` в литеральных шаблонах РВ

Используйте синтаксис "сырых" строк во всех литеральных шаблонах РВ и только в них: это избавляет от необходимости экранировать обратную косую черту (`\`) и улучшает читаемость кода, поскольку позволяет сразу увидеть литеральный шаблон РВ.

Пара символов `.*`, используемая в качестве подстроки шаблона регулярного выражения, означает "любое количество повторений (нуль или более раз) любого символа". Другими словами, символы `.*` совпадают с любой подстрокой целевой строки, включая пустую подстроку. Пара символов `.+` играет аналогичную роль, но ей соответствует лишь непустая подстрока. Например, шаблону

```
r'pre.*post'
```

соответствует строка, содержащая подстроку `'pre'`, за которой далее следует подстрока `'post'`, причем последняя подстрока может примыкать к первой (т.е. этому шаблону соответствует как строка `'prepost'`, так и строка `'pre23post'`). С другой стороны, шаблону

```
r'pre.+post'
```

соответствуют только строки, в которых подстроки `'pre'` и `'post'` не являются смежными (например, ему соответствует строка `'pre23post'`, но не соответствует строка `'prepost'`). Кроме того, обоим шаблонам удовлетворяют строки, имеющие продолжение после подстроки `'post'`. Чтобы шаблону соответствовали только строки, заканчивающиеся подстрокой `'post'`, следует завершить приведенный шаблон символами `\Z`. Например, шаблону

```
r'pre.*post\Z'
```

соответствует строка `'prepost'`, но не соответствует строка `'preposterous'`.

Во всех приведенных выше примерах шаблон выполнял так называемый *жадный* поиск совпадений, т.е. захватывал подстроку, начинающуюся с первого вхождения строки `'pre'` и заканчивающуюся последним вхождением строки `'post'`. Если для вас имеет значение, какую именно часть целевой строки захватывает регулярное выражение, то вы, возможно, захотите применить нежадный поиск совпадений, который ограничится захватом подстроки, начинающейся с первого вхождения строки `'pre'`, но заканчивающейся первым найденным вхождением строки `'post'`.

Например, в случае строки `'preposterous and post facto'` жадному шаблону РВ `r'pre.*post'` соответствует подстрока `'preposterous and post'`, тогда как его нежадному варианту `r'pre.*?post'` соответствует подстрока `'prepost'`.

К числу других часто используемых элементов шаблонов РВ относится `\b`, которому соответствует граница слова. Чтобы найти сочетание `'his'` в виде самостоятельного

слова, а не в виде подстроки в таких словах, как 'this' и 'history', необходимо использовать шаблон с границами слова в начале и в конце:

```
r'\bhis\b'
```

Чтобы найти начало любого слова, начинающегося со строки 'her', например самого слова 'her' или слова 'hermetic', но не начало таких слов, лишь содержащих подстроку 'her' в других местах, как 'ether' или 'there', используйте шаблон с границей слова перед соответствующей строкой, но не после нее:

```
r'\bher'
```

Чтобы найти конец любого слова, заканчивающегося подстрокой 'its', например самого слова 'its' или слова 'fits', но не конец таких слов, лишь содержащих подстроку its' в других местах, как 'itsy' или 'jujitsu', используйте шаблон с границей слова после соответствующей строки, но не перед ней:

```
r'its\b'
```

Чтобы найти целые слова с указанными ограничениями, а не просто их начало или конец, добавьте элемент шаблона \w*, которому соответствуют нуль или более символов слова. Чтобы найти любое полное слово, начинающееся с подстроки 'her', используйте такой шаблон:

```
r'\bher\w*'
```

Чтобы сопоставлялись лишь первые три буквы любого слова, начинающегося с подстроки 'her', но не самого слова 'her', используйте отрицание границы слова — элемент \B:

```
r'\bher\B'
```

Чтобы найти любое целое слово, заканчивающееся подстрокой 'its', в том числе и само слово 'its', используйте такой шаблон:

```
r'\w*its\b'
```

Наборы символов

Последовательность символов шаблона, заключенная в квадратные скобки ([]), называется *набором*. Набору соответствует любой одиночный символ из числа тех, которые в нем указаны. Наборы могут включать диапазоны символов, обозначаемые дефисом (-), который помещается между границами диапазона. В отличие от других диапазонов Python, последний символ диапазона набора включается в диапазон. В пределах набора специальные символы интерпретируются как литералы, т.е. означают сами себя. Исключение составляют символы \,] и -, которые требуют экранирования (посредством символа обратной косой черты) в тех случаях, когда без экранирования они интерпретировались бы как часть синтаксиса набора. Кроме того, наборы могут включать *классы символов*, обозначаемые экранированными буквами, такие как \d или \S. В наборе последовательность символов \b означает символ возврата на один шаг, или

символ забоя (`chr(8)`), а не границу слова. Символ "крышка" (`^`), следующий сразу же за открывающей скобкой набора, изменяет смысл набора на противоположный: такому набору соответствует любой одиночный символ, *не* входящий в набор.

Символьные наборы часто применяют для поиска слов, которые могут включать символы, не входящие в набор, принятый по умолчанию для элемента `\w` (только буквы и цифры). Например, для поиска слов, состоящих из одного или нескольких символов, каждый из которых может быть буквой, апострофом или дефисом, но не цифрой (как в слове "Finnegan-O'Hara"), следует использовать такой шаблон:

```
r"[a-zA-Z'\-]+"
```

Экранируйте дефисы, являющиеся частью символьного набора, для улучшения читаемости PB

Строго говоря, в данном случае символ дефиса можно было бы не экранировать, поскольку его расположение в конце набора делает ситуацию однозначной. И все же лучше его экранировать, поскольку символ обратной косой черты отчетливо укажет на то, что вы включаете экранированный символ дефиса в набор, и отличит его от аналогичного символа, разделяющего границы диапазона.

Альтернативы

Вертикальная черта (`|`) в шаблоне регулярного выражения, используемая для указания альтернативных вариантов, имеет низкий синтаксический приоритет. В отсутствие круглых скобок, изменяющих группирование, элемент `|` применяется ко всему шаблону по обе его стороны вплоть до начала или конца шаблона или до следующего элемента `|`. Шаблон может состоять из произвольного количества подшаблонов, объединенных элементами `|`. При поиске соответствий такому шаблону PB сначала проверяется первый подшаблон, и если для него находится соответствие, то остальные подшаблоны не используются. Если же найти соответствие для первого подшаблона не удается, то в работу включается второй подшаблон и т.д. Элемент `|` не является ни жадным, ни нежадным: он не учитывает длину совпадения.

Если задан список `L`, состоящий из слов, то шаблон PB, который будет находить любое из этих слов, можно сформировать следующим образом:

```
'|'.join(r'\b{}\b'.format(word) for word in L)
```

Экранирование строк

Если бы элементами списка `L` могли быть более общие строки, а не просто слова, то вам понадобилось бы экранировать их с помощью функции `re.escape` (см. описание в табл. 9.3) и, возможно, не использовать маркеры границ слов `\b` с двух сторон. В этом случае шаблон PB имел бы следующий вид:

```
'|'.join(re.escape(s) for s in L)
```

Группы

Регулярное выражение может содержать любое количество групп, от нуля до 99 (их может быть больше, но полностью поддерживаются только первые 99 групп). Признаком группы служат круглые скобки. Элемент `(?P<id>...)` также обозначает группу и присваивает ей имя *id*, в качестве которого можно указать любой допустимый идентификатор Python. Все группы, именованные и неименованные, нумеруются слева направо, от 1 до 99; группа 0 — это все регулярное выражение.

При любом сопоставлении РВ со строкой каждой группе соответствует подстрока (возможно, пустая). Если в РВ используется элемент `|`, то для некоторых групп соответствие может отсутствовать, хотя при этом строка будет соответствовать регулярному выражению в целом. Если для группы не находится соответствие, то говорят, что для данной группы совпадения отсутствуют. За исключением случаев, оговоренных далее, для любой такой группы в качестве соответствия используется пустая строка (`''`). Например, следующему регулярному выражению соответствует любая строка, состоящая из двух или более повторений любой непустой подстроки:

```
r'(.+)\1+\Z'
```

Поскольку часть `(.+)` шаблона, которой соответствует любая непустая подстрока (любые символы в количестве не менее одного), заключена в скобки, она определяет группу. Части `\1+` шаблона соответствует одно или несколько повторений данной группы, а элемент `\Z` привязывает соответствие к концу строки.

Флаги опций

Элемент регулярного выражения в виде одной или нескольких букв `iLmsux`, заключенных между элементами `(?` и `)`, позволяет задавать опции в самом шаблоне, а не с помощью аргумента *`flags`* функции `compile` модуля `re`. Опции применяются ко всему регулярному выражению, независимо от того, в каком именно месте шаблона они встречаются.

Чтобы сделать код более понятным, всегда помещайте опции в начале шаблона РВ

В частности, размещение опций в начале шаблона является обязательным, если они включают опцию *x*, поскольку эта опция изменяет способ анализа шаблона, применяемый Python. В версии 3.6, если опции располагаются не в начале шаблона, выводится предупреждение об ошибке `DeprecationWarning`.

Использование явно указываемого аргумента *`flags`* улучшает читаемость кода по сравнению с указанием опций в шаблоне. Аргумент *`flags`* функции `compile` представляет собой целочисленный код, который создается с помощью побитовой операции ИЛИ (выполняемой посредством побитового оператора `|`), применяемой

к одному или нескольким нижеперечисленным атрибутам модуля re. Для удобства их использования каждый атрибут имеет как короткое имя (состоящее из одной буквы в верхнем регистре), так и длинное (идентификатор, состоящий из нескольких букв в верхнем регистре), которое является более предпочтительным, поскольку оно сразу же раскрывает назначение флага.

I *или* IGNORECASE
: При сопоставлении с шаблоном регистр букв не учитывается.

L *или* LOCALE
: Влияет на сопоставление с элементами \w, \W, \b и \B, в зависимости от того, какие символы считаются алфавитно-цифровыми в текущей локали; объявлен устаревшим в версии v3.

M *или* MULTILINE
: Если задана эта опция, то символам ^ и $ соответствуют начало и конец каждой строки (т.е. позиции непосредственно после каждого символа перевода строки или перед ним), а также начало и конец всей строки многострочного текста (элементам \A и \Z всегда соответствуют начало и конец всей строки).

S *или* DOTALL
: Если задана эта опция, то специальному символу . соответствует любой символ, включая символ перевода строки.

U *или* UNICODE
: Влияет на сопоставление с элементами \w, \W, \b и \B, в зависимости от того, какие символы считаются алфавитно-цифровыми в Unicode; объявлен устаревшим в версии v3.

X *или* VERBOSE
: Если задана эта опция, то неэкранированные пробельные символы или пробельные символы, указанные в наборе, игнорируются, а символы # становятся началом комментария, распространяющегося до конца строки.

В качестве примера ниже приведены три способа определения эквивалентных РВ с помощью функции compile (см. табл. 9.3). Каждому из них соответствует слово "hello", записанное с использованием любого сочетания верхнего и нижнего регистров букв.

```
import re
r1 = re.compile(r'(?i)hello')
r2 = re.compile(r'hello', re.I)
r3 = re.compile(r'hello', re.IGNORECASE)
```

Третий подход явно выигрывает в отношении удобочитаемости и поэтому облегчает сопровождение кода, несмотря на то, что требует ввода большего количества текста. В данном случае использовать "сырые" строки необязательно, поскольку

шаблоны не включают символ обратной косой черты. Вместе с тем их использование ничему не повредит, и мы рекомендуем всегда использовать "сырые" строки, чтобы сделать код более понятным и удобочитаемым.

Опция `re.VERBOSE` (или `re.X`) позволяет улучшить читаемость шаблонов за счет использования пробелов и комментариев. Как правило, длинные сложные шаблоны РВ лучше представлять строками, располагающимися на нескольких строках текста, и обычно в таких случаях вам будет удобнее использовать формат "сырых" строк, заключенных в тройные кавычки. Ниже представлен пример регулярного выражения для сопоставления с целочисленными литералами в старом стиле версии v2.

```
repat_num1 = r'(0[0-7]*|0x[\da-fA-F]+|[1-9]\d*)L?\Z'
repat_num2 = r'''(?x)                  # шаблон, соответствующий
                                       # целочисленным литералам
               (0 [0-7]*           |   # восьмеричные: ведущий 0, 0+
                                       # восьмеричных цифр
                0x [\da-fA-F]+ |       # шестнадцатеричные: 0x, затем 1+
                                       # шестнадцатеричных цифр
                [1-9] \d*          )   # десятичные: ведущий не 0, 0+ цифр
                L?\Z                   # необязательное замыкающее L,
                                       # конец строки
                '''
```

В этом примере оба шаблона эквивалентны, но свободное использование пробелов для визуального выделения логических частей шаблона и использование комментариев делает второй вариант более понятным и удобочитаемым.

Сопоставление и поиск

До сих пор мы использовали регулярные выражения для установления соответствия между регулярными выражениями и строками путем их *сопоставления*. Например, регулярному выражению с шаблоном `r'box'` соответствуют такие строки, как `'box'` и `'boxes'`, но строка `'inbox'` ему не соответствует. Другими словами, РВ неявно привязывается к началу целевой строки, как если бы шаблон начинался с элемента `\A`.

Часто вы заинтересованы в нахождении возможных соответствий РВ в любом месте строки, без привязки к ее началу (например, нужно найти соответствие шаблону `r'box'` в таких строках, как `'inbox'`, а также `'box'` и `'boxes'`). Для подобных операций в Python предусмотрен термин *поиск* (search), в отличие от термина *сопоставление* (match). Эти операции требуют использования метода `search` объекта РВ; метод `match` находит соответствия только в начале строки.

```
import re
r1 = re.compile(r'box')
if r1.match('inbox'):
    print('успешное сопоставление')
else:
    print('неудачное сопоставление') # вывод: неудачное сопоставление
```

```
if r1.search('inbox'):
    print('успешный поиск')          # вывод: успешный поиск
else:
    print('неудачный поиск')
```

Привязка к началу и к концу строки

Гарантией того, что выполняемая регулярным выражением операция поиска (или сопоставления) будет привязана к началу или к концу строки, служат элементы `\A` и `\Z` соответственно, известные как *якорные привязки*. Если объект РВ не является многострочным (т.е. не содержит элемент шаблона `(?m)` и не компилируется с флагом `re.M` или `re.MULTILINE`), то вместо элементов `\A` и `\Z` традиционно используют эквивалентные им в данном случае элементы `^` и `$` соответственно. Однако в случае многострочных объектов РВ элемент `^` обеспечивает привязку к началу любой строки (т.е. как к началу всей строки, так и к позиции непосредственно после любого символа новой строки `\n`), а элемент `$` — к концу любой строки (т.е. как к концу всей строки, так и к позиции непосредственно перед любым символом `\n`). С другой стороны, элементы `\A` и `\Z` обеспечивают привязку к началу и к концу строки, независимо от того, является ли объект РВ многострочным или не является. Ниже приведен пример, показывающий, как можно проверить, содержит ли файл строки, заканчивающиеся цифрами.

```
import re
digatend = re.compile(r'\d$', re.MULTILINE)
with open('afile.txt') as f:
    if digatend.search(f.read()):
        print('некоторые строки заканчиваются цифрами')
    else:
        print('строки, заканчивающиеся цифрами, отсутствуют')
```

Шаблон `r'\d\n'` почти эквивалентен приведенному выше, но поиск с его помощью будет неудачным, если самым последним символом в файле является цифра, за которой не следует символ перевода строки. В то же время шаблон, используемый в примере, успешно выполнит поиск как в случае, когда цифра является самым последним символом файла, так и в случае, когда за ней следует символ перевода строки.

Объекты регулярных выражений

Ниже перечислены доступные только для чтения атрибуты объекта *r* регулярного выражения, содержащие информацию о способе его создания (посредством функции `compile` модуля `re`, описанной в табл. 9.3).

`flags`
: Аргумент *`flags`*, переданный функции `compile`, или `0`, если аргумент *`flags`* был опущен (в версии v2; в версии v3 значением по умолчанию, если аргумент *`flags`* был опущен, является `re.UNICODE`).

`groupindex`

Словарь, ключами которого являются имена групп, определенные элементами `(?P<id>...)`; соответствующими значениями являются номера именованных групп.

`pattern`

Строка шаблона, компилированием которой был получен объект `r`.

Используя эти атрибуты, можно очень просто получить из объекта PB его строку шаблона и флаги, что позволяет не хранить их отдельно.

Кроме того, объект *r* регулярного выражения содержит методы, позволяющие определять местонахождение соответствий для *r* в строке, а также выполнять замену найденных соответствий (табл. 9.2). Обычно соответствия представляются отдельными объектами (раздел "Объекты совпадений").

Таблица 9.2. Методы объекта регулярного выражения

Метод	Описание

`findall`	*r*`.findall(`*s*`)`
	Если *r* не имеет групп, то `findall` возвращает список строк, каждая из которых является непересекающейся подстрокой *s*, представляющей соответствие *r*. Например, для вывода на отдельных строках всех слов, содержащихся в файле, можно использовать следующий код:
	`import re`
	`reword = re.compile(r'\w+')`
	`with open('afile.txt') as f:`
	`  for aword in reword.findall(f.read()):`
	`  print(aword)`
	Если *r* содержит одну группу, то `findall` также возвращает список строк, но в этом случае каждая строка является подстрокой *s*, соответствующей указанной группе *r*. Например, для вывода только слов, за которыми следуют пробелы (а не знаки пунктуации), в примере потребуется изменить только одну инструкцию:
	`reword = re.compile('(\w+)\s')`
	Если *r* имеет *n* групп (*n*>1), то `findall` возвращает список кортежей, по одному на каждое непересекающееся соответствие *r*. Например, для вывода только первого и последнего слова каждой строки, включающей по крайней мере два слова, можно использовать следующий код:
	`import re`
	`first_last =`
	`  re.compile(r'^\W*(\w+)\b.*\b(\w+)\W*$',re.MULTILINE)`
	`with open('afile.txt') as f:`
	`  for first, last in first_last.findall(f.read()):`
	`  print(first, last)`

Метод	Описание
finditer	*r*.finditer(*s*) Метод finditer подобен методу findall, но вместо списка строк (или кортежей) он возвращает итератор, элементы которого являются объектами соответствия. Поэтому в большинстве случаев метод finditer оказывается более гибким по сравнению с методом findall и работает быстрее

match	*r*.match(*s*, *start*=0, *end*=sys.maxsize)
	Возвращает подходящий объект соответствия, если подстрока *s*, начинающаяся с индекса *start* и заканчивающаяся не далее индекса *end*, соответствует *r*. В противном случае возвращается значение None. Обратите внимание на то, что сопоставление неявно привязывается к начальной позиции *start* в *s*. Чтобы выполнить поиск соответствия *r* с любой точки от *start* и выше, следует вызвать метод *r*.search, а не *r*.match. Например, ниже представлен один из способов вывода всех строк файла, начинающихся с цифр.
	`import re`
	`digs = re.compile(r'\d')`
	`with open('afile.txt') as f:`
	`  for line in f:`
	`  if digs.match(line):`
	`  print(line, end='')`

search	*r*.search(*s*, *start*=0, *end*=sys.maxsize)
	Возвращает подходящий объект соответствия для крайней слева подстроки *s*, начинающейся не ранее индекса *start* и не выходящей за пределы индекса *end*, которая соответствует *r*. Если такой подстроки не существует, то метод search возвращает значение None. Например, для вывода всех строк, содержащих цифры, можно использовать следующий простой подход:
	`import re`
	`digs = re.compile(r'\d')`
	`with open('afile.txt') as f:`
	`  for line in f:`
	`  if digs.search(line):`
	`  print(line, end='')`

split	*r*.split(*s*, *maxsplit*=0)
	Возвращает список *L* всех *разбиений* *s* по *r* (т.е. список подстрок *s*, разделенных непересекающимися, непустыми подстроками, соответствующими *r*). Например, ниже продемонстрирован один из способов удаления из строки всех вхождений подстроки 'hello' (при любых комбинациях верхнего и нижнего регистров букв):
	`import re`
	`rehello = re.compile(r'hello', re.IGNORECASE)`
	`astring = ''.join(rehello.split(astring))`

Метод	Описание

	Если *r* имеет *n* групп, то все они попадут в список *L*, перемежаясь с подстроками между разделителями. Каждый из этих *n* дополнительных элементов является подстрокой *s*, представляющей соответствующую группу в данном сопоставлении, или значение None, если эта группа в сопоставлении не участвует. Например, ниже продемонстрирован один из способов удаления пробела только в том случае, если он встречается между двоеточием и цифрой:
	`import re`
	`re_col_ws_dig = re.compile(r'(:)\s+(\d)')`
	`astring = ''.join(re_col_ws_dig.split(astring))`
	Если *maxsplit* больше 0, то в список *L* попадают самое большее *maxsplit* разбиений, за каждым из которых следуют *n* элементов в соответствии с приведенным выше описанием, тогда как замыкающая подстрока *s*, расположенная после *maxsplit* соответствий *r*, если таковые имеются, выступает в качестве последнего элемента *L*. Например, для удаления только первого вхождения подстроки 'hello', а не всех таких подстрок, замените последнюю инструкцию в приведенном выше первом примере следующей инструкцией:
	`astring = ''.join(rehello.split(astring, 1))`

sub	*r*.sub(*repl*, *s*, *count*=0)
	Возвращает копию *s*, где непересекающиеся соответствия *r* заменены аргументом *repl*, который может быть строкой или вызываемым объектом, таким как функция. Пустое соответствие заменяется только в том случае, если оно не является смежным по отношению к предыдущему соответствию. Если *count* больше 0, то в строке *s* заменяются только первые *count* соответствий *r*. Если *count* равно 0, заменяются все соответствия *r* в пределах *s*. Например, ниже продемонстрирован другой, более естественный способ удаления только первого вхождения подстроки 'hello' при любых комбинациях верхнего и нижнего регистров букв.
	`import re`
	`rehello = re.compile(r'hello', re.IGNORECASE)`
	`astring = rehello.sub('', astring, 1)`
	Без указания 1 в качестве последнего аргумента метода sub в этом примере удалялись бы все вхождения 'hello'.
	Если *repl* — вызываемый объект, то он должен принимать один аргумент (объект соответствия) и возвращать строку (или значение None, эквивалентное возврату пустой строки ''), используемую для замены соответствий. В данном случае sub вызывает repl с подходящим аргументом в виде объекта соответствия для каждого соответствия *r*, заменяемого методом sub. Например, ниже показан один из способов перевода в верхний регистр всех вхождений слов, начинающихся с буквы 'h' и заканчивающихся буквой 'o' при любых комбинациях верхнего и нижнего регистров букв

Метод	Описание

	`.import re`
	`h_word = re.compile(r'\bh\w*o\b', re.IGNORECASE)`
	`def up(mo): return mo.group(0).upper()`
	`astring = h_word.sub(up, astring)`
	Если *repl* — строка, то sub использует сам аргумент *repl* в качестве строки замены, если не считать того, что при этом расширяются обратные ссылки. *Обратная ссылка* — это подстрока *repl* вида \g<*id*>, где *id* — это имя группы в *r* (установленное элементом (?P<*id*>...) в строке шаблона *r*), или *dd*, где *dd* — одна или две цифры номера группы. Каждая обратная ссылка, именованная или нумерованная, заменяется подстрокой *s*, соответствующей группе *r*, на которую указывает обратная ссылка. Например, ниже показано, как заключить каждое слово в фигурные скобки.
	`import re`
	`grouped_word = re.compile('(\w+)')`
	`astring = grouped_word.sub(r'{\1}', astring)`

subn	*r*.subn(*repl*, *s*, *count*=0)
	Метод subn аналогичен методу sub, за исключением того, что он возвращает пару (*new_string*, *n*), где *n* — количество замен, выполненных методом subn. Например, ниже продемонстрирован один из способов подсчета количества вхождений подстроки 'hello' при любых комбинациях верхнего и нижнего регистров букв.
	`import re`
	`rehello = re.compile(r'hello', re.IGNORECASE)`
	`_, count = rehello.subn('', astring)`
	`print('Found', count, 'occurrences of "hello"')`

Объекты совпадений

Объекты совпадений создаются и возвращаются методами match и search объекта регулярного выражения и являются элементами итератора, возвращаемого методом finditer. Кроме того, они неявно создаются методами sub и subn, если в качестве аргумента repl указан вызываемый объект, поскольку в этом случае соответствующий объект совпадения передается в качестве аргумента при каждом вызове repl. Ниже перечислены доступные только для чтения атрибуты объекта совпадения *m*, содержащие информацию о способе его создания.

pos
: Значение аргумента *start*, который был передан методу search или match (т.е. индекс элемента в *s*, с которого начинался поиск соответствия).

endpos
: Значение аргумента *end*, который был передан методу search или match (т.е. индекс элемента в *s*, вплоть до которого должен был выполняться поиск подстроки *s*, соответствующей регулярному выражению).

`lastgroup`
: Имя последней группы, для которой было найдено соответствие (значение `None`, если последняя из этих групп не обладает именем или если группы в сопоставлении не участвовали).

`lastindex`
: Целочисленный индекс (`1` и выше) последней группы, для которой было найдено соответствие (значение `None`, если группы в сопоставлении не участвовали).

`re`
: Объект РВ, метод которого был использован для создания *m*.

`string`
: Строка *s*, которая была передана методу `finditer`, `match`, `search`, `sub` или `subn`.

Методы, предоставляемые объектом *m*, приведены в табл. 9.3.

Таблица 9.3. Методы объекта совпадений

Метод	Описание
end, span, start	*m*.end(*groupid*=0) *m*.span(*groupid*=0) *m*.start(*groupid*=0) Эти методы возвращают граничные индексы подстроки *m*.string, которая соответствует группе, идентифицируемой аргументом *groupid* (номер или имя группы; используемый по умолчанию аргумент "группа 0" означает "все РВ"). Если совпавшей подстрокой является *m*.string[*i*:*j*], то *m*.start возвращает *i*, *m*.end возвращает *j*, а *m*.span возвращает кортеж (*i*, *j*). Если группа не участвовала в сопоставлении, то каждый из индексов *i* и *j* равен -1
expand	*m*.expand(*s*) Возвращает копию *s*, в которой экранированные последовательности и обратные ссылки заменены тем же способом, что и в случае метода *r*.sub, описанного в табл. 9.2
group	*m*.group(*groupid*=0, **groupids*) Если метод group вызывается с единственным аргументом *groupid* (номер или имя группы), то он возвращает подстроку, которая совпадает с группой, идентифицируемой аргументом *groupid*, или значение None, если данная группа не участвовала в сопоставлении. Идиома *m*.group(), которая также может быть записана в виде *m*.group(0), возвращает всю совпавшую подстроку, поскольку "группа 0" означает "все РВ". Если метод group вызывается с несколькими аргументами, то каждый аргумент должен быть номером или именем группы. В этом случае метод group возвращает кортеж, содержащий по одному элементу на каждый

Метод	Описание
	аргумент. Каждый элемент представляет собой подстроку, совпавшую с соответствующей группой, или значение `None`, если данная группа не участвовала в сопоставлении
`groups`	`m.groups(`*`default`*`=None)` Возвращает кортеж, содержащий по одному элементу на каждую группу в *`r`*. Каждый элемент представляет собой строку, совпавшую с соответствующей группой, или значение *`default`*, если данная группа не участвовала в сопоставлении
`groupdict`	`m.groupdict(`*`default`*`=None)` Возвращает словарь, ключами которого являются имена всех именованных групп в *`r`*. Значением каждого имени является подстрока, совпавшая с соответствующей группой, или значение *`default`*, если данная группа не участвовала в сопоставлении

Функции модуля `re`

Модуль `re` предоставляет атрибуты, перечисленные в разделе "Флаги опций". Он также предоставляет по одной функции для каждого из методов объекта регулярного выражения (`findall`, `finditer`, `match`, `search`, `split`, `sub` и `subn`), и каждая функция имеет дополнительный первый аргумент, представляющий собой строку шаблона, который функция неявно компилирует в объект РВ. Как правило, лучше компилировать строку шаблона в объект РВ явным образом и вызывать методы объекта РВ, но иногда, как разовая мера, вызов функции модуля `re` может оказаться более удобным. Например, для подсчета количества вхождений строки `'hello'` при любых комбинациях верхнего и нижнего регистров букв можно предложить следующий компактный способ, основанный на вызове функций:

```
import re
_, count = re.subn(r'(?i)hello', '', astring)
print('Found', count, 'occurrences of "hello"')
```

В подобных случаях опции РВ (в данном случае игнорирование регистра букв) должны задаваться в виде элементов шаблона РВ (в данном случае в виде элемента `(?i)`): функции модуля `re` не поддерживают аргумент *`flags`*. Модуль `re` кеширует объекты РВ, которые создаются на основе строк шаблонов, передаваемых функциям. Чтобы очистить кеш и освободить занимаемую им память, следует вызвать функцию `re.purge()`.

Модуль `re` также предоставляет класс `error`, являющийся классом исключений, которые возбуждаются при возникновении ошибок (обычно синтаксических ошибок, обнаруженных в строке шаблона), и две дополнительные функции, описание которых приводится ниже.

compile `compile(`*pattern*`, `*flags*`=0)`

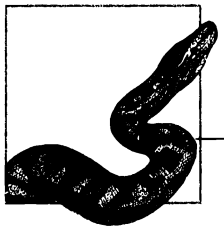
Создает и возвращает объект РВ, осуществляя разбор строки в соответствии с синтаксисом, описанным в разделе "Синтаксис строковых шаблонов", и используя целочисленные флаги, описанные в разделе "Флаги опций".

escape `escape(`*s*`)`

Возвращает копию строки *s*, в которой каждый символ, не являющийся алфавитно-цифровым, экранирован (посредством символа обратной косой черты, \). Полезна при литеральном сопоставлении строки *s* как части строки шаблона РВ.

III

Библиотека Python и модули расширения



10

Работа с файлами и текстом

В этой главе мы обсудим большинство вопросов, относящихся к работе с файлами и файловыми системами в Python. *Файл* — это поток текста или байтов, который программа может читать и/или записывать; *файловая система* — это иерархическое хранилище файлов в компьютерной системе.

Другие главы, также посвященные работе с файлами

Данная глава имеет довольно внушительный объем, однако роль файлов в программировании настолько велика, что рассказ о них будет продолжен еще в нескольких главах, в которых обсуждаются специфические типы файлов. В частности, в главе 11 будут рассмотрены разновидности файлов, связанных с обеспечением функциональности постоянного хранения данных (персистентности) и базами данных (файлы JSON в разделе "Модуль `json`", файлы `pickle` в разделе "Модули `pickle` и `cPickle`", файлы `shelve` в разделе "Модуль `shelve`", файлы DBM и DBM-подобные файлы в разделе "Пакет v3 `dbm`" и файлы базы данных SQLite в разделе "SQLite"), в главе 22 речь пойдет о файлах и других потоках в формате HTML, тогда как глава 23 посвящена файлам и другим потокам в XML-формате.

Структура этой главы

Файлы и потоки имеют много обличий. Они могут содержать произвольные байты или текст (в различных кодировках, если базовое хранилище или канал работает только с байтами, как это и бывает в большинстве случаев). Они могут предназначаться только для чтения, или только для записи данных, или для того и другого одновременно. Одни файлы используют *буферизацию*, другие не используют. Файлы могут разрешать или не разрешать прямой или, как говорят, "произвольный" доступ к содержащимся в них данным, допускающий свободное перемещение между позициями в файле в обоих направлениях (например, потоки, работающие через сокеты,

допускают лишь последовательный доступ к данным с продвижением только вперед — возможность перемещаться между позициями в таком файле в прямом и обратном направлениях отсутствует).

Python старого стиля традиционно объединял всю эту разнородную функциональность в одном встроенном объекте `file`, работающем по-разному в зависимости от того, каким его создала встроенная функция `open`. Эти встроенные объекты все еще доступны в версии v2 в интересах обратной совместимости.

Однако как в версии v2, так и в версии v3 средства ввода-вывода (I/O) структурированы в более логичной манере и теперь находятся в модуле `io` стандартной библиотеки Python. В версии v3 встроенная функция `open` по сути является просто псевдонимом функции `io.open`. В версии v2 встроенная функция `open` работает как и раньше, создавая и возвращая старый встроенный объект `file` (отсутствующий в версии v3). Однако инструкция `from io import open` позволяет использовать вместо встроенной функции `open` новую и лучше структурированную функцию `io.open`: возвращаемые ею файловые объекты в большинстве простых случаев работают почти аналогично прежним встроенным объектам `file`, что позволяет вызывать ее точно так же, как и старую встроенную функцию `open`. (Разумеется, как в версии v2, так и в версии v3 вы вправе действовать по-другому и, руководствуясь великолепным принципом "явное лучше, чем неявное", сначала явно импортировать модуль `io`, а затем непосредственно использовать функцию `io.open`.)

Если вам приходится поддерживать старый код, в котором сложным образом используются встроенные объекты `file`, и вы не хотите портировать его в новые версии, вы сможете узнать о них больше, обратившись к онлайн-документации (`https://docs.python.org/2.7/library/stdtypes.html#file-objects`). В этой книге старые встроенные объекты `file` не рассматриваются, поэтому данная глава начинается с обсуждения функции `io.open` и различных классов, включенных в модуль `io`.

Сразу после этого в разделе "Объекты, подобные файлам, и полиморфизм" мы обсудим полиморфное понятие объектов, подобных файлам (объекты, которые на самом деле не являются файлами, но в некотором роде ведут себя как таковые).

Далее рассматриваются модули, предназначенные для работы с временными файлами и подобными им объектами (модуль `tempfile` — в разделе "Модуль `tempfile`" и модули `io.StringIO` и `io.BytesIO` — в разделе "Файлы в памяти: функции `io.StringIO` и `io.BytesIO`").

Следующим предметом рассмотрения этой главы являются модули, обеспечивающие доступ к содержимому текстовых и двоичных файлов (модуль `fileinput` — в разделе "Модуль `fileinput`", модуль `linecache` — в разделе "Модуль `linecache`" и модуль `struct` — в разделе "Модуль `struct`"), а также поддержка сжатых файлов и других архивов данных (модуль `gzip` — в разделе "Модуль `gzip`", модуль `bz2` — в разделе "Модуль `bz2`", модуль `tarfile` — в разделе "Модуль `tarfile`", модуль `zipfile` — в разделе "Модуль `zipfile`" и модуль `zlib` — в разделе "Модуль `zlib`"). В версии v3 также поддерживается алгоритм сжатия LZMA, используемый, например, программой `xz`: этот вопрос не рассматривается в данной книге, но

информацию о его ретроподдержке в версии v2 можно получить в онлайн-документации (`https://docs.python.org/3/library/lzma.html`) и на сайте PyPI (`https://pypi.python.org/pypi/backports.lzma`).

В Python модуль `os` предоставляет множество функций для работы с файловой системой, поэтому данная глава продолжается знакомством с модулем `os` в разделе "Модуль `os`". Затем в разделе "Операции в файловой системе" обсуждаются операции над файловой системой (сравнение, копирование и удаление каталогов и файлов, работа с путями доступа к файлам, а также низкоуровневые дескрипторы файлов), обеспечиваемые модулями `os` (раздел "Функции модуля `os` для работы с файлами и каталогами"), `os.path` (раздел "Модуль `os.path`") и другими модулями (модуль `dircache` — в описании функции `listdir` в табл. 10.3, модуль `stat` — в разделе "Модуль `stat`", модуль `filecmp` — в разделе "Модуль `filecmp`", модуль `fnmatch` — в разделе "Модуль `fnmatch`", модуль `glob` — в разделе "Модуль `glob`" и модуль `shutil` — в разделе "Модуль `shutil`"). Мы исключили из рассмотрения модуль `pathlib`, предоставляющий возможность использования объектно-ориентированного подхода для работы с файловой системой, поскольку на момент выхода книги он был включен в стандартную библиотеку лишь как *экспериментальный*, а это означает, что в любой момент в него могут быть внесены изменения, нарушающие обратную совместимость, не говоря уже о том, что он может быть вообще удален из библиотеки. Если вы все же хотите поработать с ним, ознакомьтесь с соответствующим разделом онлайн-документации (`https://docs.python.org/3/library/pathlib.html`) и информацией, приведенной на сайте PyPI (`https://pypi.python.org/pypi/pathlib/`).

Несмотря на то что большинство современных программ основано на графическом интерфейсе пользователя (GUI), чисто текстовые, лишенные графики пользовательские интерфейсы командной строки по-прежнему в ходу, поскольку они отличаются простотой, их можно быстро программировать и они не требуют больших накладных расходов. Завершающая часть главы посвящена рассмотрению средств текстового ввода-вывода в Python (раздел "Ввод и вывод текста"), расширенных средств текстового ввода-вывода (раздел "Расширенные возможности текстового ввода-вывода"), интерактивных сеансов работы с командной строкой (раздел "Интерактивные сеансы") и, наконец, такой важной темы, как *интернационализация*. Описанию способов создания программного обеспечения, отвечающего требованиям различных региональных и языковых стандартов, посвящен раздел "Интернационализация".

Модуль `io`

Как уже отмечалось в предыдущем разделе, модуль `io` стандартной библиотеки Python обеспечивает выполнение большинства распространенных операций, связанных с чтением и записью файлов. Прежде чем выполнять операции чтения и/или записи данных с использованием файла в том виде, в каком он известен операционной системе, необходимо создать объект "файла" с помощью вызова функции `io.open`.

В зависимости от переданных ей параметров эта функция создает экземпляр io.TextIOWrapper для текстового "файла" или, если речь идет о двоичном "файле", экземпляр io.BufferedReader, io.BufferedWriter или io.BufferedRandom, доступный, соответственно, только для чтения, только для записи или для чтения и записи. Мы используем кавычки, ссылаясь на эти объекты как на "файлы", чтобы отличить их от встроенных объектов file, которые по-прежнему существуют в версии v2.

В версии v3 встроенная функция open — синоним функции io.open. В версии v2 тот же эффект обеспечивает инструкция from io import open. Для большей ясности и во избежание двусмысленности мы предпочитаем использовать явный вызов функции io.open (разумеется, это предполагает предварительное выполнение инструкции import io).

В этом разделе также рассматривается важный вопрос, касающийся создания и использования *временных* файлов (сохраняемых на диске или в памяти).

На любую ошибку ввода-вывода, связанную с объектом "файла", Python реагирует возбуждением экземпляра встроенного класса исключения IOError (в версии v3 это синоним исключения OSError, но существуют и другие полезные классы исключений, рассмотренные в разделе "Класс OSError и его подклассы (только в версии v3)" главы 5). К числу ошибок, приводящих к возбуждению этого исключения, относятся неудачная попытка открытия файла, вызов метода для "файла", к которому данный метод неприменим (например, вызов метода write для "файла", доступного только для чтения, или вызов метода seek для файла, не допускающего выполнение операции поиска), — что также может приводить к возбуждению исключения ValueError или AttributeError, — и ошибки ввода-вывода, диагностируемые методами объекта "файла".

Кроме того, модуль io предоставляет множество базовых классов, как абстрактных, так и конкретных, которые с помощью наследования или композиции (также известной как "обертывание" или "упаковывание" классов) создают объекты "файлов" (экземпляры классов, упомянутых в первом абзаце этого раздела), используемые вашей программой. Эти более сложные темы в данной книге не рассматриваются. Если вам приходится иметь дело с необычными каналами передачи данных или типами хранилищ, отличными от файловых систем, и вы хотите обеспечить "файловый" интерфейс для доступа к таким каналам и хранилищам, можете облегчить свою задачу, наследуя и упаковывая другие классы, предоставляемые модулем io. Для получения более подробной информации по этому вопросу обратитесь к онлайн-документации (https://docs.python.org/3/library/io.html).

Создание объекта "файла" с помощью метода io.open

Чтобы создать объект "файла" Python, следует вызвать метод io.open, который имеет следующий синтаксис:

```
open(file, mode='r', buffering=-1, encoding=None, errors='strict',
newline=None, closefd=True, opener=os.open)
```

Аргументом *file* может быть строка, представляющая путь доступа к файлу в том виде, в каком он известен базовой операционной системе, или целое число, представляющее дескриптор файла на уровне ОС, возвращаемый функцией os.open (или, только в версии v3, функцией, переданной в качестве аргумента opener; эта функция не поддерживается в версии v2). Если аргументом *file* является строка с именем файла, то функция open открывает этот файл (возможно, создавая его, в зависимости от значения аргумента *mode* — несмотря на свое название, функция open способна не только открывать файлы, но и создавать новые файлы). Если аргумент *file* является целым числом, то файл в операционной системе уже должен быть открыт (посредством вызова функции os.open или любым другим возможным способом).

Открывайте файлы способом, соответствующим "духу Python"

Функция open является менеджером контекста: используйте инструкцию `with io.open(...) as f:`, а не инструкцию `f = io.open(...)`, тем самым гарантируя, что "файл" *f* будет закрыт сразу же по завершении тела инструкции with.

Функция open создает и возвращает экземпляр *f* соответствующего класса модуля io, в зависимости от режима открытия файла и буферизации данных, определяемых аргументами *mode* и *buffering* (мы называем экземпляры подобного рода объектами "файлов"; все они в достаточной степени полиморфны по отношению друг к другу).

Аргумент mode

Аргумент *mode* — это строка, задающая режим открытия (или создания) файла. Этот аргумент может принимать следующие значения.

`'r'`
: Существующий файл открывается в режиме "только для чтения".

`'w'`
: Файл открывается в режиме "только для записи". Файл урезается до нулевой длины и его содержимое полностью заменяется, если файл уже существвовал, или создается в противном случае.

`'a'`
: Файл открывается в режиме "только для записи". Прежнее содержимое файла, если к этому моменту он уже существовал, сохраняется, и записываемые вами новые данные присоединяются к уже имеющемуся содержимому. Если указанного файла не существует, то он создается. Вызов метода *f*.seek для файла изменяет результат вызова метода *f*.tell, но не изменяет позицию записи в файл.

`'r+'`

Существующий файл открывается как для чтения, так и для записи, поэтому для *f* может быть вызван любой метод.

`'w+'`

Файл открывается как для чтения, так и для записи, поэтому для *f* может быть вызван любой метод. Файл урезается до нулевой длины, и его содержимое полностью заменяется, если он уже существует, или создается, если до этого он не существовал.

`'a+'`

Файл открывается как для чтения, так и для записи, поэтому для *f* может быть вызван любой метод. Прежнее содержимое файла, если к этому моменту он уже существовал, сохраняется, и записываемые новые данные присоединяются к существующему содержимому. Если указанного файла не существует, то он создается. В зависимости от базовой операционной системы вызов метода `f.seek` может не оказывать никакого воздействия, если следующей операцией, выполняемой для объекта *f*, является запись данных, или же работать как обычно, если следующей операцией является чтение данных.

Двоичный и текстовый режимы

Любому из описанных выше значений *mode* может предшествовать буква `b` или `t`. Буква `b` означает двоичный файл, буква `t` — текстовый. По умолчанию отсутствие этих букв означает текстовый файл (т.е. `'r'` — это то же самое, что `'rt'`, `'w'` — то же самое, что `'wt'`, и т.д.).

Двоичные файлы позволяют читать и/или записывать строки типа `bytes`; текстовые файлы позволяют читать и/или записывать текстовые строки Unicode (`str` в версии v3, `unicode` в версии v2). В случае текстовых файлов, если базовый канал или хранилище работает с байтами (как это бывает в большинстве случаев), аргументы *encoding* (название кодировки, известное Python) и *errors* (имя обработчика ошибок, например `'strict'`, `'replace'` и т.п., в соответствии с описанием метода `decode` в табл. 8.6) играют важную роль, поскольку они определяют способ преобразования байтов в текст и наоборот, а также порядок действий при возникновении ошибок кодирования/декодирования.

Аргумент *buffering*

Аргумент *buffering* — это целое число, обозначающее запрашиваемый для файла размер буфера. Если это число меньше 0, используется значение, заданное по умолчанию. Как правило, для файлов, которые соответствуют интерактивным консолям, используется буферизация строк, тогда как для других типов файлов размер буфера в байтах по умолчанию определяется значением `io.DEFAULT_BUFFER_SIZE`. Если значение *buffering* равно `0`, то буфер не используется. В этом случае операции выполняются так, как если бы буфер сбрасывался каждый раз при записи в файл.

Значению *buffering*, равному 1, соответствует буферизация строк (в этом случае должен быть установлен текстовый режим), т.е. буфер файла сбрасывается всякий раз, когда в файл записывается символ \n . Если значение *buffering* превышает 1, то размер буфера в байтах округляется до некой разумной величины, близкой к значению *buffering*.

Последовательный и произвольный виды доступа

Объект "файла" имеет последовательную внутреннюю структуру (поток байтов или текст). При выполнении операции чтения вы последовательно получаете байты или текст в том порядке, в каком они представлены. При выполнении операции записи записываемые байты или текст добавляются в том порядке, в каком вы их записываете.

Чтобы предоставить возможность произвольного доступа (также известного как *случайный доступ*), объект "файла", базовое хранилище которого допускает выполнение подобных операций, отслеживает свою текущую позицию (позицию в реальном файле, с которой начнется выполнение следующей операции чтения или записи данных). Если объект *f* поддерживает произвольный доступ, то метод `f.seekable()` возвращает значение `True`.

Когда вы открываете файл, начальная позиция совпадает с началом файла. Любой вызов метода `f.write` для объекта *f* "файла", открытого в режиме `'a'` или `'a+'`, всегда устанавливает указатель позиции *f* на конец файла, прежде чем записать данные в *f*. Когда вы читаете *n* байтов из объекта "файла" *f* или записываете их в "файл", указатель позиции получает приращение *n*. Для выяснения или изменения текущей позиции служат, соответственно, методы `f.tell` и `f.seek`, описанные в следующем разделе.

Методы `f.tell` и `f.seek` работают также в текстовом режиме, но в этом случае смещение, передаваемого методу `f.seek`, должно иметь значение `0` (для перемещения в начало или конец файла, в зависимости от значения второго параметра метода `f.seek`) или непрозрачное значение, возвращенное перед этим методом `f.tell`, для перемещения в "отмеченную" ранее позицию в *f*.

Атрибуты и методы файловых объектов

В этом разделе описаны атрибуты и методы, предоставляемые объектом "файла" *f*.

close `f.close()`

Закрывает файл. После этого вызова для *f* невозможен никакой другой вызов. Повторные вызовы `f.close` допускаются, но не оказывают никакого воздействия.

closed `closed`

`f.closed` — атрибут, доступный только для чтения и имеющий значение `True`, если до этого был выполнен вызов `f.close()`; в противном случае он имеет значение `False`.

encoding encoding

f.encoding — атрибут, доступный только для чтения и представляющий собой строку с названием кодировки (см. раздел "Unicode" в главе 8). Двоичные "файлы" не имеют этого атрибута.

flush *f*.flush()

Запрашивает сброс буфера *f* для его записи в операционную систему, чтобы содержимое реального файла, известного системе, было в точности таким, каким его записал код на Python. В зависимости от платформы и природы реального базового файла, которому соответствует *f*, вызов *f*.flush может не гарантировать достижения желаемого эффекта.

isatty *f*.isatty()

Возвращает значение True, если базовым файлом, которому соответствует f, является интерактивный терминал; в противном случае возвращает значение False.

fileno *f*.fileno()

Возвращает целое число, представляющее дескриптор базового файла, которому соответствует f, на уровне операционной системы. Дескрипторы файлов описаны в разделе "Функции модуля os для работы с файлами и каталогами".

mode mode

f.mode — атрибут, доступный только для чтения и содержащий значение строки *mode*, использованное при вызове функции io.open, с помощью которой был создан объект *f*.

name name

f.name — атрибут, доступный только для чтения и содержащий значение строки *file* или целое значение, использованное при вызове функции io.open, с помощью которой был создан объект *f*.

read *f*.read(*size*=-1)

В версии v2 или в версии v3, если *f* открыт в двоичном режиме, метод read читает вплоть до *size* байтов из базового объекта *f* и возвращает их в виде байтовой строки. Если конец файла достигается прежде, чем считываются *size* байт, то метод read считывает и возвращает меньшее количество байтов, чем определено аргументом *size*. Если *size* меньше 0, метод read считывает и возвращает все байты до конца файла. Если текущая позиция находится в конце файла или аргумент *size* задан равным 0, то метод read возвращает пустую строку. В версии v3, если *f* открыт в текстовом режиме, то аргумент *size* определяет количество символов, а не байтов, и метод read возвращает тестовую строку.

readline *f*.readline(*size*=-1)

Читает и возвращает одну текстовую строку из файла *f* до достижения символа конца строки (\n), включая сам этот символ. Если значение *size*

больше или равно 0, то метод `readline` читает не более чем *size* байт. В этом случае возвращенная строка может не заканчиваться символом `\n`. Символ `\n` также может отсутствовать, если метод `readline` выполняет чтение до конца файла, не находя символа `\n`. Если текущая позиция находится в конце файла или аргумент *size* задан равным 0, то метод `readline` возвращает пустую строку.

readlines `f.readlines(size=-1)`

Читает и возвращает список всех строк, содержащихся в файле *f*, причем каждая строка заканчивается символом *\n*. Если *size*>0, то метод `readlines` прекращает работу и возвращает список после того, как соберет данные, общий размер которых составляет примерно *size* байт, а не читает данные до конца файла. В этом случае последняя строка может не заканчиваться символом `\n`.

seek `f.seek(pos, how=io.SEEK_SET)`

Устанавливает в качестве текущей позицию *f*, отстоящую от начала отсчета на *pos* байт, где *pos* — целое число со знаком. Начало отсчета задает аргумент *how*. Модуль `io` имеет атрибуты `SEEK_SET`, `SEEK_CUR` и `SEEK_END`, позволяющие задавать в качестве начала отсчета начало файла, текущую позицию и конец файла соответственно.

Если *f* открыт в текстовом режиме, то аргумент *pos* метода `f.seek` должен иметь значение 0 или, только в случае начала отсчета `io.SEEK_SET`, значение результата, возвращенного предыдущим вызовом метода `f.tell`.

Если *f* открыт в режиме `'a'` или `'a+'`, то на некоторых, но не на всех, платформах данные, записываемые в *f*, присоединяются к данным, уже содержащимся в *f*, независимо от результатов вызовов `f.seek`.

tell `f.tell()`

Возвращает текущую позицию *f*: для двоичного файла — целочисленное смещение в байтах от начала файла; для текстового файла — непрозрачное значение, которое впоследствии может быть использовано вызовами метода `f.seek` для возврата в позицию *f*, которая в данный момент является текущей.

truncate `f.truncate([size])`

Усекает базовый файл объекта *f*, который должен быть открыт для записи. Если задан аргумент *size*, файл усекается до размера, не превышающего *size* байт. Если аргумент *size* не задан, то в качестве нового размера файла используется значение, возвращаемое методом `f.tell()`.

write `f.write(s)`

Записывает в файл байты строки *s* (двоичные или текстовые, в зависимости от режима *f*).

writelines `f.writelines(lst)`

Аналогичен инструкции

```
for line in lst: f.write(line)
```

Не имеет значения, содержат ли строки итерируемого объекта маркеры конца строки: несмотря на свое название, метод `writelines` просто записывает каждую из строк в файл, одну за другой. В частности, метод `writelines` не добавляет маркеры конца строки: такие маркеры, если они нужны, уже должны быть в списке.

Итерирование по файловым объектам

Объект *f* "файла", открытого для чтения в текстовом режиме, является также итератором, элементами которого служат строки файла. Таким образом, следующий цикл итерирует по строкам файла:

```
for line in f:
```

Если мы вспомним о буферизации, то станет понятно, что преждевременное прерывание такого цикла (например, с помощью инструкции `break`) или вызов функции `next(f)` вместо метода `f.readline()` оставляет в указателе позиции произвольное значение. Если вы хотите перейти от использования объекта *f* в качестве итератора к вызовам других методов чтения для объекта *f*, не забудьте установить для позиции в файле известное значение посредством соответствующего вызова метода `f.seek`. Положительной стороной применения цикла непосредственно к объекту *f* является повышение производительности, поскольку в этих циклах используется внутренняя буферизация, минимизирующая количество операций ввода-вывода, которые в этом случае потребляют лишь незначительные объемы памяти даже при работе с файлами огромных размеров.

Объекты, подобные файлам, и полиморфизм

Об объекте *x* говорят, что он *подобен файлу*, если он ведет себя *полиморфно* по отношению к объекту "файла", возвращаемому функцией `io.open`, т.е. его можно использовать так, как если бы он был "файлом". Код, использующий такой объект (*клиентский код* объекта), обычно получает его в виде аргумента или посредством вызова функции-фабрики, которая возвращает результат в виде объекта. Например, если единственным методом, который вызывается клиентским кодом для *x*, является метод `x.read()` без аргументов, то все, что требуется от *x* для того, чтобы с точки зрения данного кода его поведение было подобно поведению файла, — это предоставить метод `read`, который вызывается без аргументов и возвращает строку. Другому клиентскому коду объект *x* может понадобиться для реализации большего подмножества методов файлов. Объекты, подобные файлам, и полиморфизм не являются абсолютными понятиями: они определяются запросами, предъявляемыми к объекту конкретным клиентским кодом.

Полиморфизм — мощный аспект объектно-ориентированного программирования, и объекты, подобные файлам, могут служить отличным примером полиморфизма. Модуль клиентского кода, который записывает или читает данные из файлов,

можно повторно использовать для данных, находящихся где угодно, при условии, что он не разрушает полиморфизм сомнительной практикой проверки типов.

При обсуждении встроенных объектов `type` и `isinstance`, описанных в табл. 7.1, отмечалось, что проверку типов чаще всего лучше избегать, поскольку она блокирует обычный полиморфизм, предоставляемый Python. В большинстве случаев все, что вы должны сделать для того, чтобы ваш клиентский код поддерживал полиморфизм, — это избегать проверки типов.

Вы сможете самостоятельно реализовать объект, подобный файлу, написав собственный класс (о том, как это делается, рассказано в главе 4) и определив специфические методы, в которых нуждается клиентский код, такие как метод `read`. Объект *`fl`*, подобный файлу, не должен реализовывать все атрибуты и методы истинного объекта *`f`* "файла". Если вы способны определить, какие методы вызываются клиентским кодом для объекта *`fl`*, то вам достаточно реализовать только это подмножество методов. Например, если известно, что объект *`fl`* будут использоваться только для записи, то ему не нужны такие методы, осуществляющие чтение данных, как `read`, `readline` и `readlines`.

Если основной причиной того, что вы хотите использовать объект, подобный файлу, вместо объекта реального файла является желание держать данные в памяти, то используйте входящие в модуль `io` классы `StringIO` и `BytesIO`, описанные в разделе "Файлы в памяти: функции `io.StringIO` и `io.BytesIO`". Эти классы предоставляют объекты "файлов", которые хранят данные в памяти и в значительной степени ведут себя полиморфно по отношению к другим объектам "файлов".

Модуль `tempfile`

Модуль `tempfile` позволяет создавать временные файлы и каталоги, делая эти операции максимально безопасными в рамках возможностей вашей платформы. Временные файлы часто могут выручить, если приходится работать с большими объемами данных, не умещающимися в памяти, или если программа должна записывать данные, предназначенные для последующего использования другими процессами.

Порядок следования параметров в функциях этого модуля немного сбивает с толку: чтобы улучшить читаемость кода, всегда вызывайте эти функции, используя синтаксис именованных аргументов. Функции и классы, предлагаемые модулем `tempfile`, кратко описаны в табл. 10.1.

Таблица 10.1. Функции и классы модуля `tempfile`

Функция	Описание
`mkdtemp`	`mkdtemp(`*`suffix`*`=None, `*`prefix`*`=None, `*`dir`*`=None)` Безопасно создает новый временный каталог, разрешающий выполнять операции чтения, записи и поиска только текущему пользователю, и возвращает абсолютный путь к вновь созданному каталогу. Необязательные аргументы *`suffix`*, *`prefix`* и *`dir`* имеют тот же смысл, что и в функции `mkstemp`.

Функция	Описание

	Ответственность за удаление временного каталога после того, как необходимость в нем отпала, возлагается на программу. Ниже приведен типичный пример использования этой функции, в котором сначала создается временный каталог, затем путь доступа к нему передается другой функции, после чего обеспечивается удаление временного каталога вместе со всем его содержимым.
	`import tempfile, shutil`
	`path = tempfile.mkdtemp()`
	`try:`
	`  use_dirpath(path)`
	`finally:`
	`  shutil.rmtree(path)`

mkstemp	`mkstemp(`*`suffix`*`=None, `*`prefix`*`=None, `*`dir`*`=None, `*`text`*`=False)`
	Безопасно создает новый временный файл, предоставляющий текущему пользователю права чтения и записи, но не выполнения файла, и не наследуемый подпроцессами. Возвращает пару (*`fd`*, *`path`*), где *`fd`* — дескриптор временного файла (возвращаемый функцией `os.open`, описанной в табл. 10.5), а строка *`path`* — абсолютный путь доступа к временному файлу. Необязательные аргументы позволяют определить строки, которые будут добавлены в начале (*`prefix`*) и в конце (*`suffix`*) имени временного файла, а также путь к каталогу, в котором создается временный файл (*`dir`*). Для создания текстового временного файла следует явно задать аргумент *`text`*`=True`.
	Ответственность за удаление временного файла после того, как необходимость в его использовании отпала, возлагается на вас: `mkstemp` не является менеджером контекста, поэтому вы не можете воспользоваться инструкцией `with` — вместо этого обычно используют инструкцию `try/finally`. Ниже приведен типичный пример использования данной функции, в котором сначала создается временный файл, затем этот файл закрывается, а путь доступа к нему передается другой функции, после чего файл удаляется.
	`import tempfile, os`
	`fd, path =`
	`  tempfile.mkstemp(suffix='.txt', text=True)`
	`try:`
	`  os.close(fd)`
	`  use_filepath(path)`
	`finally:`
	`  os.unlink(path)`

Функция	Описание
SpooledTemporaryFile	`SpooledTemporaryFile(`*mode*`='w+b', `*bufsize*`=-1, `*suffix*`=None, `*prefix*`=None, `*dir*`=None)` Подобна функции `TemporaryFile`, за исключением того, что объект "файла", возвращаемый функцией `SpooledTemporaryFile`, может оставаться в памяти, если для этого хватает места, до тех пор, пока вы не вызовете его метод `fileno` (или метод `rollover`, гарантирующий материализацию файла на диске). Использование функции `SpooledTemporaryFile` позволяет повысить производительность при условии, что имеется достаточный запас свободной памяти
TemporaryFile	`TemporaryFile(`*mode*`='w+b', `*bufsize*`=-1, `*suffix*`=None, `*prefix*`=None, `*dir*`=None)` Создает временный файл с помощью функции `mkstemp` (передавая ей необязательные аргументы *suffix*, *prefix* и *dir*), создает на его основе объект "файла" с помощью функции `os.fdopen`, описанной в табл. 10.5 (передавая ей необязательные аргументы *mode* и *bufsize*), и возвращает объект "файла". Временный файл удаляется сразу после закрытия объекта файла (явным или неявным способом). В целях повышения безопасности у временного файла нет имени в файловой системе, если ваша платформа допускает такое (это возможно в случае Unix-подобных систем и невозможно в случае Windows). Возвращенный функцией `TemporaryFile` объект "файла" является менеджером контекста, поэтому его можно использовать вместе с инструкцией `with`, гарантирующей его закрытие после того, как он выполнит свои функции
NamedTemporaryFile	`NamedTemporaryFile(`*mode*`='w+b', `*bufsize*`=-1, `*suffix*`=None, `*prefix*`=None, `*dir*`=None)` Подобна функции `TemporaryFile`, за исключением того, что у временного файла есть имя в файловой системе. Для доступа к этому имени используйте атрибут `name` объекта "файла". Некоторые платформы (в основном это касается Windows) не допускают повторного открытия файлов. Поэтому использование имени может принести лишь ограниченную пользу, если вы заинтересованы в кросс-платформенности вашей программы. Если вам необходимо передать имя файла другой программе, открывающей данный файл, используйте функцию `mkstemp` вместо функции `NamedTemporaryFile`, чтобы обеспечить необходимое кросс-платформенное поведение программы. Разумеется, если вы решите использовать функцию `mkstemp`, вам придется самостоятельно позаботиться об удалении файла, когда необходимость в его использовании отпадет.

Функция	Описание
	Возвращенный функцией `NamedTemporaryFile` объект "файла" является менеджером контекста, поэтому его можно использовать вместе с инструкцией `with`

Вспомогательные модули файлового ввода-вывода

Объекты "файлов" обеспечивают минимальную функциональность, необходимую для выполнения операций файлового ввода-вывода. Однако существует ряд вспомогательных библиотечных модулей Python, предлагающих дополнительную функциональность, которая оказывается удобной в некоторых важных ситуациях, еще более упрощая выполнение указанных операций.

Модуль `fileinput`

Модуль `fileinput` позволяет организовывать циклы по всем строкам текстовых файлов из предоставленного списка. Модуль обеспечивает более высокую производительность по сравнению с непосредственным итерированием по элементам каждого файла в отдельности за счет буферизации, минимизирующей количество операций ввода-вывода. Поэтому вы можете смело использовать модуль `fileinput` для построчного ввода текста из файлов всякий раз, когда сочтете, что предлагаемая им богатая функциональность будет для вас удобной, не заботясь о проблемах производительности. Ключевой функцией модуля `fileinput` является функция `input`. Кроме того, данный модуль предоставляет класс `FileInput`, методы которого поддерживают ту же функциональность, что и функции модуля. Содержимое модуля `fileinput` описано ниже.

`close` `close()`

Закрывает всю последовательность, что приводит к прекращению итерирования, так что ни один файл не остается открытым.

`FileInput` `class FileInput(`*`files`*`=None, `*`inplace`*`=False, `*`backup`*`='', `*`bufsize`*`=0, `*`openhook`*`=None)`

Создает и возвращает экземпляр *`f`* класса `FileInput`. Аргументы те же, что и для метода `fileinput.input`, а методы *`f`* имеют те же имена, аргументы и семантику, что и функции модуля `fileinput`. Кроме того, *`f`* предоставляет метод `readline`, который читает и возвращает следующую строку. Вы можете использовать класс `FileInput` явно, если хотите организовать вложенные или смешанные циклы, которые читают строки из более чем одной последовательности файлов.

filelineno `filelineno()`

Возвращает количество строк, полученных из файла, чтение которого в данный момент выполняется. Например, если к данному моменту из текущего файла была прочитана одна строка, то функция `filelineno()` вернет значение `1`.

filename `filename()`

Возвращает имя читаемого файла или значение `None`, если к данному времени из этого файла не была прочитана ни одна строка.

input `input(`*`files`*`=None, `*`inplace`*`=False, `*`backup`*`='', `*`bufsize`*`=0, `*`openhook`*`=None)`

Возвращает последовательность строк, содержащихся в файлах, пригодную для использования в цикле `for`. Аргумент *`files`* — это последовательность имен файлов, которые открываются и из которых читаются данные в том порядке, в каком указаны их имена. Имя файла `'-'` означает стандартное устройство ввода (`sys.stdin`). Если *`files`* — строка, то она задает единственный файл, который должен быть открыт и прочитан. Если *`files`* имеет значение `None`, то в качестве списка имен файлов используется `sys.argv[1:]`. В случае пустой последовательности имен файлов функция `input` читает данные из стандартного потока `sys.stdin`.

Объект последовательности, возвращаемый функцией `input`, является экземпляром класса `FileInput`; этот экземпляр используется в качестве глобального состояния для всех функций данного модуля. Каждая функция модуля `fileinput` непосредственно соответствует одному из методов класса `FileInput`.

Если аргумент *`inplace`* имеет значение `False` (значение по умолчанию), то функция `input` только читает файлы. Если *`inplace`* имеет значение `True`, то функция `input` сохраняет резервную копию каждого читаемого файла (за исключением стандартного ввода) в файле с расширением *`backup`* и перенаправляет стандартный вывод (`sys.stdout`) в файл с тем же путем доступа, что и исходный файл. Благодаря этому можно имитировать перезапись файлов на месте. Если аргумент *`backup`* содержит строку, начинающуюся с символа точки, то функция `input` использует *`backup`* в качестве расширения файлов резервных копий и не удаляет эти файлы. Если *`backup`* — пустая строка (значение по умолчанию), то `input` использует расширение `.bak` и удаляет файлы резервных копий после закрытия исходных файлов.

Аргумент *`bufsize`* — это размер внутреннего буфера, используемого функцией `input` для чтения строк из входных файлов. Если *`bufsize`* имеет значение `0`, то функция `input` использует буфер размером 8192 байт.

Необязательный аргумент *`openhook`* позволяет задать функцию, которая будет использоваться вместо функции `io.open`. Популярный вариант аргумента — *`openhook`*`=fileinput.hook_compressed`, позволяющий прозрачно распаковывать входные файлы с расширениями `.gz` и `.bzip` (несовместимо с опцией *`inplace`*`=True`).

isfirstline `isfirstline()`

Возвращает значение `True` или `False` аналогично выражению `filelineno()==1`.

isstdin `isstdin()`

Возвращает значение `True`, если текущим файлом, из которого читаются данные, является файл `sys.stdin`; в противном случае возвращается значение `False`.

lineno `lineno()`

Возвращает общее количество строк, прочитанных с момента вызова функции `input`.

nextfile `nextfile()`

Закрывает текущий файл, из которого читаются данные: следующей строкой, подлежащей чтению, является первая строка следующего файла.

Ниже приведен типичный пример использования модуля `fileinput` для выполнения "многофайловых" операций поиска и замены, где одна строка заменяет другую во всех текстовых файлах, имена которых были переданы в качестве аргументов командной строки сценарию при его запуске.

```
import fileinput
for line in fileinput.input(inplace=True):
    print(line.replace('foo', 'bar'), end='')
```

В подобных случаях важно не забыть указать аргумент `end=''` функции `print`, поскольку каждая строка завершается символом конца строки `\n`, и вам необходимо позаботиться о том, чтобы к этому символу не был добавлен еще один (иначе вы получите вывод с удвоенным междустрочным интервалом).

Модуль `linecache`

Модуль `linecache` позволяет получить любую строку (указываемую с помощью ее номера) из файла с указанным именем и сохранить ее в кеше. Это обеспечивает ускорение операций, выполняемых над несколькими строками из одного файла, по сравнению с обработкой строк по отдельности, при которой файл открывается всякий раз, когда из него требуется извлечь строку. Предоставляемые модулем `linecache` функции описаны ниже.

checkcache `checkcache()`

Гарантирует, что кеш-память будет отражать текущее состояние системы, а не хранить устаревшие данные. Вызывайте функцию `checkcache` в тех случаях, когда хотите проверить, что прочитанные данные не изменились в файловой системе, ради уверенности в том, что последующие вызовы для получения строк будут возвращать обновленную информацию.

clearcache `clearcache()`

Освобождает занимаемую кешем модуля память, чтобы ее можно было использовать для других целей. Вызывайте функцию `clearcache` в тех случаях, когда вам известно, что в течение некоторого времени операции с кешем выполняться не будут.

getline `getline(`*`filename, lineno`*`)`

Читает (включая завершающие символы `\n`) и возвращает строку с номером *`lineno`* (нумерация строк начинается со значения `1`, а не `0`, как это принято в Python) из текстового файла *`filename`*. При возникновении любых ошибок функция `getline` не возбуждает исключение, а возвращает пустую строку `''`. Если файл с указанным именем не удается найти, то функция `getline` ищет его в каталогах, перечисленных в `sys.path` (игнорируя ZIP-файлы).

getlines `getlines(`*`filename`*`)`

Читает и возвращает все строки из текстового файла с указанным именем в виде списка строк, каждая из которых включает завершающий символ `\n`. При возникновении любых ошибок функция `getlines` не возбуждает исключение, а возвращает пустой список `[]`. Если файл с указанным именем не удается найти, то функция `getlines` ищет его в каталогах, перечисленных в `sys.path`.

Модуль `struct`

Модуль `struct` позволяет упаковать двоичные данные в байтовую строку, которую впоследствии можно будет распаковать для преобразования в исходные данные. Такие операции полезны для многих целей в низкоуровневом программировании. Чаще всего вы будете использовать модуль `struct` для интерпретации данных из двоичных файлов, записанных с использованием определенного формата, или для подготовки данных к записи в двоичные файлы. Название этого модуля происходит от ключевого слова `struct` языка программирования C, которое используется для аналогичных целей. При возникновении любой ошибки функции этого модуля возбуждают исключения, являющиеся экземплярами класса `struct.error` — единственного класса, предоставляемого данным модулем.

Работа модуля `struct` основана на *строках форматирования структуры*, в которых используется специальный синтаксис. Первый символ форматной строки задает порядок байтов, размер и выравнивание упакованных данных в соответствии с приведенным ниже описанием.

`@`

Нативные (присущие данной платформе) порядок байтов, размеры и выравнивание данных. Этот режим используется по умолчанию, если ни один из перечисленных ниже символов не указан в качестве первого (обратите внимание на то, что формат `P` в табл. 10.2 доступен только для этого вида строк форматирования структуры). Чтобы узнать порядок байтов, используемый вашей платформой, проверьте значение `sys.byteorder` (`'little'` или `'big'`).

=

Нативный порядок байтов текущей платформы, но стандартные размеры и выравнивание данных.

<

Порядок байтов от младшего к старшему (как на платформах Intel); стандартные размеры и выравнивание данных.

>, !

Порядок байтов от старшего к младшему (сетевой стандарт); стандартные размеры и выравнивание данных.

Стандартные размеры типов данных указаны в табл. 10.2. Стандартное выравнивание означает отсутствие принудительного выравнивания и явное использование заполняющих байтов по мере необходимости. Под нативными размерами типов данных и выравниванием понимаются параметры, используемые компилятором C, установленным на платформе. Нативный порядок байтов может поместить самый значимый байт либо в ячейку с меньшим адресом области памяти, занимаемой элементом данных (порядок байтов от старшего к младшему, "big-endian"), либо в ячейку с большим адресом (порядок байтов от младшего к старшему, "little-endian"), в зависимости от платформы.

После необязательного первого символа в строке формата указывается один или несколько символов формата, каждому из которых может предшествовать счетчик (целое число, представленное десятичными цифрами). Символы формата приведены в табл. 10.2. Для большинства символов формата значение счетчика указывает на количество повторений (например, `'3h'` — это то же самое, что `'hhh'`). Если символом формата является `s` или `p` (оба они определяют байтовую строку), то значение счетчика указывает на общее количество байтов в строке. Между символами формата, но только не между счетчиком и его символом формата, могут свободно располагаться пробельные символы.

Таблица 10.2. Форматные символы для `struct`

Символ	Тип C	Тип Python	Стандартный размер
B	unsigned char	int	1 байт
b	signed char	int	1 байт
c	char	bytes	1 байт
d	double	float	8 байт
f	float	float	4 байта
H	unsigned short	int	2 байта
h	signed short	int	2 байта
I	unsigned int	long	4 байта
i	signed int	int	4 байта

Символ	Тип C	Тип Python	Стандартный размер
L	unsigned long	long	4 байта
l	signed long	int	4 байта
P	void*	int	Не определено
p	char[]	bytes	Не определено
s	char[]	bytes	Не определено
x	padding byte	Отсутствует	1 байт

Формат s означает байтовую строку фиксированной длины, определяемой значением ее счетчика (строки Python усекаются или дополняются нулевыми байтами b'\0' в соответствии с необходимостью). Формат p означает байтовую строку "в стиле Pascal": первый байт содержит количество последующих значимых байтов, а фактическое содержимое начинается со второго байта. Значением счетчика является общее количество байтов, включая байт длины.

Предоставляемые модулем struct функции описаны ниже.

`calcsize` `calcsize(`*fmt*`)`

Возвращает размер в байтах, соответствующий строке формата *fmt*.

`pack` `pack(`*fmt*`, *`*values*`)`

Упаковывает значения в соответствии со строкой формата *fmt* и возвращает результирующую байтовую строку. Аргумент *values* должен соответствовать строке формата *fmt* по количеству и типу значений.

`pack_into` `pack_into(`*fmt*`, `*buffer*`, `*offset*`, *`*values*`)`

Упаковывает значения в соответствии со строкой формата *fmt* в записываемый буфер *buffer* (обычно экземпляр bytearray), начиная с индекса *offset*. Аргумент *values* должен соответствовать строке формата *fmt* по количеству и типу значений. Значение len(*buffer*[*offset*:]) должно быть бо́льшим или равным значению struct.calcsize(*fmt*).

`unpack` `unpack(`*fmt*`, `*s*`)`

Распаковывает байтовую строку *s* в соответствии со строкой формата *fmt* и возвращает кортеж значений (если имеется всего одно значение, то ему соответствует кортеж из одного элемента). Значение len(*s*) должно совпадать со значением struct.calcsize(*fmt*).

`unpack_from` `unpack_from(`*fmt*`, `*s*`, `*offset*`)`

Распаковывает байтовую строку (или другой читаемый буфер) *s*, начиная со смещения *offset*, в соответствии со строкой форматирования *fmt*, и возвращает кортеж значений (если имеется всего одно значение, то ему соответствует кортеж из одного элемента). Значение len(*s*[*offset*:]) должно быть бо́льшим или равным значению struct.calcsize(*fmt*).

Файлы в памяти: функции `io.StringIO` и `io.BytesIO`

Объекты, подобные файлам (файловые объекты), нетрудно реализовать самостоятельно, написав классы Python, предоставляющие необходимые методы. Если все, что вам нужно, — это данные, хранящиеся в памяти, то для этого можно использовать классы `StringIO` и `BytesIO` модуля `io`. Различаются эти классы тем, что экземпляры `StringIO` представляют текстовые "файлы", а потому операции чтения и записи, выполняемые с их помощью, обрабатывают строки Unicode, тогда как экземпляры `BytesIO` представляют байтовые строки.

При создании экземпляра любого из этих классов ему можно передать необязательный строковый аргумент, представляющий соответственно строку Unicode или байтовую строку, которая будет использована в качестве начального содержимого "файла". В дополнение к методам "файла" экземпляр *f* любого из этих классов предоставляет еще один метод, описание которого приведено ниже.

`getvalue` *f*`.getvalue()`

Возвращает текущее содержимое *f* в виде строки (текстовой или байтовой). После закрытия "файла" посредством вызова *f*`.close` выполнение вызова метода *f*`.getvalue` становится невозможным: метод `close` освобождает буфер, поддерживаемый *f*, в то время как метод `getvalue` должен вернуть в качестве результата полное содержимое этого буфера.

Сжатые файлы

В наши дни хранилища данных огромного объема и высокоскоростные соединения не являются редкостью, однако во многих случаях можно сэкономить ресурсы за счет сжатия данных. Вычислительные мощности постоянно дешевеют и растут быстрее, чем, скажем, предлагаемая пропускная способность сетей, поэтому сжатие данных приобретает все бо́льшую популярность. Python упрощает его поддержку программными средствами и предлагает несколько модулей, специально предназначенных для этих целей.

Учитывая тот факт, что Python предлагает довольно большое количество разнообразных способов сжатия данных, имеет смысл подробнее остановиться на этой теме. Файлы, сжатые средствами модуля `zlib`, не обеспечивают автоматический обмен данными с другими программами, за исключением файлов, созданных с помощью модуля `zipfile`, поддерживающего стандартный формат архивных ZIP-файлов. Для чтения файлов, созданных программой на Python с помощью модуля `zlib`, можно написать специальные программы на любом языке программирования, позволяющем использовать разработанную группой InfoZip свободную библиотеку сжатия *zlib*. Но если вы планируете обмениваться данными с программами, написанными на других языках, и при этом иметь возможность выбирать метод сжатия, то мы

рекомендуем вместо этого использовать модули bzip2 (наилучший вариант), gzip или zipfile. Однако модуль zlib может быть полезным, если требуется сжатие некоторой части файлов данных, подготовленных с использованием собственного нестандартного формата, и вы не собираетесь обмениваться ими с другими программами, кроме тех, которые входят в состав вашего приложения.

В версии v3 можно также использовать новый модуль lzma, обеспечивающий улучшенную степень сжатия и лучшую совместимость с новой утилитой xz. В этой книге модуль lzma не рассматривается. Более подробно о нем можно прочитать в онлайн-документации (https://docs.python.org/3/library/lzma.html?highlight=lzma#module-lzma), а информацию о том, как использовать его в версии v2, можно получить на сайте PiPY (https://pypi.python.org/pypi/backports.lzma).

Модуль gzip

Модуль gzip позволяет читать и записывать файлы, которые совместимы с форматами, обрабатываемыми более мощными программами сжатия GNU gzip и gunzip. Программы GNU поддерживают многие форматы сжатия, но модуль gzip поддерживает только формат *gzip*, часто обозначаемый присоединением расширения *.gz* к имени файла. Модуль gzip предоставляет класс GzipFile и функцию-фабрику open.

GzipFile class GzipFile(*filename*=None, *mode*=None, *compresslevel*=9, *fileobj*=None)

Создает и возвращает файловый объект *f*, который служит оберткой для "файла" или файлового объекта *fileob*. Если *fileobj* имеет значение None, то аргумент *filename* должен быть строкой, содержащей имя файла. Функция GzipFile открывает этот файл в режиме *mode* (по умолчанию 'rb') и упаковывает результирующий файл в объект *f*.

Аргумент *mode* может принимать значения 'ab', 'rb', 'wb' или None. Если *mode* имеет значение None, то объект *f* использует режим объекта *fileobj*, при условии, что может определить этот режим; в противном случае используется режим 'rb'. Если аргумент *filename* имеет значение None, то объект *f* использует имя файла *fileobj*, при условии, что может определить это имя; в противном случае используется пустая строка ''. Аргумент *compresslevel* — это целое число в диапазоне от 1 до 9. Значение 1 запрашивает самую низкую степень сжатия, но обеспечивает самое высокое быстродействие; значение 9 запрашивает самую высокую степень сжатия, но это достигается за счет более интенсивных вычислений, снижающих производительность.

Файловый объект *f* делегирует вызовы большинства методов лежащему в его основе файловому объекту *fileobj*, прозрачно учитывая степень сжатия по мере необходимости. Однако *f* не допускает произвольного доступа и потому не должен предоставлять методы seek и tell. Вызов метода *f*.close не закрывает *fileobj*, если *f* был создан со значением *fileobj*, не равным None. Это особенно важно, если *fileobj* является экземпляром io.BytesIO: вызвав метод

fileobj.getvalue после вызова *f*.close, вы получите строку сжатых данных. Поэтому после вызова метода *f*.close необходимо всегда вызывать метод *fileobj*.close (явно или неявно посредством использования инструкции with).

open open(*filename*, *mode*='rb', *compresslevel*=9)

Аналогичен вызову GzipFile(*filename*, *mode*, *compresslevel*), но задание имени файла является обязательным, а передача уже открытого объекта *fileobj* не предусмотрена. Только в версии v3 аргументом *filename* может быть имя уже открытого объекта "файла". Кроме того, в версии v3 режим *mode* может быть текстовым (например, 'rt'), и в этом случае вызов open возвращает "файл", обернутый экземпляром io.TextIOWrapper.

Пример использования модуля gzip

Предположим, у вас есть функция *f*(*x*), записывающая текст Unicode в объект текстового файла *x*, передаваемый в качестве аргумента, посредством вызова методов *x*.write и/или *x*.writelines. Функцию *f* можно использовать для того, чтобы записывать текст в сжатый *gzip*-файл с помощью следующего кода.

```
import gzip, io
with io.open('x.txt.gz', 'wb') as underlying:
    with gzip.GzipFile(fileobj=underlying, mode='wb') as wrapper:
        f(io.TextIOWrapper(wrapper, 'utf8'))
```

В этом примере сначала открывается базовый двоичный файл *x.txt.gz*, который затем помещается в оболочку gzip.GzipFile. Таким образом, нам нужны две вложенные инструкции with. Такое разделение не является строго необходимым: мы могли бы передать имя файла непосредственно во время вызова gzip.GzipFile (или gzip.open). Только в версии v3, используя вызов gzip.open, мы даже могли бы запросить режим mode='wt' и прозрачно получить готовый экземпляр TextIOWrapper. Однако представленный вариант кода наиболее понятен и работает в обеих версиях, v2 и v3.

Чтение сжатого текстового файла — например, для его отображения в стандартном устройстве вывода — осуществляется с помощью кода, имеющего аналогичную структуру.

```
import gzip, io
with io.open('x.txt.gz', 'rb') as underlying:
    with gzip.GzipFile(fileobj=underlying, mode='rb') as wrapper:
        for line in wrapper:
            print(line.decode('utf8'), end='')
```

Здесь нельзя просто использовать вызов io.TextIOWrapper, поскольку в версии v2 полученный объект не был бы итерируемым по строкам. В то же время явное декодирование каждой строки нормально работает и в версии v2, и в версии v3.

Модуль bz2

Модуль bz2 позволяет читать и записывать файлы, совместимые с форматами программ сжатия bzip2 и bunzip2, которые зачастую обеспечивают еще более высокую степень сжатия, чем программы gzip и gunzip. Модуль bz2 предоставляет класс BZ2File, обеспечивающий прозрачное сжатие и распаковку файлов, а также функции compress и decompress, выполняющие операции сжатия и распаковки строк данных в памяти. Он также предоставляет объекты, предназначенные для инкрементного (отдельными порциями) сжатия и распаковки данных, что позволяет работать с потоками данных, размер которых слишком велик для того, чтобы разместить в памяти сразу все данные без ущерба для производительности. Для получения более подробных сведений относительно последней возможности обратитесь к онлайн-документации Python (https://docs.python.org/3.5/library/bz2.html#incremental-de-compression).

Аналогичного расширения функциональности версии v2 можно добиться за счет использования стороннего модуля bz2file, который предлагает более полный набор возможностей, соответствующих аналогичным возможностям стандартной библиотеки версии v3. Краткое описание компонентов модуля bz2file приводится ниже.

BZ2File `class BZ2File(`*filename*`=None, `*mode*`='r', `*buffering*`=0, `*compresslevel*`=9)`

Создает и возвращает файловый объект *f*, который соответствует сжатому с помощью программы bzip2 файлу, определяемому аргументом *filename*. Этот аргумент должен быть строкой, обозначающей путь к файлу (только в версии v3 *filename* также может быть открытым объектом файла). Аргумент *mode* может принимать значение 'r' для чтения и 'w' для записи (только в версии v3 также допускаются значения 'a' — присоединение данных и 'x' — исключительно для открытия вновь создаваемого файла, аналогично 'w', но если файл уже существует, то возбуждается исключение; кроме того, допускается присоединение буквы 'b' к строке *mode*, поскольку результирующий файловый объект *f*, является двоичным).

Аргумент *buffering* признан устаревшим, поэтому не передавайте его. Аргумент *compresslevel* — это целое число в диапазоне от 1 до 9. Значение 1 запрашивает самую низкую степень сжатия, но обеспечивает самое высокое быстродействие. Значение 9 запрашивает самую высокую степень сжатия, но это достигается за счет более интенсивных вычислений, снижающих скорость выполнения вычислений.

Объект *f* предоставляет все методы объектов "файлов", включая методы seek и tell. Таким образом, *f* допускает поиск, однако операция seek эмулируется, и хотя ее семантическая корректность гарантирована, эта операция в некоторых случаях может выполняться очень медленно.

compress `compress(`*s*`, `*level*`=9)`

Сжимает строку *s* и возвращает строку сжатых данных. Аргумент *level* — это целое число в диапазоне от 1 до 9. Значение 1 запрашивает самую

низкую степень сжатия, но обеспечивает самое высокое быстродействие. Значение 9 запрашивает самую высокую степень сжатия, но это достигается за счет более интенсивных вычислений, снижающих производительность.

decompress `decompress(`*s*`)`

Распаковывает строку сжатых данных *s* и возвращает строку распакованных данных.

Модуль tarfile

Модуль `tarfile` позволяет читать и записывать TAR-файлы (архивные файлы, совместимые с форматами, обрабатываемыми такими популярными программами архивирования, как `tar`) с возможностью использования сжатия `gzip` или `bzip2` (и, только в версии v3, `lzma`). Только в версии v3 команда **`python -m tarfile`** предлагает удобный интерфейс командной строки для работы с данным модулем: необходимые справочные сведения можно получить, выполнив эту команду без дополнительных аргументов.

При попытке обработать файлы, не являющиеся файлами TAR, функции модуля `tarfile` возбуждают экземпляры исключения `tarfile.TarError`. Функции, предоставляемые модулем `tarfile`, описаны ниже.

is_tarfile `is_tarfile(`*filename*`)`

Возвращает значение `True`, если файл, имя которого задано строкой *filename*, является TAR-файлом (возможно, сжатым), судя по нескольким первым байтам; в противном случае возвращается значение `False`.

TarInfo `class TarInfo(`*name*`='')`

Методы `getmember` и `getmembers` экземпляров класса `TarFile` возвращают экземпляры `TarInfo`, предоставляющие информацию об элементах архива. Экземпляр `TarInfo` также можно создать с помощью метода `gettarinfo` экземпляра `TarFile`. Наиболее полезными атрибутами, предоставляемыми экземпляром *t* класса `TarInfo`, являются следующие.

`linkname`

Строка, содержащая имя файла, если *t*`.type` имеет значение `LNKTYPE` или `SYMTYPE`.

`mode`

Права доступа и другие биты режима файла, идентифицированного объектом *t*.

`mtime`

Время последнего изменения файла, идентифицированного объектом *t*.

`name`

Имя в архиве файла, идентифицированного объектом *t*.

`size`

Размер в байтах (несжатого) файла, идентифицированного объектом *t*.

`type`

Тип файла: одна из многочисленных констант, являющихся атрибутами модуля `tarfile` (`SYMTYPE` — символические ссылки, `REGTYPE` — обычные файлы, `DIRTYPE` — каталоги и др.).

Чтобы проверить тип, вместо тестирования значения `t.type` можно вызывать методы *t*. Наиболее часто используемыми методами *t* являются следующие.

`t.isdir()`

Возвращает значение `True`, если файл является каталогом.

`t.isfile()`

Возвращает значение `True`, если файл является обычным файлом.

`t.issym()`

Возвращает значение `True`, если файл является символической ссылкой.

open `open(`*filename*`, `*mode*`='r', `*fileobj*`=None, `*bufsize*`=10240, **`*kwargs`*`)`

Создает и возвращает экземпляр *f* класса `TarFile` для чтения и создания TAR-файла посредством файлового объекта *fileobj*. Если аргумент *fileobj* имеет значение `None`, то аргумент *filename* должен быть строкой, содержащей имя файла. Метод `open` открывает файл в режиме, определяемом аргументом *mode* (по умолчанию `'r'`), а *f* обертывает результирующий объект. Вызов `f.close` не приводит к закрытию *fileobj*, если *f* был открыт с использованием значения аргумента *fileobj*, не равного `None`. Такое поведение метода `f.close` имеет важное значение, если *fileobj* является экземпляром `io.BytesIO`: вызвав метод *fileobj*`.getvalue` после вызова метода `f.close`, вы сможете получить архивированные и, возможно, сжатые данные в виде строки. Это поведение также означает, что после вызовов метода `f.close` следует явно вызывать метод *fileobj*`.close`.

Аргумент *mode*, задающий режим, может принимать следующие значения: `'r'` — для чтения существующего TAR-файла независимо от схемы сжатия (если он был сжат), `'w'` — для записи нового TAR-файла или усечения и перезаписи существующего без сжатия, `'a'` — для присоединения к существующему TAR-файлу без сжатия. Присоединение к сжатым TAR-файлам не поддерживается. Для записи нового TAR-файла со сжатием аргумент *mode* может принимать следующие значения: `'w:gz'` — со сжатием `gzip`, `'w:bz2'` — со сжатием `bzip2`. Для чтения или запяиси несжатых, не допускающих поиска TAR-файлов с использованием буфера размером *bufsize* байт можно использовать специальные строки *mode* со значениями `'r:'` или `'w:'`, а для чтения таких файлов со сжатием — со значениями `'r:gz'` и `'r:bz2'`. Только в версии v3 для указания схемы сжатия LZMA можно использовать также значения `'r:xz'` и `'w:xz'`.

В строках, задающих режим, можно использовать вертикальную черту (|) вместо двоеточия (:) для задания принудительной последовательной обработки и блоков фиксированного размера (ценная возможность, если вам когда-либо придется работать с ленточными устройствами).

Экземпляр *f* класса `TarFile` предоставляет следующие методы.

add *f*.add(*filepath*, *arcname*=None, *recursive*=True)

Добавляет в архив *f* файл, определяемый аргументом *filepath* (это может быть обычный файл, каталог или символическая ссылка). Если значением аргумента *arcname* не является `None`, то оно используется в качестве альтернативного имени файла в архиве вместо *filepath*. Если *filepath* — каталог, то метод `add` рекурсивно добавляет все поддерево файловой системы, корнем которого служит данный каталог, если только для аргумента *recursive* не было задано значение `False`.

addfile *f*.addfile(*tarinfo*, *fileobj*=None)

Добавляет в архив *f* элемент архива, определяемый аргументом `tarinfo`, который представляет собой экземпляр класса `TarInfo` (если значение *fileobj* не равно `None`, то данными являются первые *tarinfo.size* байт файлового объекта *fileobj*).

close *f*.close()

Закрывает архив *f*. Следует обязательно вызывать метод `close`, иначе на диске может остаться неполный, непригодный для использования TAR-файл. Такие обязательные завершающие операции лучше всего выполнять с помощью инструкции `try/finally` (см. раздел "Инструкция `try/finally`" в главе 5) или, что еще лучше, с помощью инструкции `with` (см. раздел "Инструкция `try/except/finally`" в главе 5).

extract *f*.extract(*member*, *path*='.')

Извлекает элемент архива, заданный аргументом *member* (имя или экземпляр `TarInfo`), в соответствующий файл, расположенный в каталоге *path* (по умолчанию — в текущем каталоге).

extractfile *f*.extractfile(*member*)

Извлекает элемент архива, заданный аргументом *member* (имя или экземпляр `TarInfo`), и возвращает доступный только для чтения файловый объект, имеющий методы `read`, `readline`, `readlines`, `seek` и `tell`.

getmember *f*.getmember(*name*)

Возвращает экземпляр `TarInfo`, содержащий информацию об элементе архива с именем, заданным строковым аргументом *name*.

getmembers *f*.getmembers()

Возвращает список экземпляров `TarInfo`, по одному на каждый элемент архива *f*, в том же порядке, в каком они перечислены в архиве.

getnames *f*.getnames()

Возвращает список строк с именами элементов архива в том же порядке, в каком они перечислены в архиве.

gettarinfo *f*.gettarinfo(*name*=None, *arcname*=None, *fileobj*=None)

Возвращает экземпляр TarInfo, содержащий информацию об открытом объекте "файла" *fileobj*, если значение *fileobj* не равно None, или существующий файл с путем доступа, заданным строковым аргументом *name*. Если значение *arcname* не равно None, то оно используется в качестве имени атрибута результирующего экземпляра TarInfo.

list *f*.list(*verbose*=True)

Выводит перечень содержимого архива *f* в поток sys.stdout. Если значение необязательного аргумента *verbose* равно False, то выводятся только имена элементов архива.

Модуль zipfile

Модуль zipfile используется для чтения и записи ZIP-файлов (архивные файлы, которые совместимы с форматами, обрабатываемыми такими популярными программами сжатия, как zip и unzip, pkzip и pkunzip, WinZip и др.). Команда **python -m zipfile** предлагает удобный интерфейс командной строки для работы с данным модулем: необходимые справочные сведения можно получить, выполнив эту команду без дополнительных аргументов.

Подробную информацию о ZIP-файлах можно получить на сайтах *pkware* (https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT) и *info-zip* (http://www.info-zip.org/pub/infozip/). Ознакомление с ней необходимо для выполнения нестандартной обработки ZIP-файлов средствами модуля zipfile. Если у вас нет острой необходимости в организации взаимодействия с другими программами с использованием стандарта ZIP-файлов, то для работы со сжатыми файлами во многих случаях лучше использовать модули gzip и bz2.

Модуль zipfile не способен обрабатывать многодисковые ZIP-файлы и не может создавать зашифрованные ZIP-файлы (однако может расшифровывать такие архивы, хотя и делает это довольно медленно). Кроме того, он обеспечивает лишь архивирование несжатых данных или архивирование с использованием сжатия без потерь (алгоритм по умолчанию для сжатия файлов с использованием ZIP-формата). (Только в версии v3 модуль zipfile позволяет работать с алгоритмами сжатия *bzip2* и *lzma*, но пользоваться этой возможностью следует с осторожностью: не все инструменты, не говоря уже о модуле zipfile версии v2, способны обрабатывать такие файлы, поэтому в данном случае, достигая более высокой степени сжатия файлов, вы несколько теряете в их переносимости.) При возникновении любых ошибок, связанных с попытками обработки некорректных ZIP-файлов, функции модуля zipfile возбуждают исключения, являющиеся экземплярами класса исключения zipfile.error.

Описание классов и функций, предоставляемых модулем `zipfile`, приводится ниже.

is_zipfile `is_zipfile(file)`

Возвращает значение `True`, если файл (или файловый объект), заданный аргументом *file*, определен как корректный ZIP-файл на основании нескольких первых или последних байтов файла; в противном случае возвращается значение `False`.

ZipInfo `class ZipInfo(file='NoName', date_time=(1980, 1, 1))`

Методы `getinfo` и `infolist` экземпляров `ZipFile` возвращают экземпляры `ZipInfo`, содержащие информацию об элементах архива. Наиболее полезными атрибутами, предоставляемыми экземпляром *z* класса `ZipInfo`, являются следующие.

`comment`

Строка комментария, относящаяся к элементу архива.

`compress_size`

Размер сжатых данных элемента архива в байтах.

`compress_type`

Целочисленный код типа сжатия элемента архива.

`date_time`

Кортеж из шести целых чисел, представляющих следующую информацию о дате и времени последнего изменения файла: год, месяц, день (1 и выше), часы, минуты, секунды (0 и выше).

`file_size`

Размер несжатых данных элемента архива в байтах.

`filename`

Имя файла в архиве.

ZipFile `class ZipFile(file, mode='r', compression=zipfile.ZIP_STORED, allowZip64=True)`

Открывает ZIP-файл (или файловый объект), заданный строковым аргументом *file*. Аргумент *mode* может иметь следующие значения: `'r'` — чтение существующего ZIP-файла, `'w'` — запись нового ZIP-файла или усечение и перезапись существующего, `'a'` — присоединение данных в конец существующего файла. (Только в версии v3 может быть задан режим `'x'`, который подобен режиму `'w'`, но при попытке записи в уже существующий ZIP-файл возбуждается исключение.) В режиме `'a'` аргументом *filename* может быть либо имя существующего ZIP-файла (в этом случае новые элементы добавляются в существующий архив), либо имя существующего файла, не являющегося ZIP-архивом. В последнем случае создается новый архив, подобный ZIP-файлу, который присоединяетсяв конец существующего файла (например, *python.exe*). Это позволяет создавать самораспаковывающиеся исполняемые файлы, которые автоматически распаковываются при их запуске.

Аргумент *compression* — это целочисленный код в виде одного из атрибутов модуля zipfile. Значение zipfile.ZIP_STORED задает запись данных в архив без применения сжатия; значение zipfile.ZIP_DEFLATED задает применение сжатия без потерь (метод сжатия, наиболее часто используемый в ZIP-файлах). Только в версии v3 также возможно использование атрибутов bezipfile.ZIP_BZIP2 и zipfile.ZIP_LZMA (улучшают степень сжатия, но ухудшают степень переносимости файлов). Если аргумент *allowZip64* имеет значение True, то экземпляру ZipFile разрешается использовать расширение ZIP64, позволяющее создавать архивы размером свыше 4 Гбайт. В противном случае любая попытка создания архивов такого размера приводит к возбуждению исключения LargeZipFile. По умолчанию имеет значение True в версии v3 и False в версии v2.

Экземпляр *z* класса ZipFile имеет следующие атрибуты: compression и mode, соответствующие аргументам, используемым при инстанциализации *z*; fp и filename — файловый объект, с которым работает *z*, и имя файла, если оно известно; comment — возможно, пустая строка, которая служит комментарием к архиву, и filelist — список экземпляров ZipInfo в архиве.

Кроме того, *z* имеет записываемый атрибут debug, представляющий собой целое число в диапазоне от 0 до 3, которое управляет объемом отладочной информации, выводимой в стандартный поток sys.stdout. Значению *z*.debug, равному 0, соответствует подавление вывода информации, значению 3 — максимальный объем выводимой информации. Класс ZipFile является менеджером контекста. Таким образом, вы можете использовать его вместе с инструкцией with, тем самым гарантируя закрытие базового файла после того, как работа с ним будет завершена.

```
with ZipFile('archive.zip') as z:
    data = z.read('data.txt')
```

Экземпляр *z* класса ZipFile предоставляет следующие методы.

close *z*.close()

Закрывает архивный файл *z*. Не забывайте вызывать метод close, иначе на диске останется неполный ZIP-файл, непригодный для использования. Такие обязательные завершающие операции лучше всего выполнять с помощью инструкции try/finally (см. раздел "Инструкция try/finally" в главе 5) или, что еще лучше, с помощью инструкции with см. раздел "Инструкция try/except/finally" в главе 5).

extract *z*.extract(*member*, *path*=None, *pwd*=None)

Извлекает элемент архива и сохраняет его на диске в каталоге *path* или в используемом по умолчанию текущем рабочем каталоге. Аргумент *member* задает имя элемента архива или экземпляр ZipInfo. Метод extract нормализует информацию о пути в извлекаемом элементе, преобразуя абсолютные пути в относительные и удаляя компоненты '..', а в случае Windows — преобразуя символы, недопустимые в именах файлов, в символы подчеркивания (_). Аргумент *pwd* позволяет задать пароль, который будет использоваться для работы с зашифрованными элементами архива.

Возвращает путь к вновь созданному (или перезаписанному существующему) файлу или вновь созданному (или уже существующему) каталогу.

extractall `z.extractall(path=None, members=None, pwd=None)`

Извлекает элементы архива (по умолчанию — все) и сохраняет их на диске в каталоге `path` или в используемом по умолчанию текущем рабочем каталоге. Аргумент `members`, позволяющий ограничить состав извлекаемых элементов архива, должен представлять собой подмножество списка строк, возвращаемых вызовом метода `z.namelist()`. Метод `extractall` нормализует информацию о пути в извлекаемых элементах, преобразуя абсолютные пути в относительные и удаляя компоненты `'..'`, а в случае Windows — преобразуя символы, недопустимые в именах файлов, в символы подчеркивания (_). Аргумент `pwd` позволяет задать пароль, который будет использоваться для работы с зашифрованными элементами архива.

getinfo `z.getinfo(name)`

Возвращает экземпляр `ZipInfo`, предоставляющий информацию об элементе архива с именем, заданным строковым аргументом `name`.

infolist `z.infolist()`

Возвращает список экземпляров `ZipInfo`, по одному для каждого элемента архива `z`, в том порядке, в каком они перечислены в архиве.

namelist `z.namelist()`

Возвращает список строк, содержащих имена элементов архива `z`, в том порядке, в каком они перечислены в архиве.

open `z.open(name, mode='r', pwd=None)`

Извлекает и возвращает элемент архива, заданный аргументом `name` (строка с именем элемента или экземпляр `ZipInfo`) в качестве доступного только для чтения файлового объекта. Аргумент `mode` должен иметь значение `'r'`: в версии v2 и в ранних версиях v3 он также может иметь значение `'U'` или `'rU'` для получения режима "универсальных символов перевода строки", который в настоящее время признан устаревшим. Аргумент `pwd` позволяет задать пароль, который будет использоваться для работы с зашифрованными элементами архива.

printdir `z.printdir()`

Выводит в текстовом виде каталог `z` в поток `sys.stdout`.

read `z.read(name, pwd)`

Извлекает и возвращает элемент архива, заданный аргументом `name` (строка с именем элемента или экземпляр `ZipInfo`) и возвращает байтовую строку его содержимого. Аргумент `pwd` позволяет задать пароль, который будет использоваться для работы с зашифрованными элементами архива.

setpassword `z.setpassword(pwd)`

Задает строку `pwd` в качестве пароля по умолчанию для работы с зашифрованными файлами.

testzip `z.testzip()`

Читает и проверяет файлы, хранящиеся в архиве `z`. Возвращает строку с именем первого обнаруженного поврежденного элемента архива или значение `None`, указывающее на целостность архива.

write `z.write(filename, arcname=None, compress_type=None)`

Записывает файл, имя которого задано аргументом `filename`, в архив `z`, используя имя элемента архива, заданное аргументом `arcname`. Если аргумент `arcname` имеет значение `None`, записываемый элемент архива получает имя `filename`. Если аргумент `compress_type` имеет значение `None`, то элемент архива записывается с использованием типа сжатия, определенного для `z`; в противном случае аргумент `compress_type` задает тип сжатия. Архив `z` должен быть открыт в режиме `'w'` или `'a'`.

writestr `z.writestr(zinfo, bytes)`

Аргумент `zinfo` может иметь значение в виде экземпляра `ZipInfo`, в котором должны быть определены по крайней мере атрибуты `filename` и `date_time`, или в виде строки (которая в этом случае используется в качестве имени элемента архива, а время и дата создания элементов устанавливаются в соответствии с текущими значениями времени и даты). Аргумент `bytes` — это строка байтов. Метод `writestr` добавляет элемент в архив `z`, используя метаданные, определенные аргументом `zinfo`, и данные, определенные аргументом `bytes`. Экземпляр `z` должен быть открыт в режиме `'w'` или `'a'`. Если у вас есть данные, которые хранятся в памяти, и вы хотите записать их в ZIP-файл архива `z`, то использовать метод `z.writestr` будет проще и быстрее, чем метод `z.write`, так как последний потребует от вас сначала записать данные на диск, а затем удалить ненужный дисковый файл. В приведенном ниже примере продемонстрированы оба подхода, инкапсулированные во взаимно полиморфные функции.

```
import zipfile

def data_to_zip_direct(z, data, name):
    import time
    zinfo = zipfile.ZipInfo(name,
                            time.localtime()[:6])
    zinfo.compress_type = zipfile.ZIP_DEFLATED
    z.writestr(zinfo, data)

def data_to_zip_indirect(z, data, name):
    import os
    flob = open(name, 'wb')
    flob.write(data)
    flob.close()
    z.write(name)
    os.unlink(name)

with zipfile.ZipFile('z.zip', 'w',
zipfile.ZIP_DEFLATED) as zz:
```

```
data = 'four score\nand seven\nyears ago\n'
data_to_zip_direct(zz, data, 'direct.txt')
data_to_zip_indirect(zz, data, 'indirect.txt')
```

В отличие от функции `data_to_zip_indirect`, функция `data_to_zip_direct` не только работает быстрее и более компактна, но и удобнее в использовании, поскольку работает с данными в памяти и не требует, чтобы текущий рабочий каталог разрешал запись. Разумеется, методу `write` также находится применение, если данные хранятся в файле на диске и вы хотите просто добавить файл в архив.

В приведенном ниже примере сначала выводится список всех файлов, которые содержатся в ZIP-файле архива, созданного в предыдущем примере, а затем — имя и содержимое каждого файла.

```
import zipfile
zz = zipfile.ZipFile('z.zip')
zz.printdir()
for name in zz.namelist():
    print('{}: {!r}'.format(name, zz.read(name)))
zz.close()
```

Модуль `zlib`

Модуль `zlib` предоставляет программам на Python возможность доступа к свободной библиотеке *zlib* версии 1.1.4 и выше, содержащей набор средств для сжатия данных (`http://www.info-zip.org/pub/infozip/`). Модуль `zlib` используется модулями `gzip` и `zipfile`, но также доступен непосредственно для решения специальных задач сжатия данных. Перечень наиболее часто используемых функций, предоставляемых модулем `zlib`, приводится ниже.

`compress` `compress(s, level=6)`

Сжимает данные в строке *s* и возвращает строку сжатых данных. Аргумент *level* может принимать целочисленные значения в диапазоне от 1 до 9. Значение 1 запрашивает самую низкую степень сжатия, но обеспечивает самое высокое быстродействие. Значение 9 запрашивает самую высокую степень сжатия, но это достигается за счет более интенсивных вычислений, снижающих скорость выполнения вычислений.

`decompress` `decompress(s)`

Распаковывает сжатую байтовую строку *s* и возвращает байтовую строку несжатых данных. Также может принимать необязательные аргументы, предназначенные для опытных пользователей; см онлайн-документацию (`https://docs.python.org/3/library/zlib.html?highlight=zlib#zlib.decompress`).

Кроме того, модуль `zlib` предоставляет функции для вычисления CRC (циклически избыточного кода), позволяющего обнаруживать повреждение сжатых данных, а также объекты, обеспечивающие сжатие и распаковку данных порциями, что дает возможность работать с потоками данных большого объема, которые не могут

уместиться в памяти целиком. Для получения более подробной информации об этих более развитых средствах обратитесь к онлайн-документации Python (`https://docs.python.org/3/library/zlib.html`).

Модуль os

Модуль `os` — это зонтичный модуль, обеспечивающий в достаточной степени унифицированный кросс-платформенный интерфейс к часто используемым службам различных операционных систем. Он обеспечивает выполнение низкоуровневых операций над файлами и каталогами, а также создание, управление и уничтожение процессов. В этом разделе рассматриваются функции модуля `os`, относящиеся к файловой системе; функциям, связанным с управлением процессами, посвящен раздел "Выполнение других программ с помощью модуля `os`" в главе 14.

Модуль `os` имеет атрибут `name` — строку, идентифицирующую платформу, на которой выполняется Python. Чаще всего встречаются значения `'posix'` (все типы Unix-подобных платформ, включая Linux и macOS), `'nt'` (все типы 32-разрядных платформ Windows) и `'java'` (Jython). Предоставляемые модулем `os` функции обеспечивают использование специфических возможностей конкретной платформы. Однако в этой книге основное внимание уделено кросс-платформенному программированию, а не специфике отдельных платформ, поэтому ни компоненты `os`, которые существуют лишь на какой-то одной платформе, ни платформозависимые модули в данной книге обсуждаться не будут. Описанная ниже функциональность доступна по крайней мере на платформах `'posix'` и `'nt'`. Тем не менее там, где это уместно, будут делаться оговорки, касающиеся различий в способах предоставления той или иной функциональности на различных платформах.

Исключения `OSError`

Если не удается выполнить системный запрос, то модуль `os` возбуждает экземпляр исключения `OSError`. Кроме того, модуль `os` предоставляет встроенный класс исключения `OSError` с синонимом `os.error`. Экземпляры `OSError` предоставляют несколько полезных атрибутов, которые описаны ниже.

`errno`
: Числовой код системной ошибки.

`errorstrerror`
: Строка, содержащая краткое описание ошибки.

`filename`
: Имя файла, попытка выполнения операции над которым привела к возникновению ошибки (только для функций, связанных с файлами).

В версии v3 класс `OSError` имеет ряд подклассов, более точно соответствующих природе возникающих ошибок (см. раздел "Класс `OSError` и его подклассы (только в версии v3)" в главе 5).

Кроме того, функции модуля `os` могут возбуждать другие стандартные исключения, такие как `TypeError` или `ValueError`, если ошибка связана с недопустимым типом или значением аргумента, переданного функции, и возникла еще до попытки воспользоваться функциональностью базовой операционной системы.

Модуль `errno`

Модуль `errno` определяет несколько десятков символических имен для целочисленных кодов ошибок, возвращаемых системными вызовами. Их использование для избирательной обработки возможных системных ошибок позволяет улучшить переносимость и удобочитаемость программы. В приведенном ниже примере показано, как организовать обработку ошибок "file not found" ("файл не найден"), одновременно разрешив распространение других видов ошибок (этот вариант кода работает как в версии v2, так и в версии v3).

```
try: os.some_os_function_or_other()
except OSError as err:
    import errno
    # проверка на наличие ошибки "file not found" с повторным
    # возбуждением исключения в остальных случаях
    if err.errno != errno.ENOENT: raise
    # продолжение обработки интересующего вас исключения
    print('Предупреждение: файл', err.filename, 'не найден')
```

Но если вы планируете использовать этот код только в версии v3, то его можно значительно упростить и сделать более понятным, перехватывая лишь нужный подкласс исключения `OSError`.

```
try: os.some_os_function_or_other()
except FileNotFoundError as err:
    print('Предупреждение: файл', err.filename, 'не найден')
```

Модуль `errno` также предоставляет словарь `errorcode`, ключами которого являются коды ошибок, а именами — строки с именами ошибок, например `'ENOENT'`. Отображение строк `errno.errorcode[err.errno]` в диагностической информации, выводимой экземплярами класса `OSError`, часто позволяет сделать диагностические сообщения более понятными для читателей кода программы, работающих на какой-то одной конкретной платформе.

Операции в файловой системе

Модуль `os` позволяет выполнять различные операции над файловой системой: создавать, копировать и удалять файлы и каталоги, а также сравнивать файлы между

собой и просматривать информацию о файлах и каталогах, содержащуюся в файловой системе. В этом разделе описаны атрибуты и методы модуля `os`, предназначенные для выполнения данных операций, а также некоторые родственные модули, обеспечивающие работу с файловой системой.

Атрибуты модуля `os`, связанные с путем доступа

Для идентификации файла или каталога используется строка, называемая *путь доступа*, или просто *путь*, синтаксис которой зависит от платформы. Как на Unix-подобных, так и на Windows-платформах Python допускает указание пути доступа с помощью синтаксиса Unix, предполагающего использование символа косой черты (`/`) в качестве разделителя каталогов. На платформах, отличных от Unix, Python позволяет также использовать синтаксис указания пути, специфический для платформы. В частности, в Windows в качестве разделителя каталогов можно использовать символ обратной косой черты (`\`). Однако в этом случае для его представления в строковых литералах вам придется либо использовать два символа обратной косой черты (`\\`) вместо одного, либо прибегнуть к синтаксису "сырых" строк (см. раздел "Литералы" в главе 3), что чревато потерей переносимости программы. Синтаксис указания путей доступа, принятый в Unix, удобнее и применим на любой платформе, поэтому мы настоятельно рекомендуем всегда использовать именно его, что и предполагается в примерах и объяснениях, приведенных в оставшейся части главы.

Специальные атрибуты модуля `os` предоставляют подробную информацию о строках пути, используемых на текущей платформе. Как правило, вы будете манипулировать путями с помощью высокоуровневых операций (раздел "Модуль `os.path`"), а не низкоуровневых операций, использующих значения этих атрибутов. Тем не менее иногда атрибуты также могут быть полезными.

`curdir`
: Строка, обозначающая текущий каталог (`'.'` на платформах Unix и Windows).

`defpath`
: Используемый по умолчанию путь поиска для программ, если переменная среды `PATH` не определена.

`linesep`
: Признак окончания текстовой строки (`'\n'` в Unix; `'\r\n'` в Windows).

`extsep`
: Строка, отделяющая расширение от остальной части имени файла (`'.'` в Unix и Windows).

`pardir`
: Строка, обозначающая родительский каталог (`'..'` в Unix и Windows).

`pathsep`
: Разделитель путей в списках путей, например в переменной среды `PATH` (`':'` в Unix, `';'` в Windows).

`sep`
: Разделитель элементов пути (`'/'` в Unix, `'\\'` в Windows).

Права доступа

На Unix-подобных платформах с каждым файлом и каталогом связаны девять битов, по три на каждую категорию пользователей: владелец файла, его группа и прочие пользователи (как говорят, "все остальные"). Эти биты соответственно указывают, разрешено ли данному субъекту читать данные из файла, записывать данные в файл или выполнять файл (для каталогов определен несколько иной перечень операций, требующих разрешений). Указанные девять битов, известные как *биты разрешений*, являются частью *строки режима* файла (битовая строка, включающая дополнительные биты описания файла). Эти биты часто отображаются в восьмеричном представлении, в котором три бита группируются в одну цифру. Например, режим `0o664` означает, что владельцу файла и его группе разрешен доступ к файлу для чтения и записи, тогда как всем остальным пользователям разрешен только доступ для чтения, но не записи. Если в Unix-подобной системе некий процесс создает файл или каталог, то операционная система применяет к заданному режиму битовую маску (известную как *umask*), которая может отключить отдельные биты разрешений.

На платформах, отличных от Unix-подобных, обработка разрешений доступа к файлам и каталогам осуществляется разными способами. Однако функции модуля `os`, обрабатывающие разрешения доступа к файлам, принимают аргумент *mode*, который описан в предыдущем абзаце и используется Unix-подобными системами. Каждая платформа отображает девять битов разрешений удобным для нее способом. Например, в версиях Windows, которые различают лишь права доступа только для чтения и чтения/записи, но не права владения файлом, биты разрешений отображаются либо как `0o666` (чтение/запись), либо как `0o444` (только чтение). При создании файла на такой платформе реализация проверяет только бит `0o200`, делая данный файл доступным для чтения/записи, если этот бит равен `1`, или только для чтения, если этот бит равен `0`.

Функции модуля `os` для работы с файлами и каталогами

Модуль `os` предлагает несколько функций, позволяющих запрашивать и устанавливать состояние файлов и каталогов (табл. 10.3). Во всех версиях и на всех платформах любой из этих функций можно передать в качестве аргумента *`path`* строку, содержащую путь к файлу или каталогу. На некоторых Unix-платформах отдельные функции в версии v3 дополнительно поддерживают аргумент *`path`* в виде *дескриптора файла* — целого числа, идентифицирующего файл (например, возвращаемого

функцией os.open). В этом случае атрибут os.supports_fd предоставляет набор функций модуля os, которые поддерживают дескриптор файла в качестве аргумента *path* (этот атрибут отсутствует в версии v2, а также в версии v3 на платформах, не поддерживающих такую возможность).

На некоторых Unix-платформах отдельные функции в версии v3 поддерживают необязательный, задаваемый как только именованный, аргумент *follow_symlinks*, который по умолчанию имеет значение True. Если в качестве аргумента *path* указана символическая ссылка и аргумент *follow_symlinks* имеет истинное значение, то функция использует эту ссылку для обращения к фактическому файлу или каталогу (в версии v2 функции всегда ведут себя подобным образом). В случае ложного значения указанного аргумента функция воздействует на саму символическую ссылку. Атрибут os.supports_follow_symlinks, если он существует, предоставляет набор функций модуля os, поддерживающих этот аргумент.

На некоторых Unix-платформах отдельные функции в версии v3 поддерживают необязательный, задаваемый как только именованный, аргумент *dir_fd* (дескриптор файла), который по умолчанию имеет значение None. Если этот аргумент имеется, то аргумент *path* (если он обозначает относительный путь) интерпретируется относительно каталога, открытого с помощью данного дескриптора файла. В случае отсутствия аргумента *dir_fd*, и всегда в версии v2, аргумент *path* (если он обозначает относительный путь) интерпретируется относительно текущего рабочего каталога (в версии v2 функции всегда ведут себя подобным образом). Атрибут os.supports_dir_fd, если он существует, представляет набор функций модуля os, поддерживающих этот аргумент.

Таблица 10.3. Функции модуля os

Функция	Описание
access	access(*path*, *mode*) Возвращает значение True, если файл *path* имеет все разрешения, запрашиваемые с помощью целочисленного аргумента *mode*; в противном случае возвращается значение False. В качестве аргумента *mode* можно указать константу os.F_OK для проверки существования файла или одну или несколько целочисленных констант os.R_OK, os.W_OK и os.X_OK (при необходимости используя оператор побитового ИЛИ для их объединения), позволяющих проверить наличие разрешений на доступ для чтения, записи и выполнения. Метод access не использует стандартную интерпретацию аргумента *mode*, описанную в разделе "Функции модуля os для работы с файлами и каталогами". Вместо этого он выполняет указанную проверку только в том случае, если идентификаторы реального пользователя и группы данного процесса имеют необходимые разрешения на доступ к файлу. Если вам необходимо получить более подробную информацию о битах разрешений доступа к файлу, воспользуйтесь функцией stat, описанной в табл. 10.4. В версии v3 метод access поддерживает необязательный, задаваемый только как именованный, аргумент effective_ids, который по умолчанию имеет значение False. Если этот аргумент

Функция	Описание
	имеет истинное значение, то метод `access` использует идентификаторы эффективных, а не фактических, пользователя и группы
chdir	`chdir(`*`path`*`)` Устанавливает *`path`* в качестве текущего рабочего каталога процесса
chmod	`chmod(`*`path`*`, `*`mode`*`)` Изменяет разрешения доступа к файлу *`path`* в соответствии с целочисленным аргументом *`mode`*. Аргумент *`mode`* можно указывать в виде нуля или нескольких констант `os.R_OK`, `os.W_OK` и `os.X_OK` (при необходимости используя оператор побитового ИЛИ для их объединения), устанавливающих права доступа для чтения, записи и выполнения соответственно. На Unix-подобных платформах *`mode`* может принимать значения в виде более широких сочетаний значений битов (см. раздел "Функции модуля `os` для работы с файлами и каталогами"), устанавливающих различные разрешения для владельца, группы и прочих пользователей, а также других специальных битов, определенных в модуле `stat` и перечисленных в онлайн-документации (`https://docs.python.org/3/library/os.html?#os.chmod`)
getcwd	`getcwd()` Возвращает строку типа `str`, представляющую путь к текущему рабочему каталогу. В версии v3 метод `getcwdb` возвращает то же значение в виде значения типа `bytes`; в версии v2 метод `getcwdu` возвращает то же значение в виде значения типа `unicode`
link	`link(`*`src`*`, `*`dst`*`)` Создает жесткую ссылку *`dst`*, указывающую на *`src`*. В версии v2 это доступно только на платформах Unix; в версии v3 эта возможность доступна также на платформах Windows
listdir	`listdir(`*`path`*`)` Возвращает список имен всех файлов и каталогов, содержащихся в каталоге *`path`*. Элементы перечисляются в произвольном порядке и не включают специальные имена каталогов `'.'` (текущий каталог) и `'..'` (родительский каталог). См. также доступную только в версии v3 аальтернативную функцию `scandir`, описанную далее в таблице, которая в некоторых случаях обеспечивает повышение производительности. В версии v2 модулем `dircache` предлагается одноименная функция `listdir`, которая работает аналогично функции `os.listdir`, но улучшена в двух отношениях: она возвращает отсортированный список и кеширует его, поэтому повторные запросы, относящиеся к одному и тому же каталогу, работают быстрее, если содержимое каталога не изменилось. Модуль `dircache` автоматически обнаруживает изменения: вызывая функцию `dircache.listdir`, вы получаете список содержимого, актуальный на момент вызова

Функция	Описание

makedirs, mkdir	`makedirs(path, mode=0777)`
	`mkdir(path, mode=0777)`
	Функция `makedirs` создает все каталоги, указанные в *path*, если они не существуют. Функция `mkdir` создает только самый последний вложенный каталог, указанный в `path`, и возбуждает исключение `OSError`, если любой из предыдущих каталогов, указанных в `path`, не существует. Обе функции используют *mode* в качестве битов разрешений создаваемых каталогов и возбуждают исключение `OSError`, если создать каталог не удается либо если файл или каталог с именем *path* уже существует

remove, unlink	`remove(path)`
	`unlink(path)`
	Удаляет файл *path* (относительно удаления каталога см. описание функции `rmdir` далее в таблице). `unlink` — синоним `remove`

removedirs	`removedirs(path)` Последовательно удаляет все каталоги, указанные в *path*, начиная с самого последнего вложенного каталога. Этот процесс прекращается, если попытка удаления каталога возбуждает исключение, что обычно случается, когда каталог не является пустым. Функция `removedirs` не распространяет исключение, если удалось удалить хотя бы один каталог
rename	`rename(source, dest)` Переименовывает (т.е. перемещает) файл или каталог *source* в *dest*. Если *dest* уже существует, то функция `rename` может либо заменить *dest*, либо возбудить исключение. Только в версии v3, чтобы гарантировать выполнение замены, а не возбуждение исключения, вызовите вместо функции `rename` функцию `os.replace`
renames	`renames(source, dest)` Подобна функции `rename`, за исключением того, что `renames` пытается создать все промежуточные каталоги, необходимые для *dest*. Кроме того, после выполнения переименования функция `renames` пытается удалить пустые каталоги из пути *source* с помощью функции `removedirs`. Функция `renames` не распространяет возникающие исключения: если начальный каталог *source* не становится пустым после переименования, то это не считается ошибкой
rmdir	`rmdir(path)` Удаляет пустой каталог *path* (возбуждает исключение `OSError`, если каталог не удается удалить и, в частности, если каталог не является пустым)
scandir	`scandir(path)` Возвращает итератор по экземплярам `os.DirEntry`, представляющим элементы *path*. Использование функции `scandir` и вызов методов каждого результирующего экземпляра для определения его

Функция	Описание
	характеристик может улучшить производительность по сравнению с использованием функций `listdir` и `stat`, в зависимости от базовой платформы. `class DirEntry` Экземпляр *d* класса `DirEntry` предоставляет строковые атрибуты `name` и `path`, в которых хранится базовое имя и полный путь доступа к элементу соответственно, и несколько методов, из которых наиболее часто используемыми, не принимающими аргументов и возвращающими булевы значения, являются методы `is_dir`, `is_file` и `is_symlink`. По умолчанию функции `is_dir` и `is_file` следуют символическим ссылкам: чтобы избежать этого поведения, функции следует передать аргумент `follow_symlinks=False`, который задается как только именованный. За более полной информацией обратитесь к онлайн-документации (`https://docs.python.org/3/library/os.html#os.scandir`). Экземпляр *d* по возможности избегает системных вызовов, а если это все-таки потребуется, то он кеширует результаты. Если вам нужна гарантированно обновленная информация, вызовите функцию `os.stat(d.path)` и используйте экземпляр `stat_result`, который она возвращает (при этом вы лишитесь возможного повышения производительности, предлагаемого функцией `scandir`)
`stat`	`stat(`*`path`*`)` Возвращает значение *x* типа `stat_result`, которое предоставляет 10 элементов информации о файле или подкаталоге *path*. Получение доступа к этим элементам по их числовым индексам возможно, но обычно не рекомендуется, поскольку это ухудшает читаемость результирующего кода. Лучше использовать имена соответствующих атрибутов. Атрибуты экземпляра `stat_result` кратко описаны в табл. 10.4.

Таблица 10.4. Элементы (атрибуты) экземпля ра `stat_result`

Индекс элемента	Имя атрибута	Описание
0	`st_mode`	Код защиты и другие биты режима
1	`st_ino`	Номер индексного узла (Inode)
2	`st_dev`	Идентификатор устройства
3	`st_nlink`	Количество жестких ссылок
4	`st_uid`	Идентификатор владельца файла
5	`st_gid`	Идентификатор группы владельца файла
6	`st_size`	Размер файла в байтах
7	`st_atime`	Время последнего обращения к файлу
8	`st_mtime`	Время последнего изменения файла
9	`st_ctime`	Время последнего изменения информации о состоянии файла

Функция	Описание

	Например, чтобы вывести размер файла `path` в байтах, можно использовать любую из следующих инструкций:
	`import os`
	`print(os.path.getsize(path))`
	`print(os.stat(path)[6])`
	`print(os.stat(path).st_size)`
	Значения времени (тип `int` на большинстве платформ) выводятся в виде количества секунд, истекших с момента наступления эпохи Unix (подробно об этом — в главе 12). Если платформа не может предоставить значение для элемента, то она использует фиктивное значение

tempnam, tmpnam	`tempnam(`*`dir`*`=None, `*`prefix`*`=None)`
	`tmpnam( )`
	Возвращает абсолютный путь, пригодный для использования в качестве имени нового временного файла.
	Примечание: функции `tempnam` и `tmpnam` — это бреши в безопасности программы. Избегайте их применения и используйте вместо них модуль `tempfile` стандартной библиотеки (см. раздел "Модуль `tempfile`")

utime	`utime(`*`path`*`, `*`times`*`=None)` Устанавливает время последнего обращения и последнего изменения файла или каталога *`path`*. Если значение *`times`* равно `None`, функция `utime` использует текущее время. В противном случае аргумент *`times`* должен быть задан в виде пары чисел (количества секунд, истекших с момента наступления эпохи Unix; см. главу 12), указываемых в следующей очередности: `(`*`время_обращения`*`, `*`время_изменения`*`)`
walk	`walk(`*`top`*`, `*`topdown`*`=True, `*`onerror`*`=None, `*`followlinks`*`=False)` Создает генератор, возвращающий элемент для каждого каталога в дереве, корнем которого является каталог *`top`*. Если *`topdown`* имеет значение `True`, используемое по умолчанию, функция `walk` обходит каталоги в направлении сверху вниз, начиная с каталога *`top`*; если *`topdown`* имеет значение `False`, функция `walk` обходит каталоги в направлении снизу вверх, начиная с листьев дерева. Если *`onerror`* имеет значение `None`, функция `walk` перехватывает и игнорирует любые исключения `OSError`, возникающие в процессе обхода дерева. В противном случае аргумент *`onerror`* должен быть функцией. Функция `walk` перехватывает любое исключение `OSError`, возникающее в процессе обхода дерева, и передает его в качестве единственного аргумента функции *`onerror`*, которая может обработать исключение, проигнорировать или возбудить его с помощью инструкции `raise`, чтобы прекратить обход дерева и распространить исключение.

Функция	Описание
	Каждый элемент, возвращаемый функцией `walk`, представляет собой кортеж из трех подэлементов: *`dirpath`* — строка пути к каталогу, *`dirnames`* — список имен подкаталогов, являющихся прямыми потомками данного каталога (исключая специальные каталоги `'.'` и `'..'`) и *`filenames`* — список имен файлов, находящихся непосредственно в каталоге. Если *`topdown`* имеет значение `True`, вы можете изменить список *`dirnames`* на месте, удалив одни элементы и/или переставив другие, чтобы воздействовать на обход поддерева с корнем в *`dirpath`*. Функция `walk` выполняет итерации только по подкаталогам, оставшимся в списке *`dirnames`*, в порядке их следования. Если *`topdown`* имеет значение `False`, то изменение списка не будет иметь никаких последствий (в этом случае функция `walk` уже успеет обойти все подкаталоги к тому моменту, когда дойдет до текущего каталога и вернет его элемент). По умолчанию функция `walk` не следует по символическим ссылкам, которые ведут в каталоги. Чтобы функция `walk` выполнила такой дополнительный обход, ей можно передать аргумент *`followlinks`* со значением `True`, но это может привести к возникновению циклических ссылок, если символическая ссылка ведет в каталог, являющийся ее предком, — функция `walk` не предпринимает никаких мер предосторожности в отношении подобных аномалий

Модуль `os.path`

Модуль `os.path` предоставляет функции, предназначенные для анализа и преобразования строк путей. Прежде чем использовать модуль `os.path`, его необходимо импортировать. Но даже если вы ограничитесь импортом модуля `os`, вы все равно будете иметь доступ к модулю `os.path` и всем его атрибутам. Наиболее часто используемые функции этого модуля приведены ниже.

`abspath` `abspath(`*`path`*`)`

Возвращает строку нормализованного абсолютного пути, эквивалентного пути *`path`*, аналогично следующей инструкции:

`os.path.normpath(os.path.join(os.getcwd(), `*`path`*`))`

Например, `os.path.abspath(os.curdir)` — это то же самое, что `os.getcwd()`.

`basename` `basename(`*`path`*`)`

Возвращает базовое имя в пути *`path`* аналогично выражению `os.path.split(`*`path`*`)[1]`. Например, `os.path.basename('b/c/d.e')` возвращает `'d.e'`.

`commonprefix` `commonprefix(`*`list`*`)`

Принимает список строк и возвращает самую длинную строку, которая является префиксом для всех элементов в списке. В отличие

от всех других функций в модуле `os.path`, функция `commonprefix` работает с любыми строками, а не только с путями. Например, вызов `os.path.commonprefix('foobar', 'foolish')` возвращает строку `'foo'`.

В версии v3 функция `os.path.commonpath` работает аналогичным образом, но возвращает только префикс, являющийся общим для строк *путей*, а не для произвольных строк.

dirname

`dirname(path)`

Возвращает часть пути, являющуюся именем каталога для базового имени в пути *path* аналогично выражению `os.path.split(path)[0]`. Например, `os.path.dirname('b/c/d.e')` возвращает строку `'b/c'`.

exists, lexists

`exists(path)`
`lexists(path)`

Возвращает значение `True`, если путь *path* соответствует существующему файлу или каталогу; в противном случае возвращается значение `False`. Другими словами, `os.path.exists(x)` — это то же самое, что `os.access(x, os.F_OK)`. Функция `lexists` аналогична предыдущей, но возвращает значение `True`, даже если аргумент *path* содержит символическую ссылку, указывающую на несуществующий файл или каталог (такие ссылки иногда называют *разорванными*), тогда как функция `exists` в подобных случаях возвращает значение `False`.

expandvars, expanduser

`expandvars(path)`
`expanduser(path)`

Возвращает копию строки *path*, в которой каждая подстрока вида *$name* или *${name}* заменена значением соответствующей переменной среды. Например, если для переменной среды `HOME` установлено значение `/u/alex`, то код

```
import os
print(os.path.expandvars('$HOME/foo/'))
```

вернет значение `/u/alex/foo/`.

Функция `os.path.expanduser` расширяет ведущие символы `~` до пути к домашнему каталогу текущего пользователя.

getatime, getmtime, getctime, getsize

`getatime(path)`
`getmtime(path)`
`getctime(path)`
`getsize(path)`

Каждая из этих функций возвращает из результата вызова `os.stat(path)` соответствующий атрибут: `st_atime`, `st_mtime`, `st_ctime` и `st_size`. Более подробная информация об этих атрибутах была приведена в табл. 10.4.

isabs `isabs(`*path*`)`

Возвращает значение `True`, если аргумент *path* содержит абсолютный путь. (Путь называется абсолютным, если он начинается с символа косой черты или, на некоторых платформах, не являющихся Unix-подобными, с обозначения диска, за которым следует символ `os.sep`.) Если путь *path* не является абсолютным, то функция `isabs` возвращает значение `False`.

isfile `isfile(`*path*`)`

Возвращает значение `True`, если аргумент *path* содержит путь к существующему обычному файлу (однако `isfile` также следует по символическим ссылкам); в противном случае возвращается значение `False`.

isdir `isdir(`*path*`)`

Возвращает значение `True`, если аргумент *path* содержит путь к существующему каталогу (однако `isfile` также следует по символическим ссылкам); в противном случае возвращается значение `False`.

islink `islink(`*path*`)`

Возвращает значение `True`, если аргумент *path* содержит путь к символической ссылке; в противном случае возвращается значение `False`.

ismount `ismount(`*path*`)`

Возвращает значение `True`, если аргумент *path* содержит путь к точке монтирования; в противном случае возвращается значение `False`.

join `join(`*path*`, *`*paths*`)`

Возвращает строку, объединяющую все строки аргументов, используя разделитель каталогов, соответствующий текущей платформе. Например, на платформе Unix разделителем каталогов служит одиночный символ косой черты (/). Если какой-либо из аргументов содержит абсолютный путь, то функция `join` игнорирует предыдущие компоненты. Например:

```
print(os.path.join('a/b', 'c/d', 'e/f'))
# в Unix выводит: a/b/c/d/e/f
print(os.path.join('a/b', '/c/d', 'e/f'))
# в Unix выводит: /c/d/e/f
```

Второй вызов `os.path.join` игнорирует первый аргумент `'a/b'`, поскольку второй аргумент представляет абсолютный путь.

normcase `normcase(`*path*`)`

Возвращает копию *path*, используя регистр символов, нормализованный применительно к данной платформе. В случае

файловых систем, нечувствительных к регистру (типично для Unix-подобных систем), аргумент *path* возвращается в неизмененном виде. В случае файловых систем, чувствительных к регистру (типично для систем Windows), все буквы в строке переводятся в нижний регистр. Кроме того, на платформах Windows все символы косой черты (/) преобразуются в символы обратной косой черты (\).

normpath `normpath(`*path*`)`

Возвращает нормализованный путь, эквивалентный *path*, одновременно удаляя избыточные символы разделителей и ссылки на родительские каталоги. Например, на платформах Unix функция `normpath` возвращает строку `'a/b'` для аргумента *path*, имеющего любое из следующих значений: `'a//b'`, `'a/./b'` или `'a/c/../b'`. Функция `normpath` использует разделители каталогов, соответствующие текущей платформе. Например, на платформах Windows разделителем становится символ `\`.

realpath `realpath(`*path*`)`

Возвращает фактический путь к указанному файлу или каталогу, попутно разрешая все имеющиеся символические ссылки.

relpath `relpath(`*path*`, `*start*`=os.curdir)`

Возвращает относительный путь к указанному файлу или каталогу, заданный относительно каталога *start* (которым по умолчанию является текущий рабочий каталог процесса).

samefile `samefile(`*path1*`, `*path2*`)`

Возвращает значение `True`, если оба аргумента ссылаются на один и тот же файл или каталог.

sameopenfile `sameopenfile(`*fd1*`, `*fd2*`)`

Возвращает значение `True`, если оба аргумента, являющиеся дескрипторами файлов, ссылаются на один и тот же открытый файл или каталог.

samestat `samestat(`*stat1*`, `*stat2*`)`

Возвращает значение `True`, если оба аргумента, которые представляют экземпляры `os.stat_result` (обычно являющиеся результатами вызовов `os.stat`), ссылаются на один и тот же файл или каталог.

split `split(`*path*`)`

Возвращает пару строк `(`*dir*`, `*base*`)`, таких, что результат вызова `join(`*dir*`, `*base*`)` равен *path*, где *base* — последний компонент пути, не содержащий символа разделителя. Если строка *path* заканчивается разделителем, то *base* становится пустой строкой `''`. Строка *dir* — начальная часть пути, предшествующая последнему разделителю, из которой удалены замыкающие разделители. Например, вызов `os.path.split('a/b/c/d')` возвращает `('a/b/c', 'd')`.

splitdrive	splitdrive(*path*) Возвращает пару строк (*drv*, *pth*), таких, что строка *drv*+*pth* равна строке *path*. Строка *drv* содержит либо спецификацию диска, либо пустую строку '', которая всегда возвращается на таких платформах, не поддерживающих спецификации дисков, как Unix-подобные системы. На платформах Windows вызов os.path.splitdrive('c:d/e') вернет кортеж ('c:', 'd/e').
splitext	splitext(*path*) Возвращает пару (*root*, *ext*), такую, что строка *root*+*ext* равна строке *path*. Строка *ext* либо является пустой строкой '', либо начинается с символа '.' и не содержит других символов '.' или разделителей каталогов. Например, вызов os.path.splitext('a.a/b.c.d') возвращает кортеж ('a.a/b.c', '.d').
walk	walk(*path*, *func*, *arg*) (Только в версии v2.) Вызывает функцию func(*arg*, *dirpath*, *namelist*) для каждого каталога в дереве, корень *root* которого является каталогом *path*, начиная с самого каталога *path*. Эта функция трудна в использовании и признана устаревшей; вместо нее (в обеих версиях, v2 и v3) лучше использовать генератор os.walk, описанный в табл. 10.4.

Модуль stat

Функция os.stat (см. табл. 10.3) возвращает экземпляры stat_result, краткое описание атрибутов которых приведено в табл. 10.4. Модуль stat предоставляет атрибуты с именами, подобными именам атрибутов экземпляров stat_result и преобразованными в верхний регистр, и значениями, соответствующими значениям индексов элементов.

Более интересной частью содержимого модуля stat являются функции, позволяющие проверять значение атрибута st_mode экземпляра stat_result и определять разновидность файла. Модуль os.path предоставляет аналогичные функции, которые работают непосредственно с путями к файлам. Функции, предлагаемые модулем stat, список которых приводится ниже, работают быстрее в случае выполнения нескольких проверок в отношении одного и того же файла: для этого требуется всего лишь один вызов функции os.stat, выполняемый в начале серии проверок, тогда как функции, предоставляемые модулем os.path, неявно запрашивают у системы аналогичную информацию при каждой проверке. Каждая из приведенных ниже функций возвращает значение True, если режим, определяемый аргументом *mode*, соответствует данной разновидности файла; в противном случае возвращается значение False.

S_ISDIR(*mode*)
: Является ли файл каталогом?

S_ISCHR(*mode*)
: Является ли файл специальным файлом символьного устройства?

S_ISBLK(*mode*)
: Является ли файл специальным файлом блочного устройства?

S_ISREG(*mode*)
: Является ли файл обычным файлом (а не каталогом, специальным файлом устройства и т.п.)?

S_ISFIFO(*mode*)
: Работает ли файл по принципу FIFO (такие файлы также известны как "именованные каналы")?

S_ISLNK(*mode*)
: Является ли файл символической ссылкой?

S_ISSOCK(*mode*)
: Является ли файл сокетом Unix-домена?

Некоторые из этих функций имеют смысл только в Unix-подобных системах, ввиду того, что другие платформы не хранят специальные файлы, такие как устройства или сокеты, в пространстве имен обычных файлов, как это делается в Unix-подобных системах.

Модуль `stat` также предоставляет две функции, которые извлекают соответствующие части режима *mode* файла (`x.st_mode` для некоторых результатов *x* вызова `os.stat`).

S_IFMT S_IFMT(*mode*)
: Возвращает биты *mode*, описывающие разновидность файла (т.е. биты, которые проверяются функциями `S_ISDIR`, `S_ISREG` и др.).

S_IMODE S_IMODE(*mode*)
: Возвращает биты *mode*, которые можно устанавливать с помощью функции `os.chmod` (т.е. биты разрешений и, на Unix-подобных платформах, несколько других специальных битов, таких как флаг `set-user-id`).

Модуль `filecmp`

Модуль `filecmp` предоставляет функции, позволяющие сравнивать файлы и каталоги, описание которых приводится ниже.

cmp cmp(*f1*, *f2*, *shallow*=True)
: Сравнивает файлы, заданные строками путей *f1* и *f2*. Если файлы считаются одинаковыми, функция `cmp` возвращает значение `True`, в противном случае возвращается значение `False`. Если аргумент *shallow* имеет значение `True`, то файлы считаются одинаковыми, когда равны их кортежи `stat`. Если значение аргумента *shallow* равно `False`, то функция `cmp` читает и сжимает содержимое файлов, кортежи `stat` которых равны.

cmpfiles cmpfiles(*dir1*, *dir2*, *common*, *shallow*=True)

Сравнивает файлы, перечисленные в последовательности *common*. Каждый элемент *common* является строкой с именем файла, содержащегося в обоих каталогах, *dir1* и *dir2*. Функция cmpfiles возвращает кортеж, элементами которого являются три списка строк: (*equal*, *diff*, *errs*). Список *equal* содержит имена файлов, одинаковых в обоих каталогах, *diff* — это список имен отличающихся файлов, а *errs* — это список имен файлов, сравнение которых не может быть выполнено (поскольку они не существуют одновременно в обоих каталогах или отсутствуют разрешения на их чтение). Аргумент *shallow* имеет тот же смысл, что и в функции *cmp*.

dircmp class dircmp(*dir1*, *dir2*, *ignore*=('RCS', 'CVS', 'tags'), *hide*=('.', '..'))

Создает новый объект сравнения каталогов, выполняющий сравнение каталогов *dir1* и *dir2*, игнорируя имена, перечисленные в аргументе *ignore*, и скрывая имена, перечисленные в аргументе *hide*. (В версии v3 значение по умолчанию для *ignore* содержит большее количество файлов и предоставляется атрибутом DEFAULT_IGNORE модуля filecmp; на момент выхода книги этот список включал следующие имена: ['RCS', 'CVS', 'tags', '.git', '.hg','.bzr', '_darcs', '__pycache__'].)

Экземпляр *d* класса dircmp предоставляет следующие три метода.

d.report()

Выводит в стандартный поток sys.stdout результаты сравнения *dir1* и *dir2*.

d.report_partial_closure()

Выводит в стандартный поток sys.stdout результаты сравнения *dir1* и *dir2* и их общие ближайшие вложенные подкаталоги.

d.report_full_closure()

Рекурсивно выводит в стандартный поток sys.stdout результаты сравнения *dir1* и *dir2* и все их общие вложенные каталоги.

Кроме того, экземпляр *d* предоставляет несколько атрибутов, которые рассматриваются в следующем разделе.

Атрибуты экземпляра dircmp

Экземпляр *d* класса dircmp предоставляет несколько атрибутов, вычисляемых "оперативно" ("just in time", т.е. тогда и только тогда, когда в этом возникает необходимость, используя для этого специальный метод __getattr__), поэтому использование экземпляра dircmp не влечет за собой никаких излишних накладных расходов. Ниже приведено краткое описание этих атрибутов.

d.common

Список файлов и подкаталогов, общих для каталогов *dir1* и *dir2*.

d.common_dirs

Список подкаталогов, общих для каталогов *dir1* и *dir2*.

`d.common_files`
Список файлов, общих для каталогов *dir1* и *dir2*.

`d.common_funny`
Список имен, общих для каталогов *dir1* и *dir2*, для которых `os.stat` выводит отчет об ошибках или возвращает различные типы для версий, хранящихся в обоих каталогах.

`d.diff_files`
Список файлов, общих для каталогов *dir1* и *dir2*, но имеющих разное содержимое.

`d.funny_files`
Список файлов, общих для каталогов *dir1* и *dir2*, сравнение которых не может быть выполнено.

`d.left_list`
Список файлов и подкаталогов, содержащихся в каталоге *dir1*.

`d.left_only`
Список файлов и подкаталогов, содержащихся в каталоге *dir1*, но не содержащихся в каталоге *dir2*.

`d.right_list`
Список файлов и подкаталогов, содержащихся в каталоге *dir2*.

`d.right_only`
Список файлов и подкаталогов, содержащихся в каталоге *dir2*, но не содержащихся в каталоге *dir1*.

`d.same_files`
Список файлов и подкаталогов, общих для каталогов *dir1* и *dir2* и имеющих одинаковое содержимое.

`d.subdirs`
Словарь, ключами которого являются строки, содержащиеся в `common_dirs`. Соответствующими значениями являются экземпляры `dircmp` для каждого подкаталога.

Модуль `fnmatch`

Модуль `fnmatch` (сокр. от *filename match* — сопоставление имен) позволяет сопоставлять строки, содержащие имена файлов, с шаблонами, похожими на те, которые используются оболочками Unix.

`*`
Соответствует любой последовательности символов.

`?`
Соответствует любому одиночному символу.

`[chars]`

Соответствует любому одиночному символу, включенному в список `chars`.

`[!chars]`

Соответствует любому одиночному символу, не включенному в список `chars`.

Модуль `fnmatch` *не* следует другим соглашениям, действующим в оболочках Unix в отношении сопоставления с шаблонами, в частности, соглашениям, касающимся использования символов косой черты (`/`) или ведущей точки (`.`). Он также не допускает экранирования специальных символов: вместо этого для литерального сопоставления со специальным символом его следует заключить в квадратные скобки. Например, для сопоставления с именем файла, представленным одиночной закрывающей квадратной скобкой, следует использовать шаблон `'[]]'`.

Модуль `fnmatch` предоставляет следующие функции.

`filter` `filter(names, pattern)`

Возвращает список элементов имен (последовательность строк), соответствующих шаблону.

`fnmatch` `fnmatch(filename, pattern)`

Возвращает значение `True`, если строка *`filename`* соответствует шаблону; в противном случае возвращается значение `False`. При сопоставлении с шаблоном регистр букв может учитываться, как, например, на любой Unix-подобной платформе, или не учитываться (например, на платформах Windows). Не забывайте об этом, если имеете дело с файловой системой, поведение которой в отношении регистра букв отличается от поведения операционной системы (например, macOS — Unix-подобная система, однако ее типичные файловые системы нечувствительны к регистру).

`fnmatchcase` `fnmatchcase(filename, pattern)`

Возвращает значение `True`, если строка *`filename`* соответствует шаблону; в противном случае возвращается значение `False`. При сопоставлении с шаблоном регистр букв всегда учитывается на любой платформе.

`translate` `translate(pattern)`

Возвращает шаблон регулярного выражения (см. раздел "Синтаксис строковых шаблонов" в главе 9), эквивалентный шаблону *`pattern`* функции `fnmatch`. Это, например, может быть весьма полезным для выполнения сопоставлений, всегда нечувствительных к регистру на любой платформе, — функциональность, которую функция `fnmatch` не предоставляет.

```
import fnmatch, re

def fnmatchnocase(filename, pattern):
    re_pat = fnmatch.translate(pattern)
    return re.match(re_pat, filename, re.IGNORECASE)
```

Модуль glob

Модуль glob позволяет получить список путей всех файлов (расположенных в произвольном порядке), соответствующих *шаблону* пути, с использованием тех же правил сопоставления, которые использует функция fnmatch. кроме того, он не обрабатывает ведущую точку (.) специальным образом, как это делают оболочки Unix.

glob glob(*pathname*)

Возвращает список путей к файлам, соответствующих шаблону *pathname*. Только в версии v3 можно дополнительно передать необязательный именованный аргумент *recursive*=True, чтобы элемент пути ** рекурсивно сопоставлялся с нулем или бо́льшим количеством уровней подкаталогов.

iglob iglob(*pathname*)

Аналог glob, но возвращает итератор, предоставляющий по одному пути за один раз.

Модуль shutil

Модуль shutil (сокр. от "shell utilities" — утилиты оболочки) предоставляет функции, позволяющие копировать и перемещать файлы, а также удалять целые деревья каталогов. На некоторых Unix-платформах в версии v3 большинство функций этого модуля поддерживает необязательный, задаваемый как только именованный, аргумент *follow_symlinks*, который по умолчанию имеет значение True. Если в качестве пути указана символическая ссылка и аргумент *follow_symlinks* имеет истинное значение, то функция использует эту ссылку для обращения к фактическому файлу или каталогу (в версии v2 функции всегда ведут себя подобным образом). В случае ложного значения указанного аргумента функция воздействует на саму символическую ссылку.

copy copy(*src*, *dst*)

Копирует содержимое файла *src* и записывает его в файл *dst*, заменяя содержимое существующего файла или создавая его, если он не существовал. Если *dst* является каталогом, то копия записывается в него с тем же базовым именем, что и *src*. Функция copy копирует также биты разрешений, но не время последнего доступа и время последнего изменения файла. В версии v3 данная функция возвращает путь к файлу, в который копируется содержимое (в версии v2 функция copy возвращает значение None, что менее полезно).

copy2 copy2(*src*, *dst*)

Аналогична copy, но копирует также время последнего доступа и время последнего изменения файла.

copyfile copyfile(*src*, *dst*)

Копирует только содержимое файла *src* (биты разрешений, время последнего доступа и время последнего изменения не копируются), заменяя содержимое существующего файла или создавая его, если он не существовал.

copyfileobj `copyfileobj(fsrc, fdst, bufsize=16384)`

Копирует все байты из объекта "файла" *fsrc*, который должен быть открыт для чтения, в объект "файла" *fdst*, который должен быть открыт для записи. Копирует не более чем *bufsize* байт за один раз, если *bufsize* больше 0. Файловые объекты были рассмотрены в разделе "Модуль io".

copymode `copymode(src, dst)`

Копирует биты разрешений файла или каталога *src* в файл или каталог *dst*. Как *src*, так и *dst* должны существовать. Не изменяет ни содержимого *dst*, ни его статуса как файла или каталога.

copystat `copystat(src, dst)`

Копирует биты разрешений, а также время последнего доступа и время последнего изменения файла или каталога *src* в файл или каталог *dst*. Объекты *src* и *dst* должны существовать. Не изменяет ни содержимого *dst*, ни его статуса как файла или каталога.

copytree `copytree(src, dst, symlinks=False, ignore=None)`

Рекурсивно копирует дерево каталогов с корнем в каталоге *src* в каталог *dst*. Каталог назначения *dst* не должен существовать: функция `copytree` создает его (а также любой недостающий родительский каталог). Функция `copytree` копирует файлы, используя функцию `copy2` (в версии v3 вместо нее можно использовать другую функцию, заданную с помощью дополнительного именованного аргумента *copy_function*).

Если аргумент *symlinks* имеет истинное значение, то все обнаруженные в исходном дереве символические ссылки будут представлены символическими ссылками в новом дереве. Если аргумент *symlinks* имеет ложное значение, то в новое дерево будут скопированы файлы, на которые указывают символические ссылки, содержащиеся в исходном дереве. На платформах, для которых понятия символической ссылки не существует, функция `copytree` игнорирует аргумент *symlinks*.

Если аргумент *ignore* не равен `None`, в нем должен передаваться вызываемый объект, принимающий два аргумента (путь к каталогу и список непосредственно вложенных в него каталогов) и возвращающий список таких дочерних каталогов, которые должны игнорироваться в процессе копирования. Если аргумент *ignore* имеется, то обычно он является результатом вызова функции `shutil.ignore_patterns`. Например, код

```
import shutil
ignore = shutil.ignore_patterns('.*', '*.bak')
shutil.copytree('src', 'dst', ignore=ignore)
```

копирует дерево с корнем в каталоге *src* в новое дерево с корнем в каталоге *dst*, игнорируя файлы и подкаталоги, имена которых начинаются с символа точки (`.`), а также файлы и подкаталоги, имена которых заканчиваются расширением `'.bak'`.

ignore_patterns `ignore_patterns(*patterns)`

Возвращает вызываемый объект, выбирающий список имен файлов и подкаталогов, которые соответствуют шаблонам *patterns*, аналогичным тем, которые используются в модуле `fnmatch` (см. раздел "Модуль `fnmatch`"). Этот результат может передаваться в качестве аргумента *ignore* функции `copytree`.

move `move(src, dst)`

Перемещает файл или каталог *src* в *dst*. Сначала делается попытка выполнить функцию `os.rename`. Если выполнить эту операцию не удается (по той причине, что *src* и *dst* принадлежат к разным файловым системам или файл *dst* уже существует), то выполняется копирование *src* в *dst* (с помощью функции `copy2` для файлов и функции `copytree` для каталогов; в версии v3 вместо функции `copy2` можно использовать другую функцию копирования, задав ее с помощью аргумента *copy_function*), после чего *src* удаляется (с помощью функции `os.unlink` для файлов и функции `rmtree` для каталогов).

rmtree `rmtree(path, ignore_errors=False, onerror=None)`

Удаляет дерево каталогов с корнем в *path*. Если аргумент *ignore_errors* равен `True`, функция `rmtree` игнорирует ошибки. Если аргумент `ignore_errors` равен `False`, а аргумент `onerror` — `None`, то ошибки возбуждают исключения. Если аргумент *onerror* не равен `None`, то он должен быть вызываемым объектом, принимающим три аргумента: *func*, *path* и *ex*. Аргумент *func* — это функция, возбудившая исключение (`os.remove` или `os.rmdir`), аргумент *path* — это путь, переданный функции *func*, а аргумент *ex* — это кортеж элементов информации, возвращенный вызовом `sys.exc_info()`. Если функция *onerror* возбуждает исключение, то функция `rmtree` завершает работу, а исключение распространяется.

Файл *shutil.py* исходного кода, входящий в состав стандартной библиотеки Python, не только предлагает функции, которые вы сможете непосредственно использовать, но и сам служит отличным примером использования многих функций модуля `os`.

Операции над файловыми дескрипторами

Многие из предоставляемых модулем `os` функций предназначены для работы с *дескрипторами файлов* — целыми числами, используемыми операционной системой для обработки непрозрачных указателей, ссылающихся на открытые файлы. Обычно для задач ввода-вывода лучше использовать объекты "файлов" Python (см. раздел "Модуль `io`"), но иногда работа на уровне файловых дескрипторов позволяет ускорить выполнение операций или (за счет потери переносимости) сделать то, чего нельзя сделать непосредственно с помощью функции `io.open`. Объекты "файлов" и файловые дескрипторы не являются взаимозаменяемыми.

Для получения файлового дескриптора *n* объекта *f* "файла" Python следует выполнить вызов `n=f.fileno()`. Чтобы поместить дескриптор *fd* открытого файла в объект *f* "файла" Python, следует выполнить вызов `f=os.fdopen(fd)` или передать *fd* в качестве первого аргумента функции `io.open`. На Unix-подобных и Windows-платформах некоторые дескрипторы файлов получают предопределенные значения при запуске процесса: `0` — дескриптор стандартного потока ввода, `1` — дескриптор стандартного потока вывода, `2` — дескриптор стандартного потока ошибок.

Количество функций, предоставляемых модулем `os` для работы с дескрипторами файлов, довольно велико. Наиболее часто используемые из них кратко описаны в табл. 10.5.

Таблица 10.5. Функции модуля os

Файл	Описание
close	`close(fd)` Закрывает дескриптор файла *fd*
closerange	`closerange(fd_low, fd_high)` Закрывает все дескрипторы от *fd_low* до *fd_high*, включая граничные значения
dup	`dup(fd)` Возвращает дескриптор файла, являющийся копией дескриптора *fd*
dup2	`dup2(fd, fd2)` Копирует дескриптор файла *fd* в дескриптор файла *fd2*. Если дескриптор файла *fd2* уже открыт, *dup2* сначала закрывает *fd2*
fdopen	`fdopen(fd, *a, **k)` Подобна функции `io.open`, за исключением того, что *fd* должен быть целым числом, являющимся дескриптором открытого файла
fstat	`fstat(fd)` Возвращает экземпляр *x* класса `stat_result`, содержащий информацию о файле, открытом с помощью файлового дескриптора *fd*. Содержимое *x* описано в табл. 10.4
lseek	`lseek(fd, pos, how)` Устанавливает указатель текущей позиции для дескриптора файла *fd* в соответствии с целочисленным значением со знаком, определяемым смещением *pos* (в байтах), и возвращает результирующее смещение от начала файла. Аргумент *how* указывает на начало отсчета (точка `0`). Значению *how*, равному `os.SEEK_SET`, соответствует начало файла; значению `os.SEEK_CUR` — текущая позиция; значению `os.SEEK_END` — конец файла. В частности, вызов `lseek(fd, 0, os.SEEK_CUR)` возвращает смещение текущей позиции в байтах от начала файла, не влияя на текущую позицию.

Файл	Описание
	Обычные дисковые файлы поддерживают поиск; вызов функции `lseek` для файла, который не поддерживает поиск (например, файла, открытого для вывода на терминал), приводит к возникновению исключения

open	`open(`*`file`*`, `*`flags`*`, `*`mode`*`=0o777)`
	Возвращает дескриптор файла, открывая или создавая файл, заданный строковым аргументом *`file`*. В случае создания файла аргумент *`mode`* используется в качестве битов разрешений для доступа к файлу. Аргумент *`flags`* является целым числом, обычно получаемым посредством применения побитовой операции ИЛИ (выполняемой с помощью оператора `|`) к одному или нескольким атрибутам модуля `os`, перечисленным ниже.
	`O_RDONLY O_WRONLY O_RDWR`
	Открыть файл для чтения, для записи или для чтения и записи соответственно (эти флаги являются взаимоисключающими, поэтому только один из них может включаться в аргумент *`flags`*)
	`O_NDELAY O_NONBLOCK`
	Открыть файл *`file`* в неблокирующем (без задержки) режиме, если платформа поддерживает такую возможность
	`O_APPEND`
	Открыть файл для присоединения нового содержимого к предыдущему содержимому файла
	`O_DSYNC O_RSYNC O_SYNC O_NOCTTY`
	Установить соответствующий режим синхронизации, если файл поддерживает эту возможность
	`O_CREAT`
	Создать файл, если он не существует
	`O_EXCL`
	Возбудить исключение, если файл существует
	`O_TRUNC`
	Отбросить предыдущее содержимое файла (несовместим с флагом `O_RDONLY`)
	`O_BINARY`
	Открыть файл в двоичном, а не текстовом режиме на платформах, отличных от Unix-подобных (на Unix-подобных платформах ни на что не влияет и ничему не может повредить)

pipe	`pipe()` Создает канал и возвращает пару дескрипторов файлов `(`*`r`*`, `*`w`*`)`, предназначенных для чтения и записи соответственно

Файл	Описание
read	`read(fd, n)` Читает до `n` байтов из файла с дескриптором `fd` и возвращает их в виде байтовой строки. Читает и возвращает `m<n` байтов, если в данный момент для чтения из файла доступны только `m` байт. В частности, возвращает пустую строку, если в данный момент нет байтов для чтения, что обычно бывает по достижении конца файла
write	`write(fd, s)` Записывает все байты из файловой строки `s` в файл с дескриптором `fd` и возвращает количество записанных байтов (т.е. `len(s)`)

Ввод и вывод текста

Предоставляемые программам на Python каналы текстового ввода и вывода являются объектами "файлов" (см. раздел "Атрибуты и методы файловых объектов"), методы которых, таким образом, можно использовать для работы с этими каналами.

Стандартные потоки вывода и ошибок

Модуль `sys` (см. раздел "Модуль `sys`" в главе 7) предоставляет атрибуты `stdout` и `stderr`, значениями которых являются записываемые объекты "файлов". Если вы не используете перенаправление каналов или конвейеры оболочки, то эти потоки подключаются к "терминалу", с которого был запущен сценарий. В наши дни реальные терминалы являются редкостью: обычно то, что подразумевают под "терминалом", — это экранное окно, поддерживающее текстовый ввод-вывод (например, консоль командной строки Windows или окно `xterm` в Unix).

Различие между потоками `sys.stdout` и `sys.stderr` — чисто условное и основывается на формальном соглашении. Объект `sys.stdout`, известный как *стандартный поток вывода*, используется для вывода результатов работы программы. Объект `sys.stderr`, известный как *стандартный поток ошибок*, используется для вывода сообщений об ошибках. Отделение результатов от сообщений об ошибках обеспечивает более эффективное использование средств перенаправления ввода-вывода оболочки. Python следует упомянутому соглашению, используя объект `sys.stderr` для вывода ошибок и предупреждений.

Функция `print`

Программам, направляющим результаты в стандартный вывод, часто требуется осуществлять запись в объект `sys.stdout`. Функция `print` (см. табл. 7.2) может выступать в качестве удобной альтернативы методу `sys.stdout.write`. (В версии v2, чтобы сделать `print` *функцией*, следует поместить в начале модуля инструкцию `from`

`__future__ import print_function`, иначе вам придется использовать `print` как инструкцию.)

Функцию `print` удобно использовать для вывода отладочной информации в процессе разработки кода. Для вывода полезной информации вам потребуются более широкие возможности управления форматированием, чем те, которые предлагает функция `print`. Возможно, вам понадобится управление пробельными промежутками, шириной полей, количеством десятичных цифр в числах с плавающей точкой и т.п. Если это так, подготовьте вывод в виде строки с помощью метода форматирования строк `format` (см. раздел "Форматирование строк" в главе 8) и выведите выходную строку с помощью метода `write` соответствующего объекта "файла". (Форматированные строки можно передавать методу `print`, но он может добавлять пробелы и символы новой строки. Метод `write` не добавляет ничего лишнего, поэтому вам будет легче управлять точным видом выводимой информации.)

Чтобы направить вывод, полученный с помощью вызова функции `print`, в определенный файл и при этом избавиться от необходимости повторного использования определений `file=`*`destination`* при каждом вызове `print`, можно временно изменить значение `sys.stdout`. В качестве примера ниже приведена универсальная функция перенаправления вывода, пригодная для внесения подобных временных изменений. Если предполагается выполнение асинхронных операций, то обязательно добавляйте блокировку, чтобы избежать конфликтов при попытках доступа к файлу.

```
def redirect(func, *args, **kwds):
  """redirect(func, ...) -> (строковый результат, возвращаемое
                             func значение)
  Аргумент func должен быть вызываемым объектом, который может
  направлять результаты в стандартный вывод. Функция redirect
  перехватывает эти результаты в виде строки и возвращает пару,
  первым элементом которой является выходная строка, а вторым -
  значение, возвращаемое func.
"""
  import sys, io
  save_out = sys.stdout
  sys.stdout = io.StringIO()
  try:
      retval = func(*args, **kwds)
      return sys.stdout.getvalue(), retval
  finally:
    sys.stdout.close()
    sys.stdout = save_out
```

Избегайте подобных манипуляций, если требуется всего лишь вывести несколько текстовых значений в объект *f* файла, не являющийся текущим значением `sys.stdout`. Для таких простых целей во многих случаях гораздо лучше вызвать метод `f.write`, а зачастую еще лучше воспользоваться удобной альтернативой в виде вызова `print(file=`*f*`,...)`.

Стандартный ввод

Модуль sys предоставляет атрибут stdin, значением которого является читаемый файловый объект. Если вам нужно получить от пользователя текстовую строку, то можно вызвать встроенную функцию input (см. табл. 7.2; в версии v2 эта функция называется raw_input), которая принимает необязательный строковый аргумент, задающий строку приглашения к вводу.

Если ожидаемый ввод не является строкой (например, если вам нужно получить число), то используйте функцию input для получения строки от пользователя, а затем преобразуйте строку в число с помощью таких встроенных функций, как int, float или ast.literal_eval (описана ниже).

Теоретически для этого можно использовать также функцию eval (вызову которой обычно предшествует вызов функции compile, что облегчает диагностику ошибок), чтобы предоставить пользователю возможность ввести любое выражение, при условии что вы полностью доверяете пользователю. Злонамеренный пользователь может воспользоваться функцией eval для создания бреши в системе безопасности и причинения ущерба (добросовестный пользователь, не отдающий отчета в своих действиях, может непреднамеренно причинить не меньший вред). Эффективной защиты от этого не существует. Самая надежная защита — полностью избегать использования функции eval (а также exec!) при получении ввода от источников, которым вы не доверяете.

Эффективным альтернативным способом, который мы рекомендуем, является использование функции literal_eval из модуля ast стандартной библиотеки (см. онлайн-документацию). Функция ast.literal_eval(*astring*) возвращает корректное значение Python для заданного литерала *astring*, когда это возможно, либо возбуждает исключение SyntaxError или ValueError. Ее вызов никогда не сопровождается побочными эффектами. Однако в этом случае для гарантии полной безопасности в *astring* нельзя использовать никакие операторы и никакие идентификаторы, кроме ключевых слов. Приведем соответствующий пример.

```
import ast
print(ast.literal_eval('23')) # выводит 23
print(ast.literal_eval('[2,3]')) # выводит [2, 3]
print(ast.literal_eval('2+3')) # возбуждает исключение ValueError
print(ast.literal_eval('2+')) # возбуждает исключение SyntaxError
```

Модуль getpass

В некоторых очень редких случаях вы захотите предоставить пользователю возможность вводить текст таким образом, чтобы никто из посторонних не мог незаметно подсмотреть из-за спины, что именно он вводит. Например, такая возможность не будет лишней, если речь идет о вводе пароля. Предназначенные для этого функции, предоставляемые модулем getpass, кратко описаны ниже.

getpass `getpass(prompt='Password: ')`

Подобна функции `input`, за исключением того, что вводимый пользователем текст не отображается на экране. Подсказка, используемая по умолчанию функцией `getpass`, отличается от подсказки, используемой функцией `input`.

getuser `getuser()`

Возвращает имя текущего пользователя. Функция `getuser` пытается получить имя пользователя в виде значения одной из переменных среды `LOGNAME`, `USER`, `LNAME` и `USERNAME` в указанной последовательности. Если ни одной из этих переменных нет в списке `os.environ`, то функция `getuser` запрашивает имя пользователя у операционной системы.

Расширенные возможности текстового ввода-вывода

Рассмотренные до сих пор инструменты представляют минимальное подмножество функциональности текстового ввода-вывода на всех платформах. На большинстве платформ предлагаются расширенные возможности текстового ввода-вывода, такие как реакция на нажатия одиночных клавиш (а не на ввод целых строк) и отображение текста в любом месте терминала.

Расширения Python и модули ядра Python обеспечивают доступ к функциональности, специфической для конкретных платформ. К сожалению, способы предоставления этой функциональности разными платформами различаются. Для разработки кросс-платформенных программ на Python с расширенной функциональностью текстового ввода-вывода вам может понадобиться унифицированный способ обертывания различных модулей и условный импорт модулей, специфических для платформ (обычно с использованием идиомы `try/except`, рассмотренной в разделе "Инструкция `try/except`" в главе 5).

Модуль `readline`

Модуль `readline` обертывает библиотеку GNU Readline. Библиотека Readline позволяет пользователю редактировать строки в процессе интерактивного ввода и вызывать ранее введенные строки для их редактирования и последующего ввода. Библиотека Readline предустанавливается на многих Unix-подобных платформах и доступна для скачивания из Интернета (`http://tiswww.case.edu/php/chet/readline/rltop.html`). На платформах Windows для этих же целей можно установить и использовать модуль `pyreadline` сторонних производителей (`https://pypi.python.org/pypi/pyreadline/2.0`).

Если доступен модуль `readline`, Python использует его для любых видов построчного ввода, например таких, как ввод с помощью функции `input`. Интерактивный интерпретатор Python всегда пытается загрузить модуль `readline`, чтобы позволить редактировать команды и обращаться к истории вызовов в интерактивных сеансах. Некоторые из функций модуля `readline` управляют расширенной

функциональностью, в том числе *историей команд*, обеспечивая вызов строк, введенных в предыдущих сеансах, и контекстно-зависимым *автодополнением* вводимых слов. (Более подробную информацию о настройке конфигурации библиотеки GNU Readline можно найти в онлайн-документации.) Доступ к функциональности, предоставляемой данным модулем, обеспечивают описанные ниже функции.

add_history `add_history(s)`

Добавляет строку *s* в конец буфера истории команд (введенных строк).

clear_history `clear_history(s)`

Очищает буфер истории команд.

get_completer `get_completer()`

Возвращает текущую функцию автодополнения (в соответствии с последним значением, установленным вызовом функции `set_completer`) или значение `None`, если такая функция не установлена.

get_history_length `get_history_length()`

Возвращает количество строк, которые сохраняются в истории команд. Если результат меньше `0`, то в истории команд хранятся все строки.

parse_and_bind `parse_and_bind(readline_cmd)`

Предоставляет Readline команду настройки конфигурации. Чтобы дать пользователю возможность запрашивать автодополнение слов нажатием клавиши <Tab>, выполните вызов `parse_and_bind('tab: complete')`. О других полезных значениях строки *readline_cmd* можно прочитать в онлайн-документации библиотеки Readline.

Неплохая функция завершения ввода содержится в модуле `rlcompleter`. Введите в интерактивном интерпретаторе (или в файле запуска, выполняемом в самом начале интерактивных сеансов, о котором рассказано в разделе "Переменные среды" в главе 2) следующий код:

```
import readline, rlcompleter
readline.parse_and_bind('tab: complete')
```

До окончания текущего интерактивного сеанса нажатие клавиши <Tab> в процессе редактирования строки ввода будет активизировать средство автодополнения глобальных имен и атрибутов объектов.

read_history_file `read_history_file(filename='~/.history')`

Загружает историю ввода команд из текстового файла *filename*.

read_init_file `read_init_file(filename=None)`

Выполняет файл инициализации Readline: каждая строка этого файла является конфигурационной командой. Если аргумент `filename` равен `None`, загружается тот же файл, который загружался в последний раз.

set_completer `set_completer(f=None)`

Устанавливает функцию автодополнения. Если аргумент `f` равен `None`, то Readline отключает автодополнение. В противном случае, если пользователь вводит часть `start` слова, а затем нажимает клавишу <Tab>, Readline вызывает `f(start, i)` с начальным значением `i`, равным `0`. Функция `f` возвращает `i`-е возможное слово, начинающееся с части `start`, или значение `None`, если такие слова отсутствуют. Readline вызывает функцию `f` в цикле со значениями `i`, равными `0`,`1`, `2` и т.д., пока `f` не вернет значение `None`.

set_history_length `set_history_length(x)`

Задает количество строк, сохраняемых в файле истории команд. Если `x` меньше `0`, то в истории команд сохраняются все строки.

write_history_file `write_history_file(filename='~/.history')`

Сохраняет строки команд в текстовом файле, имя или путь к которому задается аргументом `filename`.

Консольный ввод-вывод

В наши дни выражение "терминал" обычно относится к текстовому окну на графическом экране. Теоретически вы можете использовать для работы в текстовом режиме истинный терминал или консоль (основной экран) персонального компьютера. Все современные "терминалы" подобного рода предлагают развитую функциональность текстового ввода-вывода, доступ к которой осуществляется специфическими для конкретной платформы способами. Библиотека `curses` работает на Unix-подобных платформах. В качестве кросс-платформенного решения (Windows, Unix, Mac) можно использовать пакет `UniCurses` сторонних производителей. В противоположность этому модуль `msvcrt` доступен только в Windows.

Библиотека curses

Использованный в Unix классический подход к реализации расширенной функциональности терминальных операций ввода-вывода в силу неясных исторических причин получил название *curses* (букв. "проклятия")[1]. Библиотека `curses` в Python допускает использование достаточно простых средств, но наряду с этим позволяет брать на себя детальное управление вводом-выводом текста, если возникает такая необходимость. В этом разделе мы рассмотрим лишь небольшое подмножество функциональности

[1] Слово "curses" довольно точно отражает типичную реакцию программиста, впервые столкнувшегося с богатой функциональностью этого усложненного подхода.

curses, только ту ее малую часть, которой вам хватит для написания программ, предлагающих расширенные возможности ввода-вывода. (Более подробно эта тема изложена в практическом руководстве Эрика Реймонда *Curses Programming with Python*; https://docs.python.org/2/howto/curses.html#curses-howto). Всякий раз, когда в тексте встречается слово "экран", оно означает экран терминала (которым в наши дни обычно является текстовое окно программы, эмулирующей терминал).

Простейший и наиболее эффективный способ использования возможностей библиотеки curses обеспечивает функция curses.wrapper, описанная ниже.

wrapper wrapper(*func*, **args*)

В качестве первого аргумента, *func*, всегда должна передаваться функция. Функция wrapper инициализирует библиотеку curses, создавая объект верхнего уровня (окно) *w* типа curses.window, после чего вызывает функцию *func*, передавая ей в качестве первого аргумента объект *w*, а вслед за ним другие свои аргументы, *args*. Если *func* возвращает значение, функция wrapper переводит терминал в нормальный режим работы и завершает выполнение возвратом значения функции *func*. Итак, функция wrapper выполняет инициализацию библиотеки curses, осуществляет вызов *func*(*w*, **args*), выполняет завершающие операции по освобождению ресурсов, которые библиотеке curses уже не нужны (переводит терминал в нормальный режим работы), и возвращает результат работы функции *func* (или повторно возбуждает исключение, которое могла распространить функция *func*). Ключевым фактором является то, что функция wrapper возвращает терминал в нормальный режим работы, независимо от того, успешно ли завершила свое выполнение функция *func* или распространила исключение.

Аргумент *func* должен быть функцией, которая выполняет в вашей программе все задачи, требующие использования функциональности библиотеки curses. Другими словами, функция *func* содержит всю функциональность вашей программы (или вызывает, прямо или косвенно, функции, содержащие такую функциональность), связанную с использованием библиотеки curses, кроме, возможно, некоторых неинтерактивных операций, связанных с инициализацией системы и/или освобождением ненужных ресурсов по завершении работы программы.

Библиотека curses моделирует цвета текста и фона посредством атрибутов символов. Доступные для терминала цвета пронумерованы от 0 до curses.COLORS. Функция color_content принимает номер цвета *n* в качестве аргумента и возвращает кортеж (*r*, *g*, *b*) целых чисел в диапазоне от 0 до 1000, определяющих интенсивность каждой составляющей цвета в *n*. Функция color_pair принимает номер цвета *n* в качестве аргумента и возвращает атрибут code, который можно передавать различным методам объекта curses.Window для отображения текста в этом цвете.

Библиотека curses позволяет создавать несколько экземпляров типа curses.Window, каждому из которых соответствует прямоугольное окно на экране. Кроме того, возможно создание экзотических вариантов, таких как экземпляры класса Panel, полиморфного классу Window, но не привязанного к фиксированной прямоугольной области на экране. Подобная сложная функциональность не потребуется в простых программах: вам будет вполне достаточно использовать объект stdscr

класса Window, предоставляемый функцией curses.wrapper. Для реального отображения на экране изменений, внесенных в любой экземпляр w класса Window, включая экземпляр stdscr, вызовите метод w.refresh(). Библиотека curses может хранить изменения в буфере до тех пор, пока не будет вызван метод refresh.

Экземпляр w класса Window предоставляет, среди прочего, следующие часто используемые методы.

addstr w.addstr([y, x,]s[, attr])

Помещает цепочку символов строки s с атрибутом attr в объект w, начиная с позиции, заданной координатами (x, y), и заменяя предыдущее содержимое. Все функции и методы библиотеки curses принимают аргументы координат в обратной последовательности, т.е. y (номер строки на экране) предшествует x (номеру столбца). Если аргументы y и x опущены, метод addstr использует текущие координаты курсора объекта w. Если атрибут attr опущен, метод addstr использует для атрибута его текущее значение по умолчанию, установленное для объекта w. В любом случае после добавления строки метод addstr устанавливает текущие координаты курсора объекта w в конец добавленной строки.

clrtobot, clrtoeol w.clrtobot(), w.clrtoeol()

Метод clrtoeol очищает строку от текущей позиции курсора объекта w до конца строки. Метод clrtobot отличается тем, что дополнительно очищает все строки экрана, расположенные ниже текущей.

delch w.delch([y, x])

Удаляет из объекта w один символ в позиции с координатами (x, y). Если аргументы y и x опущены, метод delch использует текущие координаты позиции курсора объекта w. В любом случае текущие координаты позиции курсора объекта w не изменяются. Все последующие символы в строке y, если таковые имеются, сдвигаются на одну позицию влево.

deleteln w.deleteln()

Удаляет из объекта w всю строку, соответствующую координатам текущей позиции курсора, и прокручивает все строки экрана, расположенные ниже текущей, вверх на одну строку.

erase w.erase()

Записывает пробелы во все позиции экрана терминала.

getch w.getch()

Возвращает целое число c, которое соответствует клавише, нажатой пользователем. Значения от 0 до 255 представляют обычные символы, тогда как значения свыше 255 — специальные клавиши. В библиотеке curses специальным клавишам назначены имена, поэтому c можно тестировать на равенство таким константам, как curses.KEY_HOME (специальная клавиша <Home>), curses.KEY_LEFT (специальная клавиша "¬") и др. (Пространный список имен специальных клавиш приведен в онлайн-документации Python;

https://docs.python.org/3.5/library/curses.html?highlight=curses#constants.) Если для окна *w* установлен режим no-delay (без задержки) посредством вызова метода `w.nodelay(True)` и никакая клавиша еще не была нажата, то вызов `w.getch` возбуждает исключение. По умолчанию после вызова `w.getch` программа приостанавливается до тех пор, пока пользователь не нажмет какую-либо клавишу.

getyx `w.getyx()`

Возвращает координаты текущей позиции курсора объекта *w* в виде кортежа `(y, x)`.

insstr `w.insstr([y, x, ]s[, attr])`

Вставляет цепочку символов строки *s* с атрибутом *attr* в объект *w* в позицию с координатами `(x, y)`, смещая остальную часть строки вправо. Символы, смещенные за пределы строки, теряются. Если аргументы *y* и *x* опущены, то метод `insstr` использует текущие координаты курсора объекта *w*. Если атрибут *attr* опущен, метод `insstr` использует для атрибута его текущее значение по умолчанию, установленное для объекта *w*. В любом случае после вставки строки метод `insstr` устанавливает текущие координаты курсора объекта *w* в позицию первого символа вставленной строки.

move `w.move(y, x)`

Перемещает курсор объекта *w* в позицию с заданными координатами `(x, y)`.

nodelay `w.nodelay(flag)`

Устанавливает для объекта *w* режим no-delay (без задержки), если аргумент *flag* имеет истинное значение. Восстанавливает для объекта *w* нормальный режим работы, если аргумент *flag* имеет ложное значение. Режим no-delay влияет на работу метода `w.getch`.

refresh `w.refresh()`

Перерисовывает окно *w* на экране с учетом всех изменений, внесенных программой в объект *w*.

Модуль `curses.textpad` предоставляет класс `Textpad`, обеспечивающий поддержку расширенных возможностей ввода и редактирования текста.

Textpad `class Textpad(window)`

Создает и возвращает экземпляр *t* класса `Textpad`, который обертывает экземпляр *window* окна `curses`. Ниже описан один из наиболее часто используемых методов экземпляра *t*.

`t.edit()`

Предоставляет пользователю возможность интерактивно редактировать содержимое экземпляра окна, обернутого экземпляром *t*. Сеанс редактирования поддерживает простые комбинации клавиш в стиле Emacs: обычные символы заменяют предыдущее содержимое окна, клавиши со стрелками перемещают курсор, а комбинация клавиш <Ctrl+H> удаляет

символ, находящийся слева от курсора. Если пользователь нажимает комбинацию клавиш <Ctrl+G>, сеанс редактирования завершается, и метод `edit` возвращает содержимое окна в виде одной строки с использованием символа перевода строки в качестве разделителя.

Модуль `msvcrt`

Модуль `msvcrt`, доступный только в Windows, предоставляет функции, позволяющие программам на Python использовать некоторые из функций библиотеки времени выполнения Microsoft Visual C++ *msvcrt.dll*. Ниже описаны функции библиотеки `msvcrt`, обеспечивающие посимвольное считывание вводимого пользователем текста, а не всей строки сразу.

`getch`, `getche` `getch()`, `getche()`

Считывает нажатую клавишу и возвращает соответствующий символ. В отсутствие символов для считывания этот вызов блокируется и переходит в состояние ожидания нажатия клавиши. Метод `getche` также отображает эхо-символ на экране (если он печатаемый), в то время как метод `getch` этого не делает. Если пользователь нажимает специальную клавишу (клавиши со стрелками, функциональные клавиши и т.п.), то она считывается как два символа: сначала следует символ `chr(0)` или `chr(224)`, а затем второй, который вместе с первым определяет клавишу, нажатую пользователем. Если хотите увидеть, что именно возвращает функция `getch` для любой клавиши, выполните следующий небольшой скрипт на компьютере Windows.

```
import msvcrt
print("press z to exit, or any other key
      to see the key's code:")
while True:
    c = msvcrt.getch()
    if c == 'z': break
    print('{} ({!r})'.format(ord(c), c))
```

`kbhit` `kbhit()`

Возвращает значение `True`, если имеется символ, доступный для чтения (метод `getch` в этой ситуации выполняет немедленный возврат); в противном случае возвращает значение `False` (метод `getch` в этой ситуации переходит в состояние ожидания).

`ungetch` `ungetch(c)`

Отменяет выполненное считывание символа *c*. Следующий вызов метода `getch` или `getche` вернет *c*. Вызов метода `ungetch` два раза подряд без промежуточных вызовов метода `getch` или `getche` является ошибкой.

Интерактивные сеансы

Модуль `cmd` предлагает простой способ организации интерфейса командной строки. Каждая команда — это строка текста. Первым словом каждой команды является

глагол, определяющий запрашиваемое действие. Остальная часть строки представляет собой пользовательские аргументы, которые передаются методу, реализующему соответствующее действие.

Модуль cmd содержит класс Cmd, который можно использовать в качестве базового, определяя собственные подклассы класса cmd.Cmd. Ваш подкласс должен предоставлять методы, имена которых в соответствии с принятым соглашением начинаются с префикса do_ или help_ и которые могут перекрывать некоторые методы Cmd. Если в вашем подклассе определен метод do_*глагол*, то при вводе пользователем команды ***глагол аргументы*** метод Cmd.onecmd выполняет следующий вызов:

```
self.do_глагол('аргументы')
```

Точно так же, если в вашем подклассе определен метод help_*глагол*, то метод Cmd.onecmd выполнит соответствующий вызов при вводе пользователем команды 'help *глагол*' или '?*глагол*'. В тех случаях, когда пользователь пытается выполнить действие или получить справку о действии, для которого глагол не определен, модуль Cmd отображает соответствующее сообщение об ошибке.

Инициализация экземпляра Cmd

Если ваш подкласс класса cmd.Cmd определяет собственный специальный метод __init__, то он должен вызвать метод __init__ базового класса, сигнатура которого описана ниже.

__init__ Cmd.__init__(*self*, *completekey*='Tab', *stdin*=sys.stdin, *stdout*=sys.stdout)

Инициализирует экземпляр self указанными или используемыми по умолчанию значениями аргументов *completekey* (имя клавиши автодополнения, которая будет использоваться, если доступен модуль readline; чтобы отключить автодополнение, передайте значение None), *stdin* (объект файла, из которого следует получать ввод) и *stdout* (объект файла, в который должен направляться вывод). Если ваш подкласс не определяет метод __init__, то он наследует описанный перед этим метод базового класса cmd.Cmd. В таком случае инстанциализируйте свой подкласс, вызвав этот метод с необязательными параметрами *completekey*, *stdin* и *stdout* в соответствии с приведенным выше описанием.

Методы экземпляров Cmd

Экземпляр *c* подкласса класса Cmd предоставляет следующие методы (многие из них являются методами-перехватчиками, так называемыми "хуками", и могут перекрываться вашим подклассом).

cmdloop *c*.cmdloop(*intro*=None)

Выполняет цикл интерактивного сеанса работы со строчно-ориентированными командами. Метод cmdloop начинается с вызова *c*.preloop(), а затем направляет в вывод строку *intro* (*c*.intro, если аргумент *intro* равен None). Далее метод *c*.cmdloop входит в цикл. На каждой итерации цикла метод cmdloop читает строку *s* с помощью инструкции *s*=input(*c*.prompt). По достижении конца файла стандартного ввода метод cmdloop устанавливает для строки *s* значение 'EOF'. Если значение *s* не равно 'EOF', то метод cmdloop предварительно обрабатывает строку *s* с помощью инструкции *s*=*c*.precmd(*s*), после чего выполняет инструкцию *flag*=*c*.onecmd(*s*). Если метод onecmd возвращает истинное значение, то это интерпретируется как пробный запрос на выход из командного цикла. Независимо от значения *flag*, метод cmdloop выполняет инструкцию *flag*=*c*.postcmd(*flag*, *s*) для проверки того, что цикл должен быть завершен. Если значение *flag* является истинным, то происходит выход из цикла; в противном случае цикл повторяется. После выхода из цикла метод cmdloop вызывает метод *c*.postloop() и завершает работу. Вам будет проще понять только что описанную структуру метода, взглянув на следующий эквивалентный код Python.

```
def cmdloop(self, intro=None):
    self.preloop()
    if intro is None: intro = self.intro
    print(intro)
    finis_flag = False
    while not finis_flag:
        try: s = raw_input(self.prompt)
        except EOFError: s = 'EOF'
        else: s = self.precmd(s)
        finis_flag = self.onecmd(s)
        finis_flag = self.postcmd(finis_flag, s)
    self.postloop()
```

Метод cmdloop хорошо иллюстрирует классический шаблон проектирования *Template Method* (шаблонный метод). Сам по себе такой метод выполняет не очень много полезной работы, однако он структурирует и организует вызовы других методов, известных как *методы-перехватчики*, или *хуки*. Подклассы могут переопределять эти методы, изменяя поведение класса в рамках данного кода. Если вы наследуете класс Cmd, то вам почти никогда не придется переопределять метод cmdloop, поскольку основное, что вас интересует при создании подклассов класса Cmd, — это структура данного метода.

default *c*.default(*s*)

Метод *c*.onecmd вызывает метод *c*.default(*s*), если для первого слова *глагол*, встретившегося в строке *s*, не существует соответствующего метода c.do_*глагол*. Подклассы часто переопределяют метод default. Метод Cmd.default базового класса выводит сообщение об ошибке.

do_help *c*.do_help(*глагол*)

Метод *c*.onecmd вызывает метод *c*.do_help(*глагол*), если строка *s* команды начинается с подстроки 'help *глагол*' или '?*глагол*'. Подклассы редко переопределяют метод do_help. Метод Cmd.do_help вызывает метод help_*глагол*, если подкласс предоставляет его. В противном случае он отображает строку документации метода do_*глагол*, если подкласс предоставляет данный метод с непустой строкой документации. Если ни один из указанных источников справочных сведений не предоставляется подклассом, то метод Cmd.do_help выводит сообщение об отсутствии справки для данного глагола.

emptyline *c*.emptyline()

Метод *c*.onecmd вызывает метод *c*.emptyline(), если строка *s* команды пустая или содержит пробелы. Если подкласс не переопределяет этот метод, то метод Cmd.emptyline базового класса повторно выполняет последнюю непустую строку, сохраненную в атрибуте *c*.lastcmd объекта *c*.

onecmd *c*.onecmd(*s*)

Метод *c*.cmdloop вызывает метод *c*.onecmd(*s*) для каждой строки *s* команды, введенной пользователем. Вы также можете вызвать метод onecmd непосредственно, если получили строку *s* каким-либо иным способом для выполнения в качестве команды. Обычно подклассы не переопределяют метод onecmd. Метод Cmd.onecmd выполняет инструкцию *c*.lastcmd=*s*. Затем метод onecmd вызывает метод do_*глагол*, если строка *s* начинается со слова *глагол* и подкласс предоставляет этот метод. В противном случае он вызывает метод emptyline или default, как описывалось ранее. В любом случае метод Cmd.onecmd возвращает результат работы любого из вызванных им других методов. Этот результат будет интерпретирован методом postcmd в качестве флага запроса на завершение выполнения.

postcmd *c*.postcmd(*flag*, *s*)

Метод *c*.cmdloop вызывает метод *c*.postcmd(*flag*, *s*) для каждой командной строки *s* после того, как метод *c*.onecmd(*s*) вернет значение флага *flag*. Если аргумент *flag* имеет истинное значение, то только что выполненная команда возвращает пробный запрос на выход из командного цикла. Если метод postcmd возвращает истинное значение, осуществляется выход из цикла cmdloop. Если ваш класс не переопределяет этот метод, то вызывается метод Cmd.postcmd базового класса, который всего лишь возвращает значение *flag*.

postloop *c*.postloop()

Метод *c*.cmdloop вызывает метод *c*.postloop(), если происходит выход из цикла cmdloop. Ваш класс может переопределить этот метод. Метод Cmd.postloop базового класса не выполняет никаких действий.

precmd *c*.precmd(*s*)

Метод *c*.cmdloop выполняет инструкцию *s*=*c*.precmd(*s*) для предварительной обработки каждой командной строки *s*. Вся последующая обработка во время текущего прохода цикла базируется на строке, возвращаемой методом precmd. Ваш подкласс может переопределить этот метод. Метод Cmd.precmd базового класса всего лишь возвращает саму строку *s*.

preloop `c.preloop()`

Метод `c.cmdloop` вызывает метод `c.preloop()` перед тем, как начнется выполнение цикла `cmdloop`. Ваш подкласс может переопределить этот метод. Метод `Cmd.preloop` базового класса не выполняет никаких действий.

Атрибуты экземпляров Cmd

Экземпляр `c` подкласса класса `Cmd` предоставляет следующие атрибуты.

`identchars`

Строка, содержащая символы, каждый из которых может быть использован в глаголе. По умолчанию `c.identchars` содержит буквы, цифры и символ подчеркивания (_).

`intro`

Сообщение, первоначально выводимое методом `cmdloop` при его вызове без аргумента.

`lastcmd`

Последняя непустая команда, обработанная методом `onecmd`.

`prompt`

Строка, используемая методом `cmdloop` в качестве строки приглашения. Почти всегда вы будете либо явно связывать атрибут `c.prompt`, либо перекрывать `prompt` в качестве атрибута собственного подкласса. По умолчанию `Cmd.prompt` имеет значение `'(Cmd) '`.

`use_rawinput`

Если этот атрибут имеет значение `False` (значение по умолчанию — `True`), то метод `cmdloop` использует для отображения приглашения и ввода команд методы `sys.stdout.write` и `sys.stdin.readline`, а не функцию `input`.

Другие атрибуты экземпляров `Cmd`, которые не попали в этот список, обеспечивают тонкую настройку многих элементов форматирования справочных сообщений.

Пример использования класса Cmd

В следующем примере продемонстрировано, как использовать класс `cmd.Cmd` с целью создания глаголов `print` (для вывода остальной части строки) и `stop` (для выхода из цикла).

```
import cmd

class X(cmd.Cmd):
    def do_print(self, rest):
        print(rest)
    def help_print(self):
        print('print (any string): outputs (any string)')
    def do_stop(self, rest):
```

```
        return True
    def help_stop(self):
        print('stop: terminates the command loop')

if __name__ == '__main__':
    X().cmdloop()
```

Сеанс работы с использованием этого примера может выглядеть так.

```
C:\>python examples/chapter10/cmdex.py
(Cmd) help
Documented commands (type help <topic>):
= == == == == == == == == == == == == == == == == == =
print           stop
Undocumented commands:
= == == == == == == == == == =
help
(Cmd) help print
print (any string): outputs (any string)
(Cmd) print hi there
hi there
(Cmd) stop
```

Интернационализация

Большинство программ предоставляет пользователю некоторую информацию в виде текста. Этот текст должен быть понятен и приемлем для пользователя. Например, в некоторых странах и регионах дату "7 марта" можно записать в сокращенной форме как "3/7". В других странах записи "3/7" соответствует дата "3 июля", а строке, означающей "7 марта", соответствует сокращенная запись "7/3". В Python подобного рода региональные соглашения обрабатываются с помощью стандартного модуля `locale`.

Точно так же стандартное приветствие на английском языке может начинаться со строки "Welcome", а на русском — со строки "Добро пожаловать". То же самое касается текста, используемого в меню, диалоговых окнах и т.п. В Python возможности работы с различными вариантами перевода подобного рода текстов поддерживаются стандартным модулем `gettext`.

Обе проблемы объединяются в рамках общего термина *интернационализация* (часто используемого в виде аббревиатуры *i18n*, поскольку между буквами *i* и *n* в слове *internationalization* содержится ровно 18 букв) — не совсем точное название, поскольку различия в местных стандартах и культурных традициях существуют не только между странами, но и между отдельными регионами одной страны.

Модуль `locale`

В Python поддержка региональных стандартов имитирует аналогичную поддержку в языке программирования C, но несколько упрощена. Региональные настройки,

с которыми работает программа, называются *локалью*. Региональные настройки пронизывают насквозь всю программу и обычно устанавливаются при ее запуске. Настройки локали воздействуют на все потоки выполнения, а модуль `locale` не является потокобезопасным. В многопоточной программе локаль должна устанавливаться до запуска вторичных потоков.

Модуль `locale` полезен лишь для настройки параметров процесса в целом

Если приложение должно работать одновременно с несколькими локалями в одном процессе, — независимо от того, будет ли это происходить в разных потоках или асинхронно, — модуль `locale` не решит ваших проблем, поскольку он воздействует на процесс в целом. В подобных случаях вас могут выручить такие альтернативы, как пакет PyICU (см. раздел "Дополнительные ресурсы интернационализации").

Если программа не вызывает метод `locale.setlocale`, используется нейтральная локаль, известная как *локаль C*. Такое название локали обусловлено тем, что ее архитектура уходит корнями в язык C; она аналогична, но не идентична, локали U.S. English. Программа также может находить и использовать локаль по умолчанию для данного пользователя. В этом случае с целью нахождения предпочтительной для пользователя локали модуль `locale` взаимодействует с операционной системой (посредством переменных среды или иными способами, зависящими от системы). Наконец, программа может установить конкретную локаль на основании действий пользователя или посредством конфигурационных параметров.

Обычно настройка локали выполняется сразу для всех категорий региональных соглашений. Такая всеобщая настройка обозначается константой `LC_ALL` модуля `locale`. Однако региональные соглашения, обрабатываемые локалью, группируются в категории, и в некоторых случаях программа может смешивать категории, создавая искусственную композитную локаль. Для обозначения категорий используются следующие константы модуля `locale`.

`LC_COLLATE`
: Сортировка строк. Влияет на функции `strcoll` и `strxfrm` модуля `locale`.

`LC_CTYPE`
: Типы символов. Влияет на аспекты модуля `string` (и методы модуля `string`), связанные с различением верхнего и нижнего регистров букв.

`LC_MESSAGES`
: Сообщения. Может влиять на отображение сообщений операционной системы — например, на работу функции `os.strerror` и модуля `gettext`.

`LC_MONETARY`
: Форматирование денежных значений. Влияет на функцию `locale.localeconv`.

LC_NUMERIC
: Форматирование чисел. Влияет на функции `atoi`, `atof`, `format`, `localeconv` и `str` модуля `locale`.

LC_TIME
: Форматирование значений времени и даты. Влияет на функцию `time.strftime`.

Настройки одних категорий (`LC_CTYPE`, `LC_TIME` и `LC_MESSAGES`) влияют на поведение других модулей (`string`, `time`, `os` и `gettext`, в соответствии с приведенным выше описанием). Настройки других категорий (`LC_COLLATE`, `LC_MONETARY` и `LC_NUMERIC`) влияют на функции самого модуля `locale`.

Описанные ниже функции обеспечивают опрос, изменение и манипулирование локалями и реализуют региональные соглашения, касающиеся локальных категорий `LC_COLLATE`, `LC_MONETARY` и `LC_NUMERIC`.

atof `atof(`*s*`)`

Преобразует строку *s* в число с плавающей точкой, используя текущие настройки `LC_NUMERIC`.

atoi `atoi(`*s*`)`

Преобразует строку *s* в целое число, используя настройки `LC_NUMERIC`.

format `format(`*fmt*`, `*num*`, `*grouping*`=False)`

Возвращает строку, полученную путем форматирования числа `num` в соответствии со строкой формата *fmt* и настройками `LC_NUMERIC`. Если не учитывать региональные соглашения, то результат выглядит как *fmt*`%`*num*. Если аргумент *grouping* имеет истинное значение, то функция `format` группирует цифры в результирующей строке в соответствии с настройками `LC_NUMERIC`.

```
>>> locale.setlocale(locale.LC_NUMERIC, 'en')
'English_United States.1252'
>>> locale.format('%s', 1000*1000)
'1000000'
>>> locale.format('%s', 1000*1000, grouping=True)
'1,000,000'
```

Если для чисел задана локаль U.S. English, а аргумент *grouping* имеет истинное значение, то метод `format` разбивает число на группы по три цифры в каждой, используя запятые для разделения групп.

getdefaultlocale `getdefaultlocale(`*envvars*`=('LANGUAGE', 'LC_ALL', 'LC_TYPE','LANG'))`

Проверяет значения переменных среды, имена которых перечислены в переменной *envvars*. Первая из обнаруженных переменных определяет локаль, используемую по умолчанию. Функция `getdefaultlocale` возвращает пару строк (*язык*, *кодировка*),

совместимых с требованием документа RFC 1766 (за исключением локали 'C'), таких как ('en_US', 'ISO8859-1'). Каждый элемент пары может иметь значение None, если функция gedefaultlocale не может определить, какое значение он должен иметь.

getlocale

getlocale(*category*=LC_CTYPE)

Возвращает пару строк (*язык*, *кодировка*) с текущими настройками для указанной категории. Указывать категорию LC_ALL нельзя.

localeconv

localeconv()

Возвращает словарь *d* с региональными соглашениями, заданными категориями LC_NUMERIC и LC_MONETARY текущей локали. В то время как настройки LC_NUMERIC лучше всего использовать косвенно, посредством других функций модуля locale, детальные настройки категории LC_MONETARY доступны только через *d*. Денежное форматирование при локальном и международном использовании может иметь разный смысл. Например, символ '$' означает денежную единицу США только при локальном использовании. При международном использовании символ '$' неоднозначен, поскольку точно так же обозначаются другие виды валют, называемые "долларами" (Канада, Австралия, Гонконг и др.). Поэтому при международном использовании денежная единица США однозначно идентифицируется строкой 'USD'. В словаре *d* ключами, используемыми при форматировании денежных значений, служат следующие строки.

'currency_symbol'

Символ денежной единицы при локальном использовании.

'frac_digits'

Количество цифр дробной части числа при локальном использовании.

'int_curr_symbol'

Символ денежной единицы при международном использовании.

'int_frac_digits'

Количество цифр дробной части числа при международном использовании.

'mon_decimal_point'

Строка, используемая в качестве "десятичной точки" для денежных значений.

'mon_grouping'

Список чисел, определяющих группирование цифр в денежных значениях.

'mon_thousands_sep'

Строка, используемая в качестве разделителя групп в денежных значениях.

'negative_sign' 'positive_sign'

Строки, используемые в качестве символов знака для отрицательных (положительных) денежных значений.

'n_cs_precedes' 'p_cs_precedes'

Значению True соответствует размещение символа валюты перед отрицательными (положительными) денежными значениями.

'n_sep_by_space' 'p_sep_by_space'

Значению True соответствует размещение пробела между знаком и отрицательными (положительными) денежными значениями.

'n_sign_posn' 'p_sign_posn'

Числовые коды, используемые для форматирования отрицательных (положительных) денежных значений:

0 — значение и символ валюты заключаются в круглые скобки;

1 — знак помещается перед значением и символом валюты;

2 — знак помещается после значения и символа валюты;

3 — знак помещается непосредственно перед значением;

4 — знак помещается непосредственно после значения.

CHAR_MAX

Текущая локаль не определяет никаких соглашений относительно этого вида форматирования.

d['mon_grouping'] — список чисел, определяющих группирование цифр при форматировании денежных значений. Если значение d['mon_grouping'][-1] равно 0, дальнейшее группирование после указанного количества цифр не выполняется. Если значение d['mon_grouping'][-1] равно locale.CHAR_MAX, группирование продолжается неограниченно, как если бы бесконечно продолжались значения d['mon_grouping'][-2]. locale.CHAR_MAX — константа, используемая в качестве значения для всех записей в *d*, для которых текущая локаль не определяет никаких соглашений.

normalize normalize(*localename*)

Возвращает строку, пригодную для использования в качестве аргумента метода setlocale, которая является нормализованным эквивалентом строки localename. Если метод normalize не может нормализовать строку localename, он возвращает ее в исходном виде.

resetlocale resetlocale(*category*=LC_ALL)

Устанавливает для локали категории *category* значение, предоставляемое функцией getdefaultlocale.

setlocale `setlocale(category, locale=None)`

Устанавливает для локали категории *category* значение *locale*, если оно не равно None; возвращает настройки (существующие, если значение *locale* равно None, и новые в противном случае). Аргумент locale может быть строкой или парой (*язык*, *кодировка*). Обычно *язык* — код языка в соответствии с набором стандартов двухбуквенных кодов ISO 639 ('en' — английский, 'uk' — украинский и т.п.). Если аргумент *locale* — пустая строка (''), то функция setlocale устанавливает пользовательскую локаль по умолчанию.

str `str(num)`

Аналогична функции `locale.format('%f', num)`.

strcoll `strcoll(str1, str2)`

Подобна функции `cmp(str1, str2)`, но работает в соответствии с настройками LC_COLLATE.

strxfrm `strxfrm(s)`

Возвращает строку *sx*, такую, что встроенная операция сравнения преобразованных таким способом строк эквивалентна вызову функции `locale.strcoll` для двух исходных строк. Функция strxfrm позволяет использовать аргумент *key=* при сравнении строк с учетом требований текущей локали.

```
def locale_sort(list_of_strings):
    list_of_strings.sort(key=locale.strxfrm)
```

Модуль gettext

Важным элементом интернационализации является предоставление возможности использования текстов, подготовленных на разных национальных языках, — задача, известная как *локализация*. В Python поддержка локализации обеспечивается модулем gettext, получившем свое название по аналогии с утилитой GNU gettext. Модуль gettext может использовать инфраструктуру и API последней, но дополнительно предлагает высокоуровневые средства, поэтому для эффективного использования модуля gettext в Python вам вовсе не придется предварительно устанавливать и изучать утилиту GNU gettext.

Полное обсуждение различных аспектов использования модуля gettext можно найти в онлайн-документации (https://docs.python.org/3/library/gettext.html).

Использование модуля gettext в задачах локализации

Модуль gettext не имеет ничего общего с автоматическим переводом текстов с одного языка на другой. Он лишь оказывает содействие в извлечении и организации текстовых сообщений на разных языках, используемых в вашей программе. Вместо того чтобы непосредственно использовать какой-либо строковый литерал (или,

как говорят, *сообщение*), подлежащий переводу на другой язык, передайте его функции _ (символ подчеркивания). Обычно модуль gettext устанавливает функцию _ во встроенном модуле __builtin__. Чтобы гарантировать нормальное выполнение программы как с использованием, так и без использования модуля gettext, включите в нее условное определение функции _, которая всего лишь возвращает переданный ей аргумент, не выполняя никаких других действий. После этого вы сможете безопасно использовать вызов _('*сообщение*') везде, где использовали бы литерал '*сообщение*', нуждающийся в переводе на другой язык, если такая возможность предоставляется. Ниже приведен пример того, с чего может начинаться код модуля, предназначенного для условного использования модуля gettext.

```
try:
    _
except NameError:
    def _(s): return s

def greet():
    print(_('Hello world'))
```

Если какой-то другой модуль установил модуль gettext еще до того, как вы запустили этот пример, то функция greet выведет соответствующим образом локализованное приветствие. В противном случае будет выведена исходная строка 'Hello world'.

Отредактируйте свой исходный код, декорируя сообщения функцией _. После этого используйте любой из доступных инструментов для извлечения сообщений в текстовый файл (обычно файл *messages.pot*) и передайте этот файл специалистам, чтобы они перевели сообщения на различные национальные языки, которые должно поддерживать ваше приложение. Для извлечения сообщений из своего исходного кода на Python можете воспользоваться сценарием *pygettext.py* (находится в каталоге *Tools/i18n* дистрибутива Python).

Каждый переводчик отредактирует свой файл *messages.pot*, получив в конечном счете файл с переведенными на другой язык сообщениями, и сохранит его в файле с расширением *.po*. Скомпилируйте *.po*-файлы в двоичные файлы с расширением *.mo*, пригодные для выполнения быстрого поиска, используя доступные инструменты. Для этих целей Python предоставляет сценарий *Tools/i18n/msgfmt.py*. Наконец, установите каждый из *.mo*-файлов в подходящем каталоге, присвоив им подходящие имена.

На разных платформах используются разные соглашения относительно того, какие каталоги и имена являются подходящими. По умолчанию модуль gettext использует подкаталог *share/locale/<язык>/LC_MESSAGES/* каталога *sys.prefix*, где *<язык>* — код языка (двухбуквенный). Каждый файл называется *<имя>.mo*, где *<имя>* — имя вашего приложения или пакета.

Подготовив и установив необходимые *.mo*-файлы, вы обычно будете выполнять следующий код во время запуска своего приложения:

```
import os, gettext
os.environ.setdefault('LANG', 'en') # язык по умолчанию
gettext.install('имя_вашего_приложения')
```

Тем самым будет гарантировано, что вызовы наподобие `_('сообщение')` будут возвращать соответствующим образом переведенные сообщения. Для доступа к функциональности модуля `gettext` в своей программе можете выбирать те или иные варианты в зависимости от того, что вам нужно. Например, вам может потребоваться также локализация C-расширений или переключение между языками во время выполнения программы. Еще один важный момент заключается в том, локализуете ли вы все приложение целиком или только отдельно распространяемый пакет.

Часто используемые функции модуля `gettext`

Модуль `gettext` предоставляет множество функций. Ниже описаны наиболее часто используемые из них.

`install` `install(domain, localedir=None, unicode=False)`

Устанавливает во встроенном пространстве имен Python функцию `_`, которая обеспечивает доступ к переведенным сообщениям, хранящимся в файле *<язык>/LC_MESSAGES/<домен>.mo* в каталоге *localedir*, используя код языка *<язык>* в соответствии с настройками локали, возвращаемыми функцией `getdefaultlocale`. Если аргумент *localedir* имеет значение `None`, то функция `install` использует каталог `os.path.join(sys.prefix, 'share', 'locale')`. Если аргумент *unicode* имеет истинное значение, то функция `_` принимает и возвращает строки Unicode, а не байтовые строки.

`translation` `translation(domain, localedir=None, languages=None)`

Выполняет поиск *.mo*-файла аналогично функции `install`. Если аргумент *languages* имеет значение `None`, то функция `translation` выполняет поиск используемого языка в окружении аналогично функции `install`. Она последовательно проверяет переменные среды `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, `LANG`. Первое найденное непустое значение разбивается на элементы с использованием двоеточия (`':'`) в качестве разделителя для получения списка имен языков (например, строка `'de:en'` разбивается на элементы `['de', 'en']`). Если значение аргумента *languages* отлично от `None`, то аргумент *languages* должен быть списком,

включающим одно или несколько имен языков (например, `['de', 'en']`). Для перевода используется первое из имен языков, для которого удается найти файл *.mo*. Функция `translation` возвращает объект, который предоставляет методы `gettext` (для перевода байтовых строк), `ugettext` (для перевода строк Unicode) и `install` (для установки функции `gettext` или `ugettext` под именем `_` во встроенном пространстве имен Python).

Функция `translation` обеспечивает более детальный контроль по сравнению с функцией `install`, действуя как `translation(domain, localedir).install(unicode)`. Используя функцию `translation`, вы сможете локализовать одиночный пакет, не

влияя на встроенное пространство имен, посредством связывания имени _ для каждого модуля по отдельности, например с помощью следующего кода:

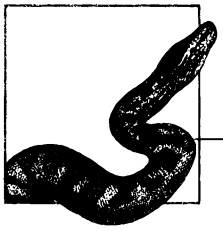
```
_ = translation(domain).ugettext
```

Кроме того, функция `translation` обеспечивает глобальное переключение между несколькими языками, поскольку вы можете передать ей явный аргумент *languages*, сохранить результирующий экземпляр, а затем вызывать метод `install` для нужного языка в случае необходимости.

```
import gettext
trans = {}
def switch_to_language(lang, domain='my_app',
use_unicode=True):
    if lang not in translators:
        trans[lang] = gettext.translation(domain,
languages=[lang])
    trans[lang].install(use_unicode)
```

Дополнительные ресурсы интернационализации

Интернационализация — весьма обширная тема. Для общего ознакомления с ней прочитайте соответствующие статьи в Википедии (`https://ru.wikipedia.org/wiki/Интернационализация`, `https://ru.wikipedia.org/wiki/Локализация_программного_обеспечения`). Одним из лучших ресурсов, содержащих код и информацию, относящиеся к интернационализации программ, является пакет ICU (`site.icu-project.org`), включающий базу данных локалей Common Locale DataRepository (CLDR), созданную консорциумом Unicode, и код для доступа к CLDR. Для использования ICU в Python следует установить пакет PyICU сторонних производителей (`https://pypi.python.org/pypi/PyICU/`).



11

Базы данных и постоянное хранение

Python поддерживает несколько способов постоянного хранения данных, одним из которых является *сериализация*, когда данные рассматриваются как коллекции объектов Python. Эти объекты можно *сериализовать* (сохранить) в виде байтового потока, а затем *десериализовать* (загрузить и восстановить их в первоначальном виде) из потока. *Персистентность объектов*, т.е. их способность существовать дольше, чем создавший их процесс, реализуется поверх поддержки сериализации, добавляя, например, такие возможности, как именование объектов. В этой главе рассматриваются модули Python, поддерживающие сериализацию и персистентность объектов.

Другим способом обеспечения персистентности данных является их сохранение в базах данных (БД). Простейшими БД могут служить файлы, формат которых позволяет использовать *доступ по ключу* для селективного чтения и обновления нужных данных. В этой главе рассматриваются модули стандартной библиотеки Python, поддерживающие несколько разновидностей формата подобного рода, известного как *DBM*.

Системы управления базами данных (СУБД) реляционного типа, такие как PostgreSQL или Oracle, предлагают более мощный подход к хранению, поиску и извлечению постоянно хранящихся данных. Реляционные БД используют диалекты языка структурированных запросов SQL для создания и изменения схемы БД, вставки и обновления данных, а также опроса БД на основании критериев поиска. (Ссылки на справочные материалы относительно SQL в этой книге не приводятся. Для получения необходимых сведений рекомендуем обратиться к книге Кевина Кляйна *SQL In a Nutshell*). К сожалению, несмотря на существование стандартов SQL, не существует двух таких СУБД, которые использовали бы в точности один и тот же диалект SQL.

Интерфейсы СУБД не включены в состав стандартной библиотеки Python. Однако существует множество сторонних модулей, с помощью которых программы

на языке Python могут обращаться к конкретным СУБД. Большинство этих модулей следует стандарту Python Database API 2.0 (https://www.python.org/dev/peps/pep-0249/), известному как *DBAPI*. В этой главе рассматривается стандарт DBAPI и упоминаются некоторые из наиболее популярных сторонних модулей, которые его реализуют.

Из всех DBAPI-модулей особенно удобен — в силу того, что он включается в каждую установку Python, — модуль sqlite3 (https://docs.python.org/3/library/sqlite3.html), реализующий интерфейс к библиотеке SQLite (www.sqlite.org) — автономной, не требующей подключения к серверу, нуждающейся лишь в минимальной настройке и наиболее широко используемой во всем мире реляционной СУБД. Модулю sqlite3 посвящен раздел "SQLite".

Помимо реляционных баз данных и более простых решений, рассмотренных в этой главе, существуют такие специфические для Python объектно-ориентированные реляционные СУБД, как ZODB (http://www.zodb.org/en/latest/), а также ряд баз данных стандарта NoSQL (http://nosql-database.org/), для каждой из которых предусмотрены интерфейсы к Python. Эти более совершенные нереляционные БД в книге не рассматриваются.

Сериализация

Python поставляется с несколькими модулями, обеспечивающими *сериализацию* (сохранение) объектов Python в байтовые потоки различного типа и их *десериализацию* (загрузку и восстановление) из потоков. Эквивалентными названиями процесса сериализации являются *маршализация* и "форматирование для обмена данными".

Существует множество подходов к сериализации данных, от не зависящего от языка JSON до низкоуровневого, специфического для версий Python модуля marshal, которые ограничены элементарными типами данных. Более богатые возможности предлагает специфический для Python подход на основе модуля pickle, а также такие кросс-языковые форматы, как XML, YAML (http://yaml.org/), Protocol Buffers (https://developers.google.com/protocol-buffers/) и MessagePack (msgpack.org).

В этом разделе мы обсудим модули json и pickle. Рассмотрению XML посвящена глава 23. Модуль marshal слишком низкоуровневый для того, чтобы его можно было использовать в приложениях. Если он вам все же потребуется для сопровождения существующего кода, обратитесь к онлайн-документации. Что касается протоколов Protocol Buffers, MessagePack, YAML и других подходов к организации обмена данными и их сериализации (каждый из которых имеет свои преимущества и недостатки), то они не могут быть рассмотрены в рамках данной книги. Для их изучения следует обратиться к ресурсам, доступным в Интернете (https://ru.wikipedia.org/wiki/Сериализация).

Модуль json

Модуль json стандартной библиотеки предоставляет следующие четыре ключевые функции.

dump `dump(value, fileobj, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=JSONEncoder, indent=None, separators=(', ', ': '), encoding='utf-8', default=None, sort_keys=False, **kw)`

Функция dump записывает сериализованное в формате JSON значение объекта *value* в файловый объект *fileobj*, который должен быть открыт для записи в текстовом режиме, посредством вызова метода *fileobj*.write. В версии v3 методу *fileobj*.write передается аргумент в виде текстовой строки (Unicode); в версии v2 передается простая (байтовая) строка, полученная с использованием кодировки *encoding*. Аргумент encoding допустим лишь в версии v2: в версии v3 функция dump не преобразует строки.

Если аргумент *skipkeys* имеет значение True (по умолчанию оно равно False), то ключи словарей, не являющиеся скалярами (т.е. не относящиеся к типам bool, float, int, str или None, а в версии v2 — не относящиеся к типам long и unicode), игнорируются без возбуждения исключений. В обеих версиях ключи, являющиеся скалярами, преобразуются в строки (например, значение None превращается в "null"): в отображениях JSON ключами могут служить только строки.

Если аргумент *ensure_ascii* имеет значение True (задано по умолчанию), то все символы, не являющиеся символами ASCII, игнорируются. Если он равен False, то символы выводятся в том виде, в каком они существуют.

Значение аргумента *check_circular*, равное True (задано по умолчанию), запускает проверку наличия циклических ссылок, и в случае обнаружения таковых возбуждается исключение ValueError. Если же это значение равно False, то указанная проверка не выполняется, что может привести к возбуждению всевозможных исключений (вплоть до краха программы).

Если аргумент *allow_nan* имеет значение True (задано по умолчанию), то скаляры в виде чисел с плавающей точкой, представляемые значениями nan, inf и -inf, будут выводиться в виде соответствующих им эквивалентных значений NaN, Infinity и -Infinity JavaScript. Если же этот аргумент равен False, то наличие указанных скаляров приведет к возбуждению исключения ValueError.

Также можно передать необязательный аргумент *cls*, чтобы использовать пользовательский подкласс JSONEncoder (необходимость в подобном собственном подклассе возникает редко, и в данной книге она не рассматривается). В этом случае именованные аргументы ***kw* передаются конструктору класса *cls* при создании его экземпляра. По умолчанию для преобразования используется непосредственно класс JSONEncoder.

Если значение аргумента *indent* представляет собой положительное целое число, то функция dump форматирует вывод, добавляя соответствующее количество пробелов перед каждым элементом массива и атрибутом объекта для создания более удобочитаемого вывода. Если это значение <=0, то вставляются символы \n.

Если оно равно None (значение по умолчанию), функция dump использует наиболее компактное представление вывода. В версии v3 аргументом *indent* может быть строка, например '\t', которая используется для создания отступов.

Аргументом *separators* должна быть пара (кортеж из двух элементов), представленная строками, используемыми в качестве символов-разделителей между элементами массивов и между ключами и значениями соответственно. Этот аргумент можно задать явным образом в виде separators=(',',':') для того, чтобы избежать появления лишних пробелов в выводе.

Также можно задать необязательный аргумент *default*, чтобы преобразовать некоторые несериализуемые объекты в сериализуемые. Аргумент *default* — это функция с единственным аргументом, представляющим несериализуемый объект, которая должна вернуть сериализуемый объект или, если это невозможно, возбудить исключение ValueError (по умолчанию, если встречается несериализуемый объект, возбуждается исключение ValueError).

Если аргумент *sort_keys* имеет значение True (по умолчанию — False), то отображения выводятся в порядке сортировки по ключам. Если он имеет значение False, то отображения выводятся в том порядке, в котором они встречаются при итерировании по ним (в основном в случайном порядке для словарей, но порядком итерирования можно управлять, используя упорядоченные словари типа collections.OrderedDict).

dumps

dumps(*value*, *skipkeys*=False, *ensure_ascii*=True, *check_circular*=True, *allow_nan*=True, *cls*=JSONEncoder, *indent*=None, *separators*=(' ', ': '), *encoding*='utf-8', *default*=None, *sort_keys*=False, ***kw*)

Функция dumps возвращает строку, являющуюся JSON-сериализацией значения объекта, т.е. строку, которую функция dump записала бы в аргумент, задающий файловый объект. Все аргументы функции dumps имеют точно тот же смысл, что и в функции dump.

load

JSON сериализует только по одному объекту в каждый файл

JSON не относится к числу покадровых форматов. Это означает, что невозможно вызвать функцию dump более одного раза для сериализации нескольких объектов в один и тот же файл, равно как невозможно вызвать функцию load более одного раза для десериализации объектов, что возможно с помощью модуля pickle. Поэтому технически JSON позволяет сериализовать только по одному объекту в каждый файл. Однако ничто не мешает создать один объект в виде списка или словаря, который, в свою очередь, может содержать любое желаемое количество объектов.

load(*fileobj*, *encoding*='utf-8', *cls*=JSONDecoder, *object_hook*=None, *parse_float*=float, *parse_int*=int, *parse_constant*=None, *object_pairs_hook*=None, ***kw*)

Функция load создает и возвращает объект *v*, ранее сериализованный в файловый объект *fileobj*, который должен быть открыт для чтения в текстовом режиме, получая содержимое *fileobj* посредством вызова метода fileobj.read.

В версии v3 вызов *fileobj*.read должен возвращать текстовую строку (Unicode). В версии v2 он также может возвратить простую (байтовую) строку, которая декодируется с использованием кодировки *encoding*. Аргумент *encoding* допустим лишь в версии v2: в версии v3 функция load не выполняет никаких текстовых преобразований.

Функции load и dump дополняют друг друга. Иными словами, вызов функции load(*f*) десериализует значение, которое было сериализовано ранее в результате создания содержимого *f* с помощью единственного вызова dump(*v*, *f*) (возможно, с некоторыми изменениями: например, все ключи в словарях преобразуются в строки).

Также можно передать необязательный аргумент *cls*, что дает возможность использовать пользовательский подкласс JSONEncoder (необходимость в собственном подклассе возникает редко, и в данной книге эта возможность не рассматривается). В этом случае именованные аргументы ***kw* передаются конструктору класса *cls* при его инстанциализации. По умолчанию для декодирования используется непосредственно класс JSONEncoder.

Кроме того, можно передать необязательный аргумент *object_hook* или *object_pairs_hook* (если передаются оба аргумента, то аргумент *object_hook* игнорируется и используется только аргумент *object_pairs_hook*), который определяет функции, позволяющие реализовать собственный декодер. Если передается аргумент *object_hook*, но не *object_pairs_hook*, то каждый раз, когда объект декодируется в словарь, функция load вызывает *object_hook* с этим словарем в качестве единственного аргумента и использует возвращаемое *object_hook* значение вместо словаря. Если же передается аргумент *object_pairs_hook*, то каждый раз, когда декодируется объект, функция load вызывает *object_pairs_hook* с единственным аргументом в виде списка пар (*key*, *value*) элементов объекта в том порядке, в каком они встречаются во входных данных, и использует возвращаемое *object_pairs_hook* значение. Это позволяет выполнять специальное декодирование, которое может зависеть от того порядка, в каком пары (*key*, *value*) встречаются среди входных данных.

parse_float, *parse_int* и *parse_constant* — это функции, вызываемые с единственным аргументом, который является строкой, представляющей значения типа float, int или одну из трех специальных констант: 'NaN','Infinity' и '-Infinity'. Функция load вызывает соответствующую функцию всякий раз, когда она идентифицирует среди входных данных строку, представляющую число, и использует возвращаемое этой функцией значение. Значением по умолчанию для аргумента *parse_float* является функция float, для аргумента *parse_int* — int, а функция *parse_constant* возвращает одно из трех специальных скалярных значений с плавающей точкой nan, inf и -inf, в зависимости от ситуации. Например, если требуется, чтобы вместо чисел с плавающей точкой были возвращены десятичные числа с фиксированной точностью (раздел "Модуль decimal" в главе 15), следует передать аргумент parse_float=decimal.Decimal.

loads loads(*s*, *encoding*='utf-8', *cls*=JSONDecoder, *object_hook*=None, *parse_float*=float, *parse_int*=int, *parse_constant*=None, *object_pairs_hook*=None, ***kw*)

Функция loads создает и возвращает объект v, ранее сериализованный в строку *s*. Все аргументы функции loads имеют точно тот же смысл, что и в функции load.

Пример JSON

Предположим, вам нужно прочитать несколько текстовых файлов, имена которых заданы в качестве аргументов вашей программы, и записать, где именно в файлах встречаются различные слова. Таким образом, записью для каждого слова будет список пар (*имя_файла*, *номер_строки*). В следующем примере для кодирования списков пар (*имя_файла*, *номер_строки*) в виде строк и их сохранения в DBM-файле (раздел "Модули DBM") используется модуль `json`. Поскольку эти списки содержат кортежи, каждый из которых включает строку и число, модуль `json` вполне может справиться с их сериализацией.

```
import collections, fileinput, json, dbm
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = fileinput.filename(), fileinput.filelineno()
    for word in line.split():
        word_pos[word].append(pos)
dbm_out = dbm.open('indexfilem', 'n')
for word in word_pos:
    dbm_out[word] = json.dumps(word_pos[word])
dbm_out.close()
```

(В версии v3 функция `dbm.open` является менеджером контекста, поэтому мы могли бы выделить отступом второй цикл `for` в качестве тела инструкции `with dbm.open('indexfilem', 'n') as dbm_out:` и опустить инструкцию `dbm_out.close()`. Однако пример в том виде, в каком он приведен выше, работает в обеих версиях, v2 и v3, если не считать того, что в версии v2, чтобы гарантировать кросс-платформенность примера, следует импортировать и использовать не пакет `dbm`, а модуль `anydbm`, причем это замечание относится также к следующему примеру.) Модуль `json` нужен нам также для того, чтобы десериализовать данные, сохраненные в DBM-файле *indexfilem*, как показано в следующем примере.

```
import sys, json, dbm, linecache
dbm_in = dbm.open('indexfilem')
for word in sys.argv[1:]:
    if word not in dbm_in:
        print('Word {!r} not found in index file'.format(word),
            file=sys.stderr)
        continue
    places = json.loads(dbm_in[word])
    for fname, lineno in places:
        print('Word {!r} occurs in line {} of file {}:'.format(
            word,lineno,fname))
        print(linecache.getline(fname, lineno), end='')
```

Модули pickle и cPickle

Модули pickle и, в версии v2, cPickle предоставляют функции-фабрики Pickler и Unpickler, используемые для создания объектов, являющихся оболочками вокруг файловых объектов, а также специфические для Python механизмы сериализации.

В версии v2 различие между этими модулями заключается в том, что Pickler и Unpickler в модуле pickle — это классы, наследованием от которых можно создавать собственные объекты-сериализаторы, переопределяя методы в соответствии с необходимостью. С другой стороны, Pickler и Unpickler в модуле cPickle — это функции-фабрики, которые генерируют экземпляры ненаследуемых типов, а не классов. При этом производительность модуля cPickle выше, однако возможности наследования в нем отсутствуют. В оставшейся части раздела рассматривается модуль pickle, однако все, что будет обсуждаться, применимо также к модулю cPickle. Версия v3 включает только модуль pickle, который работает достаточно быстро и предоставляет Pickler и Unpickler в виде классов.

Сериализации также свойственны некоторые проблемы, присущие глубокому копированию (см. раздел "Модуль copy" в главе 7). Модуль pickle справляется с этими проблемами во многом так же, как модуль copy со своими. Как и глубокое копирование, сериализация подразумевает рекурсивный обход направленного графа ссылок. Модуль pickle сохраняет форму графа: если один и тот же объект встречается несколько раз, то он сериализуется только при обнаружении первого вхождения, тогда как для всех остальных вхождений объекта сериализуются лишь ссылки на это единственное значение. Кроме того, модуль pickle корректно сериализует графы с циклическими ссылками. В то же время это означает, что если изменяемый объект o сериализуется несколько раз в один и тот же экземпляр p класса Pickler, то изменения, внесенные в объект o после его первой сериализации в p, не будут сохранены.

Не изменяйте объекты в процессе сериализации

Ради обеспечения ясности, корректности и простоты кода не следует изменять сериализуемые объекты в процессе их сериализации экземпляром Pickler.

Модуль pickle может сериализовать данные, используя традиционный протокол ASCII или один из нескольких компактных двоичных протоколов. В версии v2 в интересах обратной совместимости протоколу ASCII соответствует протокол 0, используемый по умолчанию, но обычно лучше явно запрашивать двоичный протокол 2, который поддерживается в версии v2 и отличается наибольшей экономичностью в отношении времени и памяти. В версии v3 поддерживаются протоколы с номерами от 0 до 4 включительно. По умолчанию используется протокол 3, который обычно является разумным выбором, но можно явно указать протокол 2 (гарантирующий возможность загрузки сериализованных данных программами, ориентированными на использование версии v2) или протокол 4, несовместимый с ранними версиями, но обеспечивающий преимущества в производительности для очень крупных объектов.

Работая с модулем pickle, всегда используйте протокол 2 или выше

Всегда указывайте по крайней мере протокол 2. Существенное влияние на выбор протокола оказывают размер и скорость сохранения данных, причем у двоичного формата отсутствуют существенные недостатки, за исключением потери совместимости с действительно устаревшими версиями Python.

В процессе повторной загрузки объектов модуль pickle прозрачно распознает и применяет протокол, поддерживаемый версией Python, которую вы используете.

Модуль pickle сериализует классы и функции по именам, а не по значениям, и поэтому он может десериализовать класс или функцию, лишь импортируя их из того же модуля, в котором класс или функция были найдены при их сериализации с помощью модуля pickle. В частности, модуль pickle может нормально сериализовать и десериализовать классы и функции, только если они являются именами верхнего модуля в своем модуле (т.е. атрибутами своего модуля). Рассмотрим, например, следующий код.

```
def adder(augend):
    def inner(addend, augend=augend): return addend+augend
    return inner
plus5 = adder(5)
```

Этот код связывает с именем plus5 замыкание (см. раздел "Вложенные функции и вложенные области видимости" в главе 4) — вложенную функцию inner плюс соответствующую внешнюю область видимости. Поэтому при попытке сериализации plus5 возбуждается исключение pickle.PicklingError (это относится к версии v2; в версии v3 возникает ошибка AttributeError): модуль pickle может сериализовать лишь функцию верхнего уровня, тогда как функция inner, чье замыкание привязано к имени plus5 в этом коде, не является таковой и вложена в функцию adder. Описанные проблемы в равной степени касаются как вложенных функций, так и вложенных классов (т.е. классов, определенных не на верхнем уровне).

Функции и классы модулей pickle и cPickle

Модуль pickle (и, только в версии v2, модуль cPickle) предоставляет следующие функции и классы.

dump, dumps

```
dump(value, fileobj, protocol=None, bin=None)
dumps(value, protocol=None, bin=None)
```

Функция dumps возвращает представление значения объекта в виде байтовой строки. Функция dump записывает ту же строку в файловый объект *fileobj*, который должен быть открыт для записи. Вызов dump(*v*, *f*) эквивалентен вызову *f*.write(dumps(*v*)). Передавать параметр *bin*, который существует лишь для обеспечения совместимости с предыдущими версиями Python, не следует. Параметр *protocol* может принимать значения 0 (значение по умолчанию в версии v2, предусмотренное из

соображений совместимости; ему соответствует наиболее медленный и объемный ASCII-вывод), 1 (двоичный вывод, совместимый с очень старыми версиями Python) или 2 (наиболее быстрый и экономичный вывод в версии v2); в версии v3 также возможно значение 3 или 4. В версии v2 мы рекомендуем всегда использовать значение 2. В версии v3 используйте значение 2, если необходимо обеспечить совместимость результатов с версией v2, иначе — 3 (значение по умолчанию в версии v3) или 4. Кроме случаев, когда аргумент *protocol* задан равным 0 или (в версии v2) вообще не задан, что соответствует ASCII-выводу, объект, определяемый параметром *fileobj* функции dump, должен быть открыт в режиме двоичной записи.

load, loads `load(`*fileobj*`) loads(`*s*`)` в версии v2; в версии v3 — `loads(`*s*`, *, `*fix_imports*`=True, `*encoding*`="ASCII", `*errors*`="strict")`

Функции load и dump дополняют друг друга. Иными словами, последовательность вызовов функции `load(`*f*`)` десериализует значения, которые были ранее сериализованы в результате создания содержимого *f* с помощью последовательности `dump(`*v*`, `*f*`)`. Функция load читает соответствующее количество байтов из файлового объекта *fileobj*, и возвращает объект *v*, представляющий прочитанные байты. Функции load и loads прозрачно поддерживают данные, сериализованные с помощью модулей pickle и cPickle с использованием любого двоичного или ASCII-протокола. В случае данных, сериализованных с использованием любого двоичного формата, файл должен быть открыть в двоичном режиме для обеих функций, dump и load. Вызов `load(`*f*`)` эквивалентен вызову `Unpickler(`*f*`).load()`. Функция loads создает и возвращает объект *v*, представленный байтовой строкой *s*. Поэтому для любого объекта *v* поддерживаемого типа выполняется условие `v==loads(dumps(`*v*`))`. Если строка *s* длиннее, чем `dumps(`*v*`)`, функция loads игнорирует дополнительные байты. Допустимые только в версии v3 необязательные аргументы *fix_imports*, *encoding* и *errors* предоставляются для обработки потоков, сгенерированных в версии v2. См. документацию функции pickle.loads версии v3 (https://docs.python.org/3/library/pickle.html#pickle.loads).

Не десериализуйте с помощью модуля pickle данные из ненадежных источников

Десериализация данных из источников, которые не являются доверенными, чревата угрозами безопасности. Атакующая сторона может использовать эту уязвимость для выполнения произвольного кода.

Pickler `Pickler(`*fileobj*`, `*protocol*`=None, `*bin*`=None)`

Создает и возвращает такой объект *p*, что вызов `p.dump` эквивалентен вызову dump с аргументами *fileobj*, *protocol* и *bin*, переданными классу Pickler. В случае сериализации большого количества объектов в файл класс Pickler более удобен и работает быстрее, чем повторные вызовы dump. Вы можете создать подкласс pickle.Pickler, чтобы

переопределить методы `Pickler` (в частности, метод `persistent_id`) и создать собственный фреймворк персистентности. Однако это более сложная тема, и в данной книге она не рассматривается.

`Unpickler` `Unpickler(`*`fileobj`*`)`

Создает и возвращает такой объект *u*, что вызов *u*`.load` эквивалентен вызову функции `load` с аргументом *`fileobj`*, переданным классу `Unpickler`. В случае десериализации большого количества объектов из файла класс `Unpickler` более удобен и работает быстрее, чем повторные вызовы функции `load`. Вы можете создать подкласс `pickle.Unpickler`, чтобы переопределить методы `Unpickler` (в частности, метод `persistent_load`). Но это более сложная тема, и в данной книге она не рассматривается.

Пример использования модуля `pickle`

В следующем примере решается та же задача, что и в приведенном ранее примере с модулем `json`, но теперь вместо модуля `json` для сериализации списка пар (*`имя_файла`*, *`номер_строки`*) в виде строк используется модуль `pickle`.

```
import collections, fileinput, pickle, dbm
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = fileinput.filename(), fileinput.filelineno()
    for word in line.split():
        word_pos[word].append(pos)
dbm_out = dbm.open('indexfilep','n')
for word in word_pos:
    dbm_out[word] = pickle.dumps(word_pos[word],2)
dbm_out.close()
```

Затем мы можем использовать модуль `pickle` для того, чтобы прочитать обратно данные, сохраненные в объекте *indexfilep*, подобном DBM-файлу, как показано в следующем примере.

```
import sys, pickle, dbm, linecache
dbm_in = dbm.open('indexfilep')
for word in sys.argv[1:]:
    if word not in dbm_in:
        print('Word {!r} not found in index file'.format(word),
            file=sys.stderr)
        continue
    places = pickle.loads(dbm_in[word])
    for fname, lineno in places:
        print('Word {!r} occurs in line {} of file {}:'.format(
              word,lineno,fname))
        print(linecache.getline(fname, lineno), end='')
```

В версии v2, чтобы гарантировать кросс-платформенность обоих примеров, следует импортировать и использовать не пакет `dbm`, а модуль `anydbm`.

Сериализация и десериализация экземпляров с помощью модуля pickle

Чтобы повторно загрузить сериализованный экземпляр *x*, модуль pickle должен иметь возможность импортировать класс этого экземпляра из того же модуля, в котором данный класс был определен, когда модуль pickle сохранял экземпляр. Ниже описана последовательность операций, выполняемых при сохранении модулем pickle состояния экземпляра *x* класса *T* и последующей загрузке сохраненного состояния в новый экземпляр *y* класса *T* (первым шагом повторной загрузки всегда является создание нового пустого экземпляра *y* класса *T*, если только далее не оговорено иное).

- Если *T* предоставляет метод `__getstate__`, то модуль pickle сохраняет результат *d* вызова `T.__getstate__(x)`.
- Если *T* предоставляет метод `__setstate__`, то *d* может быть любого типа, и модуль pickle загружает сохраненное состояние посредством вызова `T.__setstate__(y, d)`.
- В противном случае *d* должен быть словарем, и модуль pickle просто устанавливает атрибут `y.__dict__ = d`.
- В противном случае, если *T* предоставляет метод `__getnewargs__`, а модуль pickle использует протокол 2 или более высокий, то pickle сохраняет результат *t* вызова `T.__getnewargs__(x)`. Объект *t* должен быть кортежем.
- В этом единственном случае модуль pickle начинает работу не с создания пустого объекта *y*, а создает *y* посредством вызова `y = T.__new__(T, *t)`, который завершает процесс загрузки.
- В противном случае модуль pickle по умолчанию сохраняет в качестве объекта *d* словарь `x.__dict__`.
- Если *T* предоставляет метод `__setstate__`, то модуль pickle загружает сохраненное состояние посредством вызова `T.__setstate__(y, d)`.
- В противном случае модуль pickle просто устанавливает атрибут `y.__dict__ = d`.

Все элементы объекта *d* или *t*, которые модуль pickle сохраняет и загружает (обычно в виде словаря или кортежа) должны, в свою очередь, быть экземплярами типов, пригодных для сериализации и десериализации с помощью модуля pickle, а описанная выше процедура в необходимых случаях должна допускать рекурсивное повторение до тех пор, пока модуль pickle не достигнет сериализуемых с помощью модуля pickle встроенных типов (словарей, кортежей, списков, множеств, чисел, строк и т.п.).

Как отмечалось в разделе "Модуль copy" главы 7, специальные методы `__getnewargs__`, `__getstate__` и `__setstate__` также управляют копированием и глубоким копированием экземпляров объектов. Если класс определяет атрибут `__slots__` и поэтому его экземпляры не имеют атрибута `__dict__`, то модуль

pickle пытается использовать все возможности для сохранения или восстановления словаря с именами и значениями, эквивалентными тем, которые содержатся в слотах. Однако такой класс обязательно должен определить методы __getstate__ и __setstate__, иначе сериализация или копирование его экземпляров не смогут быть корректно выполнены.

Использование модуля copy_reg для пользовательской адаптации процесса сериализации с помощью модуля pickle

Сериализацией и десериализацией объектов произвольного типа с помощью модуля pickle можно управлять посредством регистрации функции-фабрики и функции приведения типов, используя модуль copy_reg. В частности, это оказывается полезным в тех случаях, когда вы определяете тип с помощью C-расширения Python. Модуль copy_reg предоставляет следующие функции.

constructor constructor(*fcon*)

Добавляет *fcon* в таблицу конструкторов, содержащую все функции-фабрики, которые может вызывать модуль pickle. Аргумент fcon должен быть вызываемым объектом и обычно является функцией.

pickle pickle(*type*, *fred*, *fcon*=None)

Регистрирует функцию *fred* в качестве функции приведения для типа *type*, где аргумент *type* должен быть объектом типа. Чтобы сохранить объект *o* типа *type*, модуль *pickle* вызывает функцию *fred*(*o*) и сохраняет возвращаемый ею результат. Функция *fred*(*o*) должна возвращать кортеж (*fcon*, *t*) или (*fcon*, *t*, *d*), где *fcon* — конструктор, а *t* — кортеж. Чтобы загрузить объект *o*, модуль pickle использует вызов *o*=*fcon*(*t). Затем, когда функция *fred* вернет объект *d*, модуль pickle использует *d* для восстановления состояния *o* (если *o* предоставляет метод __setstate__ — *o*.__setstate__(*d*); в противном случае — *o*.__dict__.update(*d*)), о чем говорилось в разделе "Сериализация и десериализация экземпляров с помощью модуля pickle". Если аргумент *fcon* не равен None, модуль pickle также осуществляет вызов constructor(*fcon*) для регистрации *fcon* в качестве конструктора.

Модуль pickle, в отличие от модуля marshal, не поддерживает сериализацию объектов кода. Ниже приведен пример того, как настроить поддержку сериализации объектов кода с помощью модуля copy_reg, делегируя часть работы модулю marshal.

```
>>> import pickle, copy_reg, marshal
>>> def marsh(x): return marshal.loads, (marshal.dumps(x),)
...
>>> c=compile('2+2','','eval')
>>> copy_reg.pickle(type(c), marsh)
>>>s=pickle.dumps(c, 2)
>>> cc=pickle.loads(s)
>>> print eval(cc)
4
```

Использование модуля `marshal` делает ваш код зависимым от версии Python

Остерегайтесь использовать модуль `marshal` в своем коде, как это было сделано в предыдущем примере. Результаты сериализации с помощью модуля `marshal` могут изменяться от версии к версии, поэтому использование данного модуля означает, что возможны ситуации, когда объекты, сериализованные в одной версии Python, не смогут быть загружены в других версиях.

Модуль `shelve`

Модуль `shelve` организует работу модулей `pickle` (или `cPickle` в версии v2, если он доступен), `io` (в версии v3; в версии v2 — модуль `cStringIO`, если он доступен, иначе — `StringIO`) и `dbm` (и его базовых модулей для доступа к объектам архива, подобным DBM-файлам, о чем говорилось в разделе "Модули DBM"; но это в версии v3, тогда как в версии v2 это модуль `anydbm`) таким образом, чтобы обеспечить простой и легковесный механизм постоянного хранения.

Модуль `shelve` предоставляет функцию `open`, полиморфную по отношению к функции `anydbm.open`. Отображение *s*, возвращаемое функцией `shelve.open`, ограничено в меньшей степени, чем отображение *a*, возвращаемое функцией `anydbm.open`. Ключи и значения *a* должны быть строками. Ключи *s* также должны быть строками, но значения могут иметь любой тип, сериализуемый с помощью модуля `pickle`. Адаптация модуля `pickle` (`copy_reg`, `__getnewargs__`, `__getstate__` и `__setstate__`) также применима к модулю `shelve`, поскольку `shelve` делегирует выполнение сериализации модулю `pickle`.

Остерегайтесь хитрой ловушки, когда используете модуль `shelve` с изменяемыми объектами: если вы воздействуете на изменяемый объект, находящийся в хранилище `shelf`, изменения не будут "приняты", если вы не назначите измененный объект вновь тому же индексу. Соответствующий пример приведен ниже.

```
import shelve
s = shelve.open('data')
s['akey'] = list(range(4))
print(s['akey'])          # вывод: [0, 1, 2, 3]
s['akey'].append(9)       # попытка непосредственого изменения
print(s['akey'])          # "не принято"; вывод: [0, 1, 2, 3]

x = s['akey']             # извлечение объекта
x.append(9)               # внесение изменения
s['akey'] = x             # важный шаг: обратное сохранение объекта!
print(s['akey'])          # "принято"; вывод: [0, 1, 2, 3, 9]
```

Эту проблему можно обойти, передав именованный аргумент *writeback*=`True` функции `shelve.open`, однако имейте в виду, что от этого может серьезно пострадать производительность вашей программы.

Пример использования хранилища shelve

В следующем примере решается та же задача, что и в приведенных ранее примерах с модулями json и pickle, но для постоянного хранения списков пар (*имя_файла*, *номер_строки*) в нем используется модуль shelve.

```
import collections, fileinput, shelve
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = fileinput.filename(), fileinput.filelineno()
    for word in line.split():
        word_pos[word].append(pos)
sh_out = shelve.open('indexfiles','n')
sh_out.update(word_pos)
sh_out.close()
```

Для чтения данных, сохраненных в объекте *indexfiles*, подобном DBM-файлу, необходимо использовать модуль shelve.

```
import sys, shelve, linecache
sh_in = shelve.open('indexfiles')

for word in sys.argv[1:]:
    if word not in sh_in:
        print('Word {!r} not found in index file'.format(word),
            file=sys.stderr)
        continue
    places = sh_in[word]
    for fname, lineno in places:
        print('Word {!r} occurs in line {} of file {}:'.format(
            word,lineno,fname))
        print(linecache.getline(fname, lineno), end='')
```

Эти два примера являются самыми простыми и наиболее непосредственными из различных эквивалентных пар примеров, представленных в данном разделе, тем самым подтверждая тот факт, что модуль shelve — более высокоуровневый по сравнению с модулями, использованными в предыдущих примерах.

Модули DBM

DBM (https://ru.wikipedia.org/wiki/DBM) — это семейство библиотек, которые поддерживают файлы данных с парами строк (*ключ*, *данные*), обеспечивающих быстрое извлечение и сохранение данных с использованием значения ключа, т.е. тип доступа, известный как *доступ по ключу*. И хотя доступ по ключу значительно уступает реляционным БД по своим возможностям, он характеризуется меньшими накладными расходами, и в определенных ситуациях его может вполне хватить для удовлетворения потребностей программы. В таких случаях DBM-подобные

файлы позволяют уменьшить размер программы и ускорить ее работу по сравнению с использованием реляционных БД.

В версии v3 поддержка DBM в стандартной библиотеке Python организована чистым и элегантным способом: пакет dbm предоставляет две общие функции, а вместе с ними другие модули, находящиеся в том же пакете, которые поддерживают конкретные реализации. Эта же поддержка, предлагаемая в версии v2, будучи функционально эквивалентной, эволюционировала менее организованным образом, что в конечном счете привело к созданию коллекции высокоуровневых модулей, не объединенных в один пакет. В этом разделе мы сначала рассмотрим вариант организации поддержки DBM в версии v3, а затем перейдем к обсуждению реализации тех же возможностей в версии v2, ссылаясь на предыдущее рассмотрение для демонстрации функциональной эквивалентности обоих подходов.

Пакет v3 dbm

В версии v3 пакет dbm предоставляет две высокоуровневые функции.

open `open(filepath, flag='r', mode=0o666)`

Открывает или создает DBM-файл, заданный аргументом *filepath* (любой путь к файлу), и возвращает объект отображения, соответствующий DBM-файлу. Если DBM-файл уже существует, то функция open использует функцию whichdb для того, чтобы определить, какой модуль DBM может обработать этот файл. Если функция open создает новый DBM-файл, то она выбирает первый доступный подмодуль dbm в следующем порядке предпочтений: gnu, ndbm, dumb.

Аргумент *flag* — это односимвольная строка, сообщающая функции open, каким образом следует открыть файл и следует ли его создавать (табл. 11.1). Аргумент *mode* — это целое число, которое функция open использует в качестве битов разрешения доступа, если она создает файл (см. раздел "Создание объекта "файла" с помощью метода io.open" в главе 10).

Таблица 11.1. Значения флагов для функции dbm.open

Флаг	Только чтение?	Если файл существует	Если файл не существует
'r'	Да	open открывает файл	open возбуждает ошибку
'w'	Нет	open открывает файл	open возбуждает ошибку
'c'	Нет	open открывает файл	open создает файл
'n'	Нет	open усекает файл	open создает файл

Функция dbm.open возвращает объект отображения *m* с подмножеством функциональности словарей (см. раздел "Операции над словарями" в главе 3). Объект *m* допускает использование лишь строковых ключей и значений, а его единственными неспециальными методами отображения являются *m*.get, *m*.keys и *m*.setdefault. Для связывания, повторного связывания, открепления и обращения к элементам *m* можно использовать тот же синтаксис индексирования m[*key*], который используется в отношении словарей.

Если установлен флаг `'r'`, то *m* доступен только для чтения, и в этом случае можно лишь обращаться к элементам *m*, но не связывать, повторно связывать и откреплять их. Проверить, является ли строка *s* ключом в *m*, можно с помощью обычного выражения *s* `in` *m*. Итерировать непосредственно по *m* нельзя, но можно, что эквивалентно, итерировать по представлению `m.keys()`.

Одним из дополнительных методов, предоставляемых объектом *m*, является метод `m.close`, имеющий ту же семантику, что и метод `close` объектов "файлов". Точно так же, как и в случае объектов "файлов", следует обязательно вызывать метод `m.close()`, если объект *m* больше не используется. Чтобы гарантировать выполнение этой завершающей операции, можно использовать инструкцию `try/finally` (см. разделе "Инструкция `try/finally`" в главе 5), но еще лучше воспользоваться инструкцией `with`, рассмотренной в разделе "Функции" в главе 3 (это возможно, поскольку объект *m* является менеджером контекста).

whichdb `whichdb(`*filename*`)`

Открывает и читает файл, указанный аргументом *filename*, с целью определения подмодуля `dbm`, с помощью которого был создан данный файл. Функция `whichdb` возвращает значение `None`, если файл не существует или не может быть открыт и прочитан. Если же файл существует и может быть открыт и прочитан, но невозможно определить, с помощью какого подмодуля `dbm` он был создан (обычно это означает, что данный файл не является DBM-файлом), то функция `whichdb` возвращает пустую строку. Если удается определить, какой модуль в состоянии прочитать DBM-подобный файл, функция `whichdb` возвращает строку с именем подмодуля `dbm`, например: `'dbm.ndbm'`, `'dbm.dumbdbm'` или `'dbm.gdbm'`.

В версии v3 в дополнение к этим двум высокоуровневым функциям пакет `dbm` содержит такие специфические модули, как `ndbm`, `gnu` и `dumb`, предоставляющие различные реализации функциональности DBM, к которым обычно получают доступ только через описанные выше высокоуровневые функции пакета `dbm`. Сторонние пакеты могут устанавливать дополнительные реализации модулей в пакете `dbm`.

Единственной из реализаций модулей, входящих в пакет `dbm`, которая гарантированно существует на всех платформах, является `dumb`. Подмодуль `dumb` пакета `dbm` предлагает минимальную функциональность DBM и среднюю производительность. Единственное преимущество `dumb` заключается в том, что его можно использовать где угодно, поскольку `dumb` не зависит ни от каких библиотек. Обычно вы не должны импортировать модуль `dbm.dumb`: вместо этого следует импортировать пакет `dbm` и позволить функции `dbm.open` предоставить наилучший из доступных модулей DBM, выбирая по умолчанию подмодуль `dumb` в отсутствие других доступных вариантов в текущей установке Python. Непосредственный импорт модуля `dumb` может потребоваться лишь в тех редких случаях, когда вам необходимо создать DBM-подобный файл, который будет гарантированно читаться в любой установке Python. Предоставляемая модулем `dumb` функция `open` полиморфна по отношению к одноименной функции пакета `dbm`.

dbm-модули в версии v2

В версии v2 модуль anydbm является обобщенным интерфейсом к любому другому модулю DBM. Модуль anydbm предоставляет единственную функцию-фабрику open, эквивалентную функции dbm.open версии v3.

Модуль whichdb версии v2 предоставляет единственную функцию, whichdb, эквивалентную функции dbm.whichdb версии v3.

Модули, реализующие DBM-функциональность в версии v2, — это высокоуровневые модули с именами dbm, gdbm, dumbdbm, которые во всех остальных отношениях аналогичны соответствующим подмодулям пакета dbm в версии v3.

Примеры использования DBM-подобных файлов

Предлагаемый файлами DBM доступ по ключу удобен в тех случаях, когда программе нужно записывать для постоянного хранения объекты, эквивалентные словарям Python, в которых как ключи, так и значения являются строками. Предположим, вам нужно прочитать несколько текстовых файлов, имена которых заданы в качестве аргументов вашей программы, и записать, где именно в файлах встречаются различные слова. В данном случае ключи — это слова, которые по самой своей сути являются строками. Роль данных, которые требуется записать для каждого слова, играет список пар (*имя_файла*, *номер_строки*). Однако вы можете кодировать данные в виде строк несколькими способами — например, используя тот факт, что строка разделителя пути к файлу os.pathsep (см. раздел "Атрибуты модуля os, связанные с путем доступа" в главе 10) обычно не встречается в именах файлов. (Более продуманные, общие и надежные подходы к кодированию данных в виде строк обсуждаются при рассмотрении аналогичного примера в разделе "Сериализация".) С учетом этого упрощения программа, записывающая позиции слов в файлах, могла бы иметь следующий вид в версии v3.

```
import collections, fileinput, os, dbm
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = '{}{}{}'.format(
        fileinput.filename(), os.pathsep, fileinput.filelineno())
    for word in line.split():
        word_pos[word].append(pos)
sep2 = os.pathsep * 2
with dbm.open('indexfile','n') as dbm_out:
    for word in word_pos:
        dbm_out[word] = sep2.join(word_pos[word])
```

Данные, сохраненные в DBM-подобном файле *indexfile*, можно прочитать несколькими способами. В приведенном ниже примере для версии v3 программа принимает слова в качестве аргументов командной строки и выводит строки, в которых встречаются запрошенные слова.

```
import sys, os, dbm, linecache
dbm_in = dbm.open('indexfile')
sep = os.pathsep
sep2 = sep * 2
for word in sys.argv[1:]:
    if word not in dbm_in:
        print('Word {!r} not found in index file'.format(word),
              file=sys.stderr)
        continue
    places = dbm_in[word].split(sep2)
    for place in places:
        fname, lineno = place.split(sep)
        print('Word {!r} occurs in line {} of file {}:'.format(
              word,lineno,fname))
        print(linecache.getline(fname, lineno), end='')
```

В версии v2 в обоих примерах следует импортировать и использовать не пакет dbm, а модуль anydbm.

Интерфейс к библиотеке Berkeley DB

Версия v2 поставляется с пакетом bsddb, который играет роль оболочки вокруг библиотеки Berkeley Database (также известна как BSD DB), если эта библиотека установлена в системе и установленная сборка Python поддерживает ее. Однако пакет bsddb признан устаревшим в версии v2 и отсутствует в версии v3, поэтому мы не рекомендуем его применять. Если вам все же понадобится интерфейс для доступа к архивным файлам BSD DB, то для этого лучше воспользоваться великолепным сторонним пакетом bsddb3.

Python Database API (DBAPI) 2.0

Как уже отмечалось, интерфейс к реляционной СУБД не поставляется вместе со стандартной библиотекой Python (за исключением рассмотренного в разделе "SQLite" пакета sqlite3, который является не просто интерфейсом, а многофункциональной реализацией). Многие сторонние модули позволяют программам на Python получать доступ к специфическим базам данных. Подобные модули в основном следуют стандарту Python Database API 2.0, также известному как DBAPI, в соответствии с документом **PEP 249** (https://www.python.org/dev/peps/pep-0249/).

Импортировав любой DBAPI-совместимый модуль, следует вызвать функцию connect модуля со специфическими для БД параметрами. Функция connect возвращает *x* — объект Connection, представляющий соединение с БД. Объект *x* содержит методы commit и rollback для работы с транзакциями, метод close, который должен вызываться сразу же после завершения работы с БД, и метод cursor, возвращающий объект *c* класса Cursor. Объект *c* предоставляет методы и атрибуты,

используемые для работы с БД. DBAPI-совместимый модуль также содержит классы исключений, описательные атрибуты, функции-фабрики и атрибуты описания типа.

Классы исключений

DBAPI-совместимый модуль предоставляет классы исключений `Warning`, `Error` и несколько подклассов `Error`. Исключение `Warning` указывает на возникновение таких аномалий, как усечение данных при вставке. Подклассы `Error` указывают на различные виды ошибок, с которыми может столкнуться ваша программа при работе с БД и обеспечивающим интерфейс для доступа к ней DBAPI-совместимым модулем. В общем случае для перехвата всех связанных с БД ошибок в коде используют инструкцию следующего вида.

```
try:
    ...
except module.Error as err:
    ...
```

Потоковая безопасность

Если у DBAPI-совместимого модуля имеется атрибут `threadsafety`, значение которого больше `0`, то тем самым модуль декларирует определенный уровень поддержки потоковой безопасности интерфейса к БД. Вместо того чтобы полагаться на эту поддержку, обычно безопаснее и всегда надежнее в плане переносимости организовать работу программы таким образом, чтобы в любой момент времени только один поток имел исключительный доступ к любому внешнему ресурсу, такому как БД (раздел "Архитектура многопоточной программы" в главе 14).

Формат параметров

DBAPI-совместимый модуль имеет атрибут `paramstyle`, определяющий формат маркеров, которые играют роль заполнителей, замещаемых фактическими значениями параметров. Эти маркеры вставляются в строки SQL-инструкций, передаваемых методам экземпляров `Cursor`, таким как `execute`, для использования значений параметров, определяемых во время выполнения. Допустим, вам нужно извлечь из БД строки таблицы `ATABLE`, в которых поле `AFIELD` равно текущему значению переменной *x* Python. Пусть *c* — экземпляр курсора, тогда теоретически (хотя поступать так не рекомендуется) эту задачу *можно было бы* решить с помощью следующей строки форматирования Python:

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD={!r}'.format(x))
```

Избегайте форматирования строк SQL-запросов: используйте подстановку значений параметров

Прибегать к форматированию строк запросов *не* рекомендуется. Это приводит к генерированию различных строк для каждого значения *x*,

что требует всякий раз выполнять синтаксический анализ инструкции и заново подготавливать запрос. Кроме того, это создает уязвимости к *атакам путем внедрения SQL-кода* (https://ru.wikipedia.org/wiki/Внедрение_SQL-кода). В случае же подстановки значений функции execute передается единственная строка инструкций с полями-заместителями для подстановки значений параметров. При таком подходе повышается не только производительность, поскольку функции execute достаточно проанализировать и подготовить инструкцию только один раз, но и, что более важно, надежность и безопасность программы.

Например, если атрибут paramstyle (описан ниже) определен в модуле как 'qmark', то предыдущий запрос следует записать в такой форме:

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=?', (some_value,))
```

Доступный только для чтения строковый атрибут paramstyle сообщает вашей программе, каким образом следует использовать подстановку фактических значений параметров при работе с данным модулем. Возможные значения атрибута paramstyle приведены ниже.

format
: Маркер подстановки имеет вид %s, как в строках форматирования старого стиля. Всегда используйте только символ s и не используйте другие указатели типа, каким бы ни был тип данных. Запрос выглядит так:

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=%s', (some_value,))
```

named
: Маркер подстановки имеет вид :*имя*, а параметры именуются. Запрос выглядит так:

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=:x',
          {'x':some_value})
```

numeric
: Маркер подстановки имеет вид :*n*, где *n* — номер параметра, принимающий значения от 1 и выше. Запрос выглядит так:

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=:1', (some_value,))
```

pyformat
: Маркер подстановки имеет вид %(*имя*)s, а параметры именуются. Всегда используйте только символ s и не используйте другие указатели типа, каким бы ни был тип данных. Запрос выглядит так:

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=%(x)s',
          {'x':some_value})
```

qmark
Маркером подстановки является вопросительный знак (?). Запрос выглядит так:

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=?', (x,))
```

В случае именованных параметров (т.е. если `paramstyle` имеет значение `'pyformat'` или `'named'`) вторым аргументом метода `execute` является отображение. В противном случае вторым аргументом является последовательность.

Маркеры подстановки `format` и `pyformat` допускают использование только символа `s` в качестве указателя типа

Единственным допустимым указателем типа для маркеров `format` и `pyformat` является `s`. В обоих случаях никакой другой указатель типа не принимается: например, не следует использовать маркеры `%d` или `%(`*имя*`)d`. Во всех подстановках используйте только маркеры `%s` или `%(`*имя*`)s`, независимо от типа данных.

Функции-фабрики

Как правило, такие значения параметров, как числа (целые или с плавающей точкой) и строки (байтовые или Unicode), которые соответствуют встроенным данным Python, корректно отображаются на эквивалентные типы, используемые в БД. Значению `None` в Python соответствует значение `NULL` в SQL. Однако универсальных типов для представления дат, времени и больших двоичных объектов (BLOB) не существует. DBAPI-совместимый модуль предоставляет функции-фабрики для создания подобных объектов. В большинстве DBAPI-совместимых модулей с этой целью используются типы, предлагаемые модулями `datetime` или `mxDateTime` (глава 12), а также строки или буферные типы для представления объектов BLOB. В DBAPI определены следующие функции-фабрики.

`Binary` `Binary(`*string*`)`

Возвращает объект, представляющий указанную байтовую строку в виде BLOB.

`Date` `Date(`*year*`, `*month*`, `*day*`)`

Возвращает объект, представляющий указанную дату.

`DateFromTicks` `DateFromTicks(`*s*`)`

Возвращает объект, который представляет дату на основе количества секунд *s* с начала эпохи Unix, определяемого с помощью модуля `time` (глава 12). Например, `DateFromTicks(time.time())` означает "сегодня".

Time	`Time(`*hour*`, `*minute*`, `*second*`)` Возвращает объект, представляющий указанное время.
TimeFromTicks	`TimeFromTicks(`*s*`)` Возвращает объект, который представляет время на основе количества секунд *s* с начала эпохи Unix, определяемого с помощью модуля `time` (глава 12). Например, `TimeFromTicks(time.time())` означает "текущее время суток".
Timestamp	`Timestamp(`*year*`, `*month*`, `*day*`, `*hour*`, `*minute*`, `*second*`)` Возвращает объект, представляющий указанные дату и время.
TimestampFromTicks	`TimestampFromTicks(`*s*`)` Возвращает объект, который представляет отметку времени на основе количества секунд *s* с начала эпохи Unix, определяемого с помощью модуля `time` (глава 12). Например, `TimestampFromTicks(time.time())` является отметкой текущей даты и времени.

Атрибуты описания типа

Атрибут `description` экземпляра `Cursor` описывает типы и другие характеристики каждого столбца последнего запроса `SELECT`, выполненного с использованием данного курсора. Тип каждого столбца (второй элемент кортежа, описывающего столбец) определяется одним из следующих атрибутов DBAPI-совместимого модуля.

`BINARY`	Описывает столбцы, содержащие объекты BLOB
`DATETIME`	Описывает столбцы, содержащие значения даты, времени или и то и другое
`NUMBER`	Описывает столбцы, содержащие числа любого рода
`ROWID`	Описывает столбцы, содержащие числовой идентификатор строки
`STRING`	Описывает столбцы, содержащие текст любого рода

Наличие описания курсора и, в частности, типа каждого столбца особенно полезно для интроспекции БД, с которой работает программа. Подобная интроспекция может пригодиться при написании общих модулей, а также при работе с таблицами, использующими различные схемы, в том числе схемы, которые вам могут быть неизвестны во время написания кода.

Функция connect

Функция `connect` DBAPI-совместимого модуля имеет аргументы, которые зависят от используемых БД и конкретного модуля. Стандарт DBAPI рекомендует, чтобы функция `connect` поддерживала именованные аргументы. В частности, функция `connect` должна иметь по крайней мере необязательные элементы с именами, приведенными ниже.

`database`	Имя конкретной базы данных, к которой необходимо подключиться
`dsn`	Имя источника данных для использования с соединением
`host`	Имя хоста, на котором выполняется база данных
`password`	Пароль для использования данного соединения
`user`	Имя пользователя для работы с соединением

Объекты соединения

Функция `connect` DBAPI-совместимого модуля возвращает объект *x*, являющийся экземпляром класса `Connection`. Объект *x* предоставляет следующие методы.

`close` *x*`.close()`

Закрывает соединение с БД и освобождает соответствующие ресурсы. Метод `close` должен вызываться сразу же по завершении работы с БД. Оставленное открытым без особых на то причин соединение с БД может привести к серьезной утечке системных ресурсов.

`commit` *x*`.commit()`

Подтверждает завершение текущей транзакции в БД. Если БД не поддерживает транзакции, вызов *x*`.commit()` не выполняет никаких действий.

`cursor` *x*`.cursor()`

Возвращает новый экземпляр класса `Cursor` (см. раздел "Объекты курсора").

`rollback` `x.rollback()`

Откатывает все текущие транзакции в БД. Если БД не поддерживает транзакций, то вызов *x*`.rollback()` возбуждает исключение. В соответствии с рекомендациями DBAPI в случае БД, которые не поддерживают транзакций, класс `Connection` не должен предоставлять метод `rollback`, чтобы вызов *x*`.rollback()` приводил к возбуждению исключения `AttributeError`. Вы сможете проверить, поддерживаются ли транзакции, с помощью вызова `hasattr(`*x*`,'rollback')`.

Объекты курсора

Экземпляр `Connection` содержит метод `cursor`, который возвращает объект *c*, являющийся экземпляром класса `Cursor`. Курсор SQL представляет набор результатов запроса и позволяет работать с записями этого набора последовательно, по одной за раз. Курсор, моделируемый DBAPI, представляет собой более широкое понятие, поскольку это единственный способ, с помощью которого ваша программа может выполнять SQL-запросы. С другой стороны, курсор DBAPI допускает продвижение по результатам лишь в прямом направлении (некоторые, но не все реляционные базы данных допускают возможность перемещения также в обратном направлении, а не только в прямом) и не поддерживает SQL-предложение `WHERE CURRENT OF CURSOR`. Благодаря этим ограничениям курсоры DBAPI-совместимых модулей могут

легко предоставлять DBAPI-курсоры СУРБД, которые вообще не предоставляют никаких реальных SQL-курсоров. Экземпляр *c* класса `Cursor` содержит большое количество атрибутов и методов. Наиболее часто используемые из них описаны ниже.

close `c.close()`

Закрывает курсор и освобождает соответствующие ресурсы.

description Доступный только для чтения атрибут, представляющий последовательность кортежей из семи элементов с информацией о каждом столбце в последнем выполненном запросе:

```
name, typecode, displaysize, internalsize, precision, scale,
nullable
```

Атрибут `c.description` равен `None`, если последней выполненной с использованием объекта *c* операцией не был запрос `SELECT` или он не вернул никакого полезного описания интересующих вас столбцов. Этот атрибут наиболее полезен для интроспекции БД, с которой работает ваша программа. Подобная интроспекция может пригодиться при написании общих модулей, которые способны работать с таблицами, использующими различные схемы, в том числе схемы, которые вам могут быть неизвестны во время написания кода.

execute `c.execute(`*statement*`, `*parameters*`=None)`

Выполняет инструкцию SQL *statement* по отношению к БД, используя параметры, заданные аргументом *parameters*. Аргумент *parameters* представляет собой последовательность, если атрибут `paramstyle` модуля имеет значение `'format'`, `'numeric'` или `'qmark'`, либо отображение, если атрибут `paramstyle` имеет значение `'named'` или `'pyformat'`. Некоторые модули DBAPI требуют, чтобы последовательностями были конкретно кортежи.

executemany `c.executemany(`*statement*`, *`*parameters*`)`

Выполняет инструкцию SQL *statement* по отношению к БД для каждого из элементов, заданных аргументом *parameters*. Аргумент *parameters* представляет собой последовательность последовательностей, если атрибут `paramstyle` модуля имеет значение `'format'`, `'numeric'` или `'qmark'`, либо последовательность отображений, если атрибут `paramstyle` имеет значение `'named'` или `'pyformat'`. Например, если `paramstyle` равен `'qmark'`, то инструкция

```
c.executemany('UPDATE atable SET x=? '
              'WHERE y=?',(12,23),(23,34))
```

эквивалентна, но работает быстрее, следующим двум инструкциям:

```
c.execute('UPDATE atable SET x=12 WHERE y=23')
c.execute('UPDATE atable SET x=23 WHERE y=34')
```

fetchall `c.fetchall()`

Возвращает все оставшиеся строки результата последнего запроса в виде последовательности кортежей. Если последней операцией не был запрос `SELECT`, возвращает исключение.

fetchmany `c.fetchmany(n)`

Возвращает вплоть до *n* оставшихся строк результата последнего запроса в виде последовательности кортежей. Если последней операцией не был запрос SELECT, возвращает исключение.

fetchone `c.fetchone()`

Возвращает следующую строку результата последнего запроса в виде кортежа. Если последней операцией не был запрос SELECT, возвращает исключение.

rowcount Доступный только для чтения атрибут, представляющий количество строк, которые были извлечены в результате выполнения последней операции или на которые эта операция оказала воздействие, либо `-1`, если модуль не в состоянии определить это значение.

DBAPI-совместимые модули

Какую бы реляционную БД вы ни планировали использовать, существует по крайней мере один (зачастую более одного) DBAPI-совместимый модуль, доступный для загрузки из Интернета. Существует так много БД и модулей, и набор их возможностей настолько подвержен изменениям, что невозможно перечислить их всех и (что также важно) постоянно актуализировать их список. Вместо этого мы рекомендуем начать с просмотра вики-страницы (`https://wiki.python.org/moin/DatabaseInterfaces`), поддерживаемой сообществом, что вселяет надежду на то, что приведенный в ней список является полным и своевременно обновляется.

Как следствие, ниже приведен лишь очень короткий и меняющийся со временем список немногих DBAPI-совместимых модулей, которые на момент выхода книги были популярны сами по себе и предоставляли интерфейс к очень популярным БД с открытым исходным кодом.

ODBC

Open DataBase Connectivity (ODBC), или *открытый механизм взаимодействия с базами данных*, — это стандартный способ подключения к самым различным БД, в том числе и к некоторым из тех, которые не поддерживаются другими DBAPI-совместимыми модулями. Для ODBC- и DBAPI-совместимых модулей с либеральной лицензией на ПО с открытым исходным кодом следует использовать пакет `pyodbc` (`https://pypi.python.org/pypi/pyodbc/`). Для модулей, поддерживаемых на коммерческой основе, следует использовать интерфейс `mxODBC` (`http://www.egenix.com/products/python/mxODBC/`).

MySQL

MySQL — это популярная реляционная СУБД с открытым исходным кодом, владельцем которой в настоящее время является компания Oracle. Принадлежащим Oracle "официальным" DBAPI-совместимым интерфейсом к ней является MySQL Connector/Python (`https://pypi.python.org/pypi/mysql-connector-python/2.0.4`).

PostgreSQL

PostgreSQL — отличная реляционная СУБД с открытым исходным кодом. Наиболее популярным DBAPI-совместимым интерфейсом к ней является `psycopg2` (`https://pypi.python.org/pypi/psycopg2`).

SQLite

SQLite (`www.sqlite.org`) — автономная, не требующая подключения к серверу, нуждающаяся лишь в минимальной настройке и наиболее широко используемая во всем мире транзакционная реляционная СУБД. Она представляет собой библиотеку, написанную на языке C, которая реализует базу данных в одном файле или даже в памяти в случае небольших или промежуточных задач. Стандартная библиотека Python включает пакет `sqlite3`, реализующий DBAPI-совместимый интерфейс к SQLite.

SQLite обладает богатой функциональностью со множеством доступных для выбора опций. Пакет `sqlite3` предлагает доступ к значительной части этой функциональности, а также дополнительные возможности, благодаря которым взаимодействие вашего кода на языке Python с базой данных становится еще более удобным и естественным. В книге мы не будем подробно рассматривать все тонкости работы с этими двумя мощными программными системами; наше внимание будет сосредоточено на подмножестве наиболее широко используемых и наиболее полезных возможностей. Если вам требуется более глубокий уровень детализации, включая примеры и рекомендации, касающиеся наилучших подходов, обратитесь к документации SQLite (`https://www.sqlite.org/docs.html`) и онлайн-документации `sqlite3` (`https://docs.python.org/3/library/sqlite3.html`).

Пакет `sqlite3` содержит следующие функции.

connect

`connect(`*`filepath`*`, `*`timeout`*`=5.0, `*`detect_types`*`=0, `*`isolation_level`*`='', `*`check_same_thread`*`=True, `*`factory`*`=Connection, `*`cached_statements`*`=100, `*`uri`*`=False)`

Функция `connect` создает соединение с базой данных SQLite с именем файла *`filepath`* (создавая его, если это необходимо) и возвращает экземпляр класса `Connection` (или его подкласса, переданного с помощью аргумента *`factory`*). Чтобы создать базу данных в памяти, следует передать строку `':memory:'` в качестве первого аргумента, *`filepath`*.

Аргумент *`uri`* можно передавать только в версии v3. Если он имеет значение *`True`*, то это активизирует функциональность URI (`http://www.sqlite.org/uri.html`) в SQLite, разрешающую передавать дополнительные опции наряду с путем к файлу посредством аргумента *`filepath`*.

Аргумент *timeout* задает длительность тайм-аута в секундах, который должен быть выдержан, прежде чем будет возбуждено исключение, если другое соединение блокирует базу данных в транзакции.

Модуль sqlite3 непосредственно поддерживает лишь собственные типы SQLite: BLOB, INTEGER, NULL, REAL и TEXT (любое другое имя типа трактуется как TEXT, если оно не распознано соответствующим образом и не передано через конвертер, зарегистрированный с помощью функции register_converter, которая описана далее), преобразуя их следующим образом.

Тип SQLite	Тип Python
BLOB	buffer в версии v2, bytes в версии v3
INTEGER	int (или, в версии v2, long для очень больших чисел)
NULL	None
REAL	float
TEXT	Зависит от атрибута text_factory экземпляра Connection, описанного далее; по умолчанию: unicode в версии v2, str в версии v3

Чтобы разрешить обнаружение имен типов, следует передать в качестве аргумента *detect_types* одну из констант, PARSE_COLNAMES или PARSE_DECLTYPES, предоставляемых пакетом sqlite3 (или обе константы, объединив их оператором | побитового ИЛИ).

В случае передачи аргумента *detect_types*=sqlite3.PARSE_COLNAMES имя типа берется из имени столбца в инструкции SELECT, которая извлекает данный столбец. Например, столбец, извлеченный как *foo* AS [*foo* CHAR(10)], имеет тип CHAR.

Если передается аргумент *detect_types*=sqlite3.PARSE_DECLTYPES, то имя типа берется из объявления столбца в исходной SQL-инструкции CREATE TABLE или ALTER TABLE, добавившей столбец. Например, столбец, объявленный как *foo* CHAR(10), имеет тип CHAR.

Если передается аргумент *detect_types*=sqlite3.PARSE_COLNAMES|sqlite3.PARSE_DECLTYPES, то используются оба механизма, причем приоритет отдается имени столбца, состоящему по крайней мере из двух слов (в этом случае имя типа задается вторым словом), а если это не так, то в качестве типа используется тот, который был задан для данного столбца в объявлении (в подобном случае имя типа задается первым словом в объявлении).

Аргумент *isolation_level* позволяет осуществлять определенный контроль над тем, как SQLite обрабатывает транзакции. Он может иметь значения '' (значение по умолчанию), None (автоматическая фиксация транзакций) или одну из трех строк: 'DEFERRED',

'EXCLUSIVE' и 'IMMEDIATE'. В онлайн-документации SQLite подробно описаны типы транзакций (https://www.sqlite.org/lang_transaction.html) и их связь с различными уровнями блокирования файлов (https://www.sqlite.org/lockingv3.html), выполняемого внутренними механизмами SQLite.

По умолчанию объект соединения может использоваться только в создавшем его потоке Python, чтобы избежать ситуаций, которые могут легко привести к повреждению базы данных вследствие незначительных ошибок, допущенных в программе (незначительные ошибки довольно часто встречаются в многопоточном программировании). Если вы полностью уверены в правильности использования блокировок и других механизмов синхронизации и вам необходимо повторно использовать объект соединения со многими потоками, можете передать аргумент *check_same_thread*=False: в этом случае sqlite3 не будет выполнять проверку, доверяя вашему утверждению о том, что вы знаете, что делаете, и что ваша многопоточная архитектура на все 100% свободна от ошибок. Остается лишь пожелать вам удачи!

Аргумент *cached_statements* — это количество SQL-инструкций, которые sqlite3 кеширует в проанализированном и подготовленном состоянии, чтобы избежать лишних накладных расходов, обусловленных повторным анализом инструкций. Вы можете передать меньшее значение, чем заданное по умолчанию значение 100, чтобы сэкономить немного памяти, или большее, если ваше приложение использует необычайно разнообразный набор SQL-инструкций.

register_adapter register_adapter(*type*, *callable*)

Функция register_adapter регистрирует вызываемый объект *callable* в качестве адаптера для преобразования любого объекта Python, имеющего тип *type*, в соответствующее значение одного из нескольких типов Python, которые sqlite3 обрабатывает непосредственно: int, float, str (в версии v3 также bytes; в версии v2 также buffer, long, unicode; кроме того, имейте в виду, что в версии v2 тип str должен преобразовываться в кодировку 'utf-8'). Вызываемый объект должен принимать преобразуемое значение в качестве единственного аргумента и возвращать тип, который sqlite3 обрабатывает непосредственно.

register_converter register_converter(*typename*, *callable*)

Функция register_converter регистрирует вызываемый объект *callable* в качестве конвертера, преобразующего любое значение, идентифицируемое в SQL как тип *typename* (см. описание параметра *detect_types* функции connect, где объясняется, как идентифицируется имя типа), в соответствующий объект Python. Объект *callable* должен принимать в качестве единственного аргумента строковую форму значения, полученного от SQL, и возвращать соответствующий объект Python. Сопоставление с *typename* чувствительно к регистру.

Кроме того, sqlite3 содержит классы Connection, Cursor и Row. На базе каждого из них могут создаваться подклассы; однако это более сложная тема, которая в данной книге не рассматривается. Класс Cursor — это стандартный класс курсора DBAPI, за исключением того, что он включает дополнительный вспомогательный метод executescript, имеющий единственный аргумент в виде строки, которая состоит из нескольких инструкций, разделенных символом ; (без параметров). Два других класса рассмотрены в следующих разделах.

Класс sqlite3.Connection

В дополнение к методам, общим для всех классов Connection DBAPI-совместимых модулей (см. раздел "Объекты соединения"), класс sqlite3.Connection предоставляет следующие методы и другие атрибуты.

create_aggregate

`create_aggregate(name, num_params, aggregate_class)`

Аргумент *aggregate_class* должен быть классом, предоставляющим два метода экземпляра: step, имеющий ровно *num_params* аргументов, и finalize, не имеющий аргументов и возвращающий окончательный результат агрегата — значение типа, поддерживаемого самим модулем sqlite3. Функция агрегата может использоваться в SQL-инструкциях по ее имени, заданному в виде строки *name*.

create_collation

`create_collation(name, callable)`

Вызываемый объект *callable* должен принимать две байтовые строки в качестве аргументов (в кодировке 'utf-8') и возвращать -1, если первая строка должна считаться "меньшей, чем" вторая, 1, если она должна считаться "большей, чем" вторая, и 0, если обе строки должны считаться "равными" для целей данного сравнения. Подобные сопоставления можно именовать по аргументу *name* в предложении ORDER BY инструкции SELECT.

create_function

`create_function(name, num_params, func)`

Функция *func* должна иметь ровно *num_params* аргументов и возвращать значение типа, поддерживаемого самим модулем sqlite3. Подобную пользовательскую функцию можно использовать в SQL-инструкциях по ее имени, заданному в виде строки *name*.

interrupt

`interrupt()`

Вызов данной функции из любого потока прерывает выполнение всех запросов в этом соединении (возбуждая исключение в потоке, использующем данное соединение).

isolation_level

Доступный только для чтения атрибут, который является значением, заданным в виде параметра *isolation_level* функции connect.

iterdump	`iterdump()` Возвращает итератор, который выводит строки SQL-инструкций, создающих текущую БД с нуля, включая схему и содержимое. Эту функцию удобно использовать, например, для сохранения БД, временно хранящейся в памяти, на одном из дисков для последующего повторного использования в будущем.
row_factory	Вызываемый объект, который принимает объект курсора и исходную строку в виде кортежа и возвращает объект для использования в качестве строки реального результата. Распространенной идиомой является `x.row_factory=sqlite3.Row`, использующая в высшей степени оптимизированный класс `Row` (раздел "Класс `sqlite3.Row`") и предоставляющая доступ к столбцам как по индексам, так и по именам столбцов без учета регистра, при пренебрежимо малых накладных расходах.
text_factory	Вызываемый объект, принимающий единственный параметр в виде байтовой строки и возвращающий объект, который будет использоваться для значения данного столбца типа `TEXT`. По умолчанию: `str` в v3, `unicode` в v2, но вы можете задать любой аналогичный вызываемый объект.
total_changes	Общее количество строк, измененных, вставленных или удаленных с момента создания соединения.

Класс `sqlite3.Row`

Пакет `sqlite3` содержит класс `Row`, который в основном ведет себя как тип `tuple`, но также предоставляет метод `keys()`, возвращающий список имен столбцов, и поддерживает индексирование по имени столбца в качестве дополнительной альтернативы индексированию по номеру столбца.

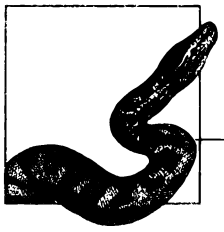
Пример использования `sqlite3`

В приведенном ниже примере решается та же задача, что и в показанных ранее примерах, но в нем персистентность обеспечивается с помощью `sqlite3`, а не путем индексирования объектов в памяти.

```
import fileinput, sqlite3
connect = sqlite3.connect('database.db')
cursor = connect.cursor()
    cursor.execute('CREATE TABLE IF NOT EXISTS Words '
                   '(Word TEXT, File TEXT, Line INT)')
for line in fileinput.input():
    f, l = fileinput.filename(), fileinput.filelineno()
    cursor.executemany('INSERT INTO Words VALUES (:w, :f, :l)',
        [{'w':w, 'f':f, 'l':l} for w in line.split()])
connect.commit()
connect.close()
```

После этого мы можем использовать sqlite3 для чтения данных, сохраненных в файле database.db, как показано в следующем примере.

```
import sys, sqlite3, linecache
connect = sqlite3.connect('database.db')
cursor = connect.cursor()
for word in sys.argv[1:]:
    cursor.execute('SELECT File, Line FROM Words '
                   'WHERE Word=?', [word])
    places = cursor.fetchall()
    if not places:
        print('Word {!r} not found in index file'.format(word),
              file=sys.stderr)
        continue
    for fname, lineno in places:
        print('Word {!r} occurs in line {} of file {}:'.format(
               word,lineno,fname))
        print(linecache.getline(fname, lineno), end='')
```



12

Работа со значениями даты и времени

Программа на языке Python может обрабатывать значения времени разными способами. Временны*е* интервалы измеряются в секундах и выражаются числами с плавающей точкой (дробной части соответствует доля секунды): все функции стандартной библиотеки, которые имеют аргумент, задающий интервал времени в секундах, принимают его в виде значения с плавающей точкой. Обычно на платформах Unix и Windows началом отсчета времени служит полночь (по UTC) с 31 декабря 1969 года на 1 января 1970 года (так называемая *эпоха Unix*). Время часто выражается с применением смешанных единиц (например, годы, месяцы, дни, часы, минуты и секунды), особенно это касается ввода и вывода значений. Разумеется, вводимые и выводимые значения даты и времени часто подвергаются форматированию для представления их в виде удобочитаемых строк и обратного преобразования из строковых форматов.

В этой главе рассматривается модуль `time`, в котором сосредоточена основная функциональность Python для работы со значениями времени. Модуль `time` в определенной мере зависит от библиотеки C базовой системы. Также будут рассмотрены модули `datetime`, `sched` и `calendar` стандартной библиотеки Python и сторонние модули `dateutil` и `pytz`.

Модуль `time`

Базовая библиотека C определяет диапазон дат, которые может обрабатывать модуль `time`. В Unix-системах годы 1970 и 2038 являются границами допустимого диапазона — ограничение, которое преодолевается в модуле `datetime`. Обычно моменты времени указываются в соответствии со стандартом UTC (Coordinated Universal Time — всемирное координированное время), ранее известным как GMT (Greenwich Mean Time — среднее время по Гринвичу). Модуль `time` также поддерживает часовые пояса и летнее время (daylight saving time — DST), но только в той мере, в какой это позволяет базовая библиотека C.

Альтернативой представлению моментов времени количеством секунд, истекших с начала эпохи Unix, может служить их представление в виде кортежей, состоящих из девяти целых чисел, получившее название *временной кортеж* (табл. 12.1). Все элементы являются целыми числами: временные кортежи не отслеживают доли секунды. Временной кортеж является экземпляром `struct_time`. Он может использоваться как кортеж, но к его элементам также можно обращаться как к доступным только для чтения атрибутам `x.tm_year`, `x.tm_mon` и т.п., имена которых приведены в табл. 12.1. Везде, где функции требуется аргумент в виде временного кортежа, ей можно передать экземпляр `struct_time` или любую другую последовательность, элементами которых являются девять целых чисел, принадлежащих к корректным диапазонам (указанные в таблице диапазоны включают свои граничные значения).

Таблица 12.1. Представление времени в форме кортежа

Элемент	Значение	Имя поля	Диапазон	Примечания
0	Год	`tm_year`	1970–2038	Расширен на некоторых платформах
1	Месяц	`tm_mon`	1–12	1 — январь, 12 — декабрь
2	День	`tm_mday`	1–31	
3	Час	`tm_hour`	0–23	0 — полночь, 12 — полдень
4	Минута	`tm_min`	0–59	
5	Секунда	`tm_sec`	0–61	60 и 61 для секунд координации
6	День недели	`tm_wday`	0–6	0 — понедельник, 6 — воскресенье
7	День года	`tm_yday`	1–366	Порядковый номер дня в году
8	Флаг DST	`tm_isdst`	от –1 до 1	–1 означает, что флаг DST (летнее время) определяется библиотекой

Чтобы преобразовать время из представления в виде числа с плавающей точкой, выражающего количество секунд, истекших с момента начала эпохи Unix, во временной кортеж, следует передать это число функции (например, `localtime`), возвращающей временной кортеж с девятью действительными элементами. При преобразовании в другом направлении функция `mktime` игнорирует избыточные шестой (`tm_wday`) и седьмой (`tm_yday`) элементы кортежа. В таком случае восьмой элемент (`tm_isdst`) задается равным `-1`, поэтому функция `mktime` сама определяет, следует ли применять флаг DST.

Модуль `time` предоставляет функции и атрибуты, приведенные в табл. 12.2.

Таблица 12.2. Функции и атрибуты модуля `time`

Функция	Описание
`asctime`	`asctime([`*`tupletime`*`])` Принимает временной кортеж и возвращает 24-символьную строку в удобочитаемом виде, например `'Sun Jan 8 14:41:06 2017'`.

Функция	Описание
	Вызов функции `asctime()` без аргументов аналогичен вызову `asctime(localtime(time()))` (приводит значение текущего времени к локальному формату времени)
clock	`clock()` Возвращает текущее время CPU в секундах в виде числа с плавающей точкой, зависящего от платформы. Функция `clock()` признана устаревшей в версии v3. Чтобы измерить время, необходимое для выполнения вычислений, следует использовать вместо нее модуль стандартной библиотеки `timeit` (раздел "Модуль `timeit`" в главе 16). В версии v3 для реализации тайм-аутов или планирования событий следует использовать функцию `perf_counter()` или `process_time()`. Для получения информации относительно потокобезопасного планирования событий обратитесь к описанию модуля `sched`
ctime	`ctime([`*secs*`])` Как и функция `asctime(localtime(`*secs*`))`, принимает время *secs*, выраженное в секундах от начала эпохи Unix, и возвращает соответствующую 24-символьную строку в локальном формате времени. Вызов функции `ctime()` без аргументов аналогичен вызову функции `asctime()` (приводит значение текущего времени к локальному формату времени)
gmtime	`gmtime([`*secs*`])` Принимает время *secs*, выраженное в секундах от начала эпохи Unix, и возвращает временной кортеж *t*, соответствующий времени по UTC (`t.tm_isdst` всегда равно `0`). Вызов функции `gmtime()` без аргументов подобен вызову `gmtime(time())` (возвращает временной кортеж, соответствующий текущему времени)
localtime	`localtime([`*secs*`])` Принимает время *secs*, выраженное в секундах от начала эпохи Unix, и возвращает временной кортеж *t*, соответствующий локальному часовому поясу (`t.tm_isdst` равно `0` или `1`, в зависимости от того, применяется ли флаг летнего времени DST в соответствии с локальными правилами). Вызов функции `localtime()` без аргументов аналогичен вызову `localtime(time())` (возвращает временной кортеж, соответствующий локальному часовому поясу)
mktime	`mktime(`*tupletime*`)` Принимает временной кортеж, соответствующий времени в локальном часовом поясе, и возвращает число с плавающей точкой, представляющее время в секундах от начала эпохи Unix*. Флаг летнего времени DST, последний элемент временного кортежа, функционирует: установите его в `0` для получения местного солнечного времени, в `1` для получения летнего времени или в `-1` для того, чтобы функция `mktime` вычислила, действует ли летнее время в данный момент

Функция	Описание
monotonic	`monotonic()` Только в версии v3. Как и функция `time()`, возвращает текущее время в секундах с момента начала эпохи Unix в виде числа с плавающей точкой. Гарантируется, что эти часы никогда не вернутся назад в промежутках между вызовами, даже если системные часы были обновлены (например, для учета високосных секунд)
perf_counter	`perf_counter()` Только в версии v3. Возвращает значение в дробных секундах, используя доступные часы высокого разрешения для точных измерений коротких длительностей. Действует в пределах всей системы и *учитывает* время, истекшее на протяжении периодов сна. Используйте только разницу между последовательными вызовами, поскольку начальная точка отсчета времени для этой функции не определена
process_time	`process_time()` Только в версии v3. Возвращает значение в дробных секундах, используя доступные часы высокого разрешения для точных измерений коротких длительностей. Действует в пределах всей системы и *не учитывает* время, истекшее на протяжении периодов сна. Используйте только разницу между последовательными вызовами, поскольку начальная точка отсчета времени для этой функции не определена
sleep	`sleep(`*`secs`*`)` Приостанавливает работу текущего потока на *`secs`* секунд. Работа текущего потока может быть возобновлена до истечения *`secs`* секунд (если это основной поток, который может быть пробужден каким-либо сигналом) или по истечении более длительной приостановки (в зависимости от того, как организована работа системного планировщика процессов и потоков). Вызов функции `sleep` с аргументом *`secs`*`=0` предоставляет другим потокам возможность выполниться и не сопровождается какой-либо значимой задержкой, если текущий поток является единственным, который готов к выполнению
strftime	`strftime(`*`fmt`*`[, `*`tupletime`*`])` Принимает местное время, представленное временны́м кортежем, и возвращает строку, представляющую время в соответствии со строкой формата *`fmt`*. Если аргумент *`tupletime`* опущен, то функция `strftime` использует функцию `localtime(time())` (форматирует текущее время). Синтаксис строки *`fmt`* аналогичен тому, который был описан в разделе "Традиционное форматирование строк с помощью оператора `%`" главы 8. Ее спецификаторы формата приведены в табл. 12.3. См. спецификацию указания времени кортежем `tupletime`; этот формат не позволяет указывать ширину поля и точность.

Функция	Описание

Таблица 12.3. Спецификаторы формата для `strftime`

Символ	Значение	Примечание
`a`	Сокращенное название дня недели	Зависит от локали
`A`	Полное название дня недели	Зависит от локали
`b`	Сокращенное название месяца	Зависит от локали
`B`	Полное название месяца	Зависит от локали
`c`	Полное представление даты и времени	Зависит от локали
`d`	Число месяца	1–31
`G`	*Новое в версии 3.6:* десятичное представление года в соответствии со стандартом ISO 8601:2000	
`H`	Десятичное представление часа (24-часовая шкала)	0–23
`I`	Десятичное представление часа (12-часовая шкала)	1–12
`j`	Десятичное представление дня года	1–366
`m`	Десятичное представление месяца	1–12
`M`	Десятичное представление минут	0–59
`p`	Национальный эквивалент обозначения A.M. (до полудня) или P.M. (после полудня)	Зависит от локали
`S`	Десятичное представление секунд	0–61
`u`	*Новое в версии 3.6:* день недели	1 — понедельник
`U`	Номер недели в году (первым днем недели считается воскресенье)	0–53
`V`	*Новое в версии 3.6:* номер недели в году в соответствии со стандартом ISO 8601:2000	
`w`	Десятичное представление дня недели	0–6 (0 — воскресенье)
`W`	Номер недели в году (началом недели считается понедельник)	0–53
`x`	Полное представление даты	Зависит от локали
`X`	Полное представление времени	Зависит от локали
`y`	Десятичное представление года без указания века	0–99
`Y`	Десятичное представление года	1970–2038 или шире
`Z`	Название часового пояса	Пусто, если не существует поясного времени
`%`	Литеральный символ %	Кодируется как %%

Функция	Описание

	Можно получить даты, отформатированные в том виде, в каком они форматируются функцией `asctime` (например, `'Tue Dec 10 18:07:14 2002'`), используя следующую строку формата:
	`'%a %b %d %H:%M:%S %Y'`
	Можно получить даты, совместимые с документом RFC 822 (например, `'Tue, 10 Dec 2002 18:07:14 EST'`), используя следующую строку формата:
	`'%a, %d %b %Y %H:%M:%S %Z'`

strptime	`strptime(str, [fmt='%a %b %d %H:%M:%S %Y'])` Анализирует строку *`str`* в соответствии со строкой формата *`fmt`* и возвращает время в представлении временного кортежа. Спецификаторы формата совпадают с теми, которые были приведены при рассмотрении функции `strftime`
time	`time()` Возвращает текущее время в секундах от начала эпохи в виде числа с плавающей точкой. На некоторых (в основном старых) платформах точность этого времени не превышает одной секунды. Может возвращать меньшее значение в последовательности вызовов, если в промежутках между ними системные часы возвращаются назад (например, для учета високосных секунд)
timezone	`timezone` Выраженное в секундах смещение местного часового пояса (без учета летнего времени) от UTC (`>0` в Америке; `<=0` в большей части Европы, Азии и Африки)
tzname	`tzname` Пара зависящих от локали строк, представляющих названия часовых поясов с учетом и без учета перехода на летнее время соответственно

* Дробная часть результата, возвращаемого функцией `mktime`, всегда равна `0`, поскольку ее аргумент *`timetuple`* не учитывает доли секунды.

Модуль `datetime`

Модуль `datetime` содержит классы для моделирования объектов даты и времени, которые могут либо *учитывать*, либо *не учитывать* (режим по умолчанию) часовые пояса. Класс `tzinfo`, экземпляры которого моделируют часовой пояс, — абстрактный: модуль `datetime` не предоставляет его реализацию (со всеми подробностями можно ознакомиться в онлайн-документации; `https://docs.python.org/3/library/datetime.html#tzinfo-objects`). В описании модуля `pytz` (раздел "Модуль

pytz") приведена неплохая простая реализация класса tzinfo, которая позволяет без труда создавать объекты datetime, способные учитывать переход на летнее время. В модуле datetime все типы имеют неизменяемые экземпляры: атрибуты доступны только для чтения, а экземпляры могут использоваться в качестве ключей словарей или элементов множеств.

Класс date

Экземпляры класса date представляют дату (без указания времени в пределах данной даты), никогда не учитывают часовой пояс и работают из предположения, что всегда использовался Григорианский календарь. Экземпляры date имеют три доступных только для чтения целочисленных атрибута: year, month и day.

date date(*year*, *month*, *day*)

Класс date поддерживает следующие методы класса, используемые в качестве альтернативы конструкторам.

fromordinal date.fromordinal(*ordinal*)

Возвращает объект date, который соответствует дате, отстоящей на *ordinal* дней от начала нашей эры согласно пролептическому григорианскому календарю (https://ru.wikipedia.org/wiki/Пролептический_григорианский_календарь), причем значению 1 соответствует первый день года 1 н. э.

fromtimestamp date.fromtimestamp(*timestamp*)

Возвращает объект date, который соответствует отметке времени *timestamp* и выражает время в секундах от начала эпохи Unix.

today date.today()

Возвращает объект date, представляющий текущую дату.

Экземпляры класса date поддерживают некоторые арифметические операции. Разностью двух экземпляров date является экземпляр timedelta. Экземпляры timedelta могут использоваться в операциях сложения и вычитания с объектами date для создания других экземпляров date. Допускается сравнение двух экземпляров класса date (большим является экземпляр, соответствующий более поздней дате).

Экземпляр *d* класса date предоставляет следующие методы.

ctime *d*.ctime()

Возвращает строку, представляющую дату *d* в том же 24-символьном формате, который используется функцией time.ctime (с установкой времени суток на полночь, 00:00:00).

isocalendar *d*.isocalendar()

Возвращает дату, представленную кортежем из трех целых чисел, выражающих год, номер недели в году и день недели в соответствии со стандартом ISO. Для получения более подробной информации относительно календаря ISO (International Standards Organization) следует обратиться к документу ISO 8601 (https://ru.wikipedia.org/wiki/ISO_8601).

isoformat *d*.isoformat()

Возвращает строку, представляющую дату *d* в формате 'YYYY-MM-DD'; то же самое, что str(*d*).

isoweekday *d*.isoweekday()

Возвращает соответствующий объекту *d* день недели в виде целого числа в диапазоне от 1 (понедельник) до 7 (воскресенье); то же самое, что *d*.weekday() + 1.

replace *d*.replace(*year*=None, *month*=None, *day*=None)

Возвращает новый объект date, аналогичный *d*, за исключением атрибутов, явно указанных в качестве аргументов, которые получают новые значения. Например:

```
date(x,y,z).replace(month=m) == date(x,m,z)
```

strftime *d*.strftime(*fmt*)

Возвращает строку, представляющую дату *d* в соответствии со строкой формата *fmt*; аналогичен следующему вызову:

```
time.strftime(fmt, d.timetuple())
```

timetuple *d*.timetuple()

Возвращает временной кортеж, соответствующий времени суток 00:00:00 (полночь) на дату, представленную объектом *d*.

toordinal *d*.toordinal()

Возвращает порядковый номер даты *d* по пролептическому григорианскому календарю. Например:

```
date(1,1,1).toordinal() == 1
```

weekday *d*.weekday()

Возвращает день недели, соответствующий дате *d*, в виде целого числа от 0 (понедельник) до 6 (воскресенье); аналогичен вызову *d*.isoweekday() - 1.

Класс time

Экземпляры класса time представляют время суток (без привязки к какой-либо дате), могут учитывать или не учитывать часовые пояса и всегда игнорируют високосные секунды. Они обладают пятью атрибутами: четыре доступных только для чтения целочисленных атрибута (hour, minute, second и microsecond) и один

необязательный атрибут `tzinfo` (равный `None` в случае экземпляров, не учитывающих часовой пояс).

time `time(`*hour*`=0, `*minute*`=0, `*second*`=0, `*microsecond*`=0, `*tzinfo*`=None)`

Экземпляры класса `time` не поддерживают арифметические операции. Допускается сравнение двух экземпляров класса `time` между собой (большим считается экземпляр, которому соответствует более позднее время суток), но только в том случае, если оба они учитывают или не учитывают часовой пояс.

Экземпляр *t* класса `time` предоставляет следующие методы.

isoformat *t*`.isoformat()`

Возвращает строку, представляющую время *t* в формате `'HH:MM:SS'`; то же самое, что `str(`*t*`)`. Если *t*`.microsecond!=0`, то результирующая строка получается более длинной: `'HH:MM:SS.mmmmmm'`. Если объект *t* учитывает часовой пояс, то в конце строки добавляются шесть дополнительных символов `'+HH:MM'`, представляющих смещение часового пояса относительно UTC. Другими словами, эта операция форматирования следует стандарту ISO 8601.

replace *t*`.replace(`*hour*`=None, `*minute*`=None, `*second*`=None, `*microsecond*`=None[, `*tzinfo*`])`

Возвращает новый объект `time`, аналогичный *t*, за исключением атрибутов, явно указанных в качестве аргументов, которые получают новые значения. Например:

```
time(x,y,z).replace(minute=m) == time(x,m,z)
```

strftime *t*`.strftime(`*fmt*`)`

Возвращает строку, представляющую время *t* в соответствии со строкой формата *fmt*.

Кроме того, экземпляр *t* класса `time` предоставляет методы `dst`, `tzname` и `utcoffset`, которые не принимают аргументов и делегируют выполнение экземпляру *t*`.tzinfo`, возвращая значение `None`, если *t*`.tzinfo` равно `None`.

Класс `datetime`

Экземпляры класса `datetime` представляют время (дату с конкретным временем суток, относящимся к данной дате), могут учитывать или не учитывать часовой пояс и всегда игнорируют високосные секунды. Класс `datetime` расширяет класс `date` и добавляет атрибуты класса `time`. Его экземпляры обладают доступными только для чтения целочисленными атрибутами `year`, `month`, `day`, `hour`, `minute`, `second` и `microsecond` и необязательным атрибутом `tzinfo` (равным `None` в случае экземпляров, не учитывающих часовой пояс).

Экземпляры класса `datetime` поддерживают некоторые арифметические операции: разность между экземплярами `datetime` (оба они должны учитывать или не учитывать часовой пояс) является экземпляром `timedelta`, а экземпляры `timedelta`

могут участвовать в операциях сложения и вычитания с участием экземпляров datetime для создания других экземпляров datetime. Допускается сравнение двух экземпляров класса datetime между собой (*б*ольшим считается экземпляр, которому соответствует более позднее время суток), но только в том случае, если оба они учитывают или не учитывают часовой пояс.

datetime

datetime(*year*, *month*, *day*, *hour*=0, *minute*=0, *second*=0, *microsecond*=0, *tzinfo*=None)

Класс datetime также предоставляет некоторые методы класса, используемые в качестве альтернативы конструкторам.

combine

datetime.combine(*date*, *time*)

Возвращает объект datetime с атрибутами даты, взятыми из атрибутов заданных объектов date и time (включая tzinfo). Вызов datetime.combine(*d*, *t*) аналогичен следующему вызову:

```
datetime(d.year, d.month, d.day,
         t.hour, t.minute, t.second,
         t.microsecond, t.tzinfo)
```

fromordinal

datetime.fromordinal(*ordinal*)

Возвращает объект datetime, который соответствует дате, отстоящей на *ordinal* дней от начала нашей эры согласно пролептическому григорианскому календарю, причем значению 1 соответствует полночь первого дня года 1 н. э.

fromtimestamp

datetime.fromtimestamp(*timestamp*, *tz*=None)

Возвращает объект datetime, который соответствует отметке времени *timestamp* и представляет местное время в секундах от начала эпохи Unix. Если аргумент *tz* не равен None, возвращает объект datetime, учитывающий часовой пояс в соответствии с заданным экземпляром *tz* класса *tzinfo*.

now

datetime.now(*tz*=None)

Возвращает объект datetime, соответствующий текущей локальной дате и времени. Если аргумент *tz* не равен None, возвращает объект datetime, учитывающий часовой пояс в соответствии с заданным экземпляром *tz* класса *tzinfo*.

strptime

datetime.strptime(*str*, *fmt*='%a %b %d %H:%M:%S %Y %z')

Возвращает объект datetime, представляющий строку *str* в соответствии со спецификациями строки формата *fmt*. В версии v3, если указан спецификатор %z, результирующий объект datetime учитывает информацию о часовом поясе.

today

datetime.today()

Возвращает не учитывающий часовой пояс объект datetime, который представляет текущую локальную дату и время и аналогичен методу класса now (но не поддерживает необязательный аргумент *tz*).

utcfromtimestamp `datetime.utcfromtimestamp(`*timestamp*`)`

Возвращает не учитывающий часовой пояс объект `datetime`, который выражает время UTC, представленное аргументом *timestamp*, в секундах от начала эпохи Unix.

utcnow `datetime.utcnow()`

Возвращает не учитывающий часовой пояс объект `datetime`, который представляет текущую дату и время UTC.

Экземпляр *d* класса `datetime` также предоставляет следующие методы.

astimezone *d*`.astimezone(`*tz*`)`

Возвращает новый объект `datetime`, учитывающий часовой пояс и аналогичный объекту *d* (который также должен учитывать часовой пояс), за исключением того, что часовой пояс преобразуется в часовой пояс, определяемый экземпляром *tz*. Обратите внимание на то, что *d*`.astimezone(`*tz*`)` — это не то же самое, что *d*`.replace(tzinfo=tz)`: последний вызов не преобразует часовой пояс, а всего лишь копирует все атрибуты экземпляра *d*, за исключением *d*`.tzinfo`.

ctime *d*`.ctime()`

Возвращает строку, представляющую дату и время *d* в том же 24-символьном формате, что и функция `time.ctime`.

date *d*`.date()`

Возвращает объект `date`, представляющий ту же дату, что и *d*.

isocalendar *d*`.isocalendar()`

Возвращает дату, представленную объектом *d*, в виде кортежа из трех целых чисел, выражающих год, номер недели в году и день недели в соответствии со стандартом ISO.

isoformat *d*`.isoformat(sep='T')`

Возвращает строку, представляющую объект *d* в формате `'YYYY-MM-DDxHH:MM:SS'`, где *x* — значение аргумента *sep* (должно быть строкой длиной 1). Если *d*`.microsecond!=0`, то вслед за частью `'SS'` строки добавляются семь символов `'.mmmmmm'`. Если объект *d* учитывает часовой пояс, то в конце строки добавляются дополнительные шесть символов `'+HH:MM'`, представляющих смещение часового пояса относительно UTC. Другими словами, эта операция форматирования следует стандарту ISO 8601. Вызов `str(`*d*`)` — это то же самое, что вызов `d.isoformat(' ')`.

isoweekday *d*`.isoweekday()`

Возвращает соответствующий объекту *d* день недели в виде целого числа в диапазоне от `1` (понедельник) до `7` (воскресенье).

replace *d*.replace(*year*=None, *month*=None, *day*=None, *hour*=None, *minute*=None, *second*=None, *microsecond*=None[,tzinfo])

Возвращает новый объект datetime, аналогичный *d*, за исключением атрибутов, явно указанных в качестве аргументов, которые получают новые значения. Например:

```
datetime(x,y,z).replace(month=m) == datetime(x,m,z)
```

strftime *d*.strftime(*fmt*)

Возвращает строку, представляющую объект *d* в соответствии со спецификациями строки формата *fmt*.

time *d*.time()

Возвращает объект time, не учитывающий часовой пояс, который представляет то же время суток, что и объект *d*.

timestamp *d*.timestamp()

Возвращает значение в виде числа с плавающей точкой, представляющее количество секунд, истекших от начала эпохи (только в версии v3). Предполагается, что экземпляры, не учитывающие часовой пояс, относятся к местному часовому поясу.

timetz *d*.timetz()

Возвращает объект time, представляющий то же время суток, что и объект *d*, и с тем же часовым поясом.

timetuple *d*.timetuple()

Возвращает временной кортеж, который соответствует времени, представленному объектом *d*.

toordinal *d*.toordinal()

Возвращает порядковый номер дня по пролептическому григорианскому календарю для даты, представленной объектом *d*. Например:

```
datetime(1,1,1).toordinal() == 1
```

utctimetuple *d*.utctimetuple()

Возвращает временной кортеж, соответствующий моменту времени *d*, нормализованному по UTC, если *d* учитывает часовой пояс

weekday *d*.weekday()

Возвращает день недели, соответствующий дате *d*, в виде целого числа от 0 (понедельник) до 6 (воскресенье). Кроме того, экземпляр *d* класса datetime предоставляет методы dst, tzname, и utcoffset, которые не имеют никаких аргументов и делегируют выполнение вызова объекту *d*.tzinfo, возвращая None, если *d*.tzinfo равно None.

Класс timedelta

Экземпляры класса timedelta представляют временные интервалы с тремя доступными только для чтения целочисленными атрибутами: days, seconds и microseconds.

timedelta timedelta(*days*=0, *seconds*=0, *microseconds*=0, *milliseconds*=0, *minutes*=0, *hours*=0, weeks=0)

Преобразует все единицы с использованием очевидных множителей (неделя — 7 дней, час — 3600 секунд и т.д.) и нормализует их до указанных трех целочисленных атрибутов, убеждаясь, что 0<=seconds<3600*24 и 0<=microseconds<1000000. Ниже приведен соответствующий пример.

```
print(repr(timedelta(minutes=0.5)))
# вывод: datetime.timedelta(0, 30)
print(repr(timedelta(minutes=-0.5)))
# вывод: datetime.timedelta(-1, 86370)
```

Экземпляры timedelta поддерживают арифметические операции (+ и - между собой и с экземплярами классов date и datetime; * и / с целыми числами) и операции сравнения между собой. В версии v3 также поддерживаются операции деления между экземплярами timedelta (деление, целочисленное деление, divmod, %). Метод экземпляра total_seconds возвращает полное количество секунд, содержащихся в экземпляре timedelta.

Модуль pytz

Сторонний модуль pytz предлагает самые простые и удобные средства для создания экземпляров tzinfo, обеспечивающих возможность использования информации о часовых поясах экземплярами классов time и datetime. Модуль pytz основан на библиотеке Olson (http://www.twinsun.com/tz/tz-link.htm), предназначенной для работы с часовыми поясами. Как почти любой сторонний пакет Python, модуль pytz доступен на сайте PyPI (https://pypi.python.org/pypi): достаточно лишь выполнить команду **pip install pytz**.

Работа с часовыми поясами

Наилучший способ обойти все возможные ловушки и подводные камни при работе с часовыми поясами состоит в том, чтобы использовать в программе только часовой пояс UTC, преобразуя в него все входные часовые пояса и преобразуя его в другие часовые пояса исключительно для целей отображения.

Модуль pytz предоставляет атрибуты common_timezones — список, содержащий свыше 400 строк с названиями наиболее употребительных часовых поясов, которые могут вам понадобиться для использования (главным образом в виде континент/город с некоторыми синонимами наподобие 'UTC' и 'US/Pacific'),

и `all_timezones` — список, содержащий свыше 500 строк, которые задают другие синонимы для часовых поясов. Например, каноническим представлением часового пояса Лиссабона (Португалия) в соответствии со стандартами библиотеки Olson является `'Europe/Lisbon'`, и это именно то, что вы найдете в атрибуте `common_timezones`. Однако допускается также название часового пояса `'Portugal'`, которое содержится только в атрибуте `all_timezones`. Кроме того, модуль `pytz` предоставляет атрибуты `utc` и `UTC` — два имени одного и того же объекта `tzinfo`, задающего универсальное координированное время (UTC).

Модуль `pytz` также содержит следующие две функции.

`country_timezones` `country_timezones(`*code*`)`

Возвращает список названий часовых поясов, соответствующих стране, двухбуквенный код ISO которой задан аргументом *code*. Например, вызов `pytz.country_timezones('US')` возвращает список из 22 строк от `'America/New_York'` до `'Pacific/Honolulu'`.

`timezone` `timezone(`*name*`)`

Возвращает экземпляр класса `tzinfo`, соответствующий часовому поясу *name*. Например, чтобы вывести время в Нью-Йорке, эквивалентное полуночи 31 декабря 2005 года, можно использовать следующий код:

```
dt = datetime.datetime(2005,12,31,
tzinfo=pytz.timezone('America/New_York'))
print(dt.astimezone(
  pytz.timezone('Pacific/Honolulu')))
# prints: 2005-12-30 19:00:00-10:00
```

Пакет `dateutil`

Сторонний пакет `dateutil` (который можно установить с помощью команды **`pip installpython-dateutil`**) предлагает модули для выполнения всевозможных операций над датами, позволяя работать с временны*ми* интервалами, повторяющимися событиями, часовыми поясами, датами Пасхи, а также выполнять парсинг на основе нечетких правил. (Для ознакомления с полной документацией этого пакета посетите сайт `https://labix.org/python-dateutil`.) Основные модули пакета `dateutil` описаны ниже.

`easter` `easter.easter(`*year*`)`

Возвращает объект `datetime.date` для даты, на которую приходится Пасха в заданном году. Например, следующий код

```
from dateutil import easter
print(easter.easter(2006))
```

выводит `2006-04-16`.

parser `parser.parse(s)`

Возвращает объект `datetime.date`, обозначенный строкой *s*, используя очень свободные (также используется термин "нечеткие") правила синтаксического анализа. Например, следующий код

```
from dateutil import parser
print(parser.parse('Saturday, January 28, 2006,
                                    at 11:15pm'))
```

выводит `2006-01-28 23:15:00`.

relativedelta `relativedelta.relativedelta(...)`

Функцию `relativedelta` можно вызвать с двумя экземплярами `datetime.datetime`: результирующий экземпляр `relativedelta` захватывает относительную разность между этими двумя аргументами. Кроме того, функцию `relativedelta` можно вызвать с одним именованным аргументом, представляющим абсолютную информацию (*year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*) или относительную информацию (*years*, *months*, *weeks*, *days*, *hours*, *minutes*, *seconds*, *microseconds*), которая может иметь положительные или отрицательные значения. Другой вариант — вызов со специальным именованным аргументом *weekday*, который может быть числом в диапазоне от `0` (понедельник) до `6` (воскресенье) или одним из атрибутов `MO`, `TU`... `SU` модуля с числовым аргументом *n*, указывающим на *n*-й день недели. В любом случае результирующий экземпляр `relativedelta` захватывает информацию, передаваемую в вызове. Например, после выполнения приведенного ниже кода *r* означает "следующий понедельник".

```
from dateutil import relativedelta
r = relativedelta.relativedelta(
                    weekday=relativedelta.MO(1))
```

Экземпляр `relativedelta` можно прибавить к экземпляру `datetime.datetime` и получить новый экземпляр `datetime.datetime`, отличающийся от предыдущего на заданную относительную дельту.

```
print(datetime.datetime(2006,1,29)+r)
# вывод: 2006-01-30
print(datetime.datetime(2006,1,30)+r)
# вывод: 2006-01-30
print(datetime.datetime(2006,1,31)+r)
# вывод: 2006-02-06
```

Обратите внимание на то, что в соответствии со способом интерпретации объекта `relativedelta` "следующий понедельник" совпадает с самой датой, если соответствующий день уже является понедельником (поэтому более точным определением было бы "первая или следующая за указанной дата, на которую приходится понедельник"). Правила, определяющие неизбежно сложное поведение экземпляров `relativedelta`, подробно объясняются на сайте модуля `dateutil` (`https://labix.org/python-dateutil`).

rrule `rrule.rrule(freq, ...)`

Модуль `rrule` реализует рекомендации документа RFC2445 (`https://www.ietf.org/rfc/rfc2445.txt`), также известного как *iCalendar RFC*, во всей полноте его 140 с лишним страниц. Аргумент *freq* должен быть одним из постоянных атрибутов модуля `rrule`: `YEARLY`, `MONTHLY`, `WEEKLY`, `DAILY`, `HOURLY`, `MINUTELY` или `SECONDLY`. После обязательного аргумента *freq* могут быть указаны некоторые из многочисленных необязательных именованных аргументов, таких как *interval*=2, позволяющих указать, что периодичность учитывается лишь в отношении определенного подмножества заданных периодов (например, `rrule.rrule(rrule.YEARLY)` повторяется каждый год, тогда как `rrule.rrule(rrule.YEARLY, interval=7)` повторяется лишь каждые семь лет, как в случае творческого отпуска сроком до года, предоставляемого преподавателям колледжей раз в семь лет).

Экземпляр *r* типа `rrule.rrule` поддерживает несколько методов.

after *r*`.after(`*d*`, `*inc*`=False)`

Возвращает самый ранний экземпляр `datetime.datetime`, который является вхождением рекуррентного правила *r* и наступает после даты *d* (если *inc* равно `True`, то вхождение, приходящееся на саму дату *d*, также считается допустимым).

before *r*`.before(`*d*`, `*inc*`=False)`

Возвращает самый поздний экземпляр `datetime.datetime`, который является вхождением рекуррентного правила *r* и наступает перед датой *d* (если *inc* равно `True`, то вхождение, приходящееся на саму дату *d*, также считается допустимым).

between *r*`.between(`*start*`, `*finish*`, `*inc*`=False)`

Возвращает все экземпляры `datetime.datetime`, которые являются вхождениями рекуррентного правила *r* и наступают в течение промежутка времени между датами *start* и *finish* (если аргумент *inc* равен `True`, то вхождения, приходящиеся на сами даты *start* и *finish*, также считаются допустимыми). Например, правилу "раз в неделю на протяжении января 2018 года" соответствует следующий фрагмент кода:

```
start=datetime.datetime(2018,1,1)
r=rrule.rrule(rrule.WEEKLY, dtstart=start)
for d in r.between(start,
                   datetime.datetime(2018,2,1),True):
    print(d.date(),end=' ')
```

Он выводит `2018-01-01 2018-01-08 2018-01-15 2018-01-22 2018-01-29`.

count *r*`.count()`

Возвращает количество вхождений рекуррентного правила *r* (может создавать бесконечный цикл, если *r* имеет неограниченное количество вхождений).

Модуль sched

Модуль sched реализует планировщик событий, позволяющий без труда (причем как в случае единственного потока выполнения, так и в многопоточной среде) работать с событиями, для планирования которых может использоваться как "реальная", так и "имитируемая" шкала времени. Модуль sched предоставляет класс scheduler.

scheduler `class scheduler([timefunc],[delayfunc])`

Аргументы *timefunc* и *delayfunc* — обязательные в версии v2 и необязательные в версии v3 (в которой по умолчанию ими являются функции time.monotonic и time.sleep соответственно).

Аргумент *timefunc* должен быть вызываемым без аргументов объектом, позволяющим получать экземпляр текущего времени (в любых единицах измерения). Например, можно передать функцию time.time (или, только в версии v3, функцию time.monotonic).

Аргумент *delayfunc* вызывается с одним аргументом (временна*я* длительность, выраженная с использованием тех же единиц, что и в аргументе *timefunc*), определяющим задержку выполнения текущего потока на это время. Например, можно передать функцию time.sleep. Класс scheduler вызывает функцию delayfunc(0) после каждого события, чтобы предоставить другим потокам возможность выполнения; этот вызов совместим с вызовом time.sleep. Принимая функции в качестве аргументов, класс scheduler позволяет вам использовать любую "имитацию времени" или "псевдовремя", которые подходят для вашего приложения: отличный пример внедрения зависимостей (https://ru.wikipedia.org/wiki/Внедрение_зависимости) для целей, не обязательно связанных с тестированием.

Если для вашего приложения важно, чтобы время изменялось монотонно (время не может повернуть вспять, даже если системные часы были переведены назад между вызовами, например для учета високосных секунд), то для планировщика следует использовать функцию time.monotonic, предлагаемую версией v3. Экземпляр класса scheduler предоставляет следующие методы.

cancel `s.cancel(event_token)`

Удаляет событие из очереди *s*. Аргумент *event_token* должен быть результатом предыдущего вызова *s*.enter или *s*.enterabs, при этом событие еще не должно произойти; в противном случае возбуждается исключение RuntimeError.

empty `s.empty()`

Возвращает значение True, если текущая очередь *s* пуста; в противном случае возвращается значение False.

enterabs `s.enterabs(when, priority, func, args=(), kwargs={})`

Планирует будущее событие (обратный вызов функции *func*(*args*, *kwargs*)) в момент времени *when*. Эта сигнатура относится к версии v3; в версии v2 последовательность *args* — обязательна, а отображение *kwargs* не допускается.

Аргумент *when* измеряется в единицах, используемых функциями времени объекта *s*. Если на одно и то же время нужно запланировать несколько событий, то *s* выполняет их в порядке возрастания приоритета. Метод `enterabs` возвращает ярлык события *t*, который можно передать методу `s.cancel` для отмены данного события.

enter *s*.`enter`(*delay*, *priority*, *func*, *args*=(), *kwargs*={})

Подобен методу `enterabs`, за исключением того, что аргумент *delay* означает относительное время (положительное смещение в сторону возрастания относительно текущего момента времени), тогда как аргумент *when* метода `enterabs` — абсолютное время (будущий момент времени). В версии v3 добавлен аргумент *kwargs*, а аргумент *args* стал необязательным.

Чтобы запланировать событие для повторного выполнения, используйте небольшую функцию-оболочку.

```
def enter_repeat(s, first_delay,
                 period, priority, func, args):
    def repeating_wrapper():
        s.enter(period, priority, repeating_wrapper, ())
        func(*args)
    s.enter(first_delay, priority, repeating_wrapper, ())
```

run *s*.`run`(*blocking*=`True`)

Выполняет запланированные события. В версии v2, или если аргумент *blocking* равен `True` (только в версии v3), вызов *s*.`run` выполняется циклически до тех пор, пока вызов *s*.`empty()` не вернет значение `True`, используя функцию *delayfunc*, переданную при инициализации объекта *s*, для ожидания каждого запланированного события. Если аргумент *blocking* равен `False` (только в версии v3), выполняет любые события, срок действия которых должен скоро истечь, а затем возвращает срок завершения следующего события (если таковое имеется). Если функция обратного вызова *func* возбуждает исключение, *s* распространяет его, но поддерживает собственное состояние, удаляя данное событие из расписания. Если функция обратного вызова *func* выполняется дольше, чем время, оставшееся до наступления следующего события, то ее выполнение замедляется, но все события выполняются в порядке очереди, и ни одно событие не отменяется. Если какое-либо событие уже не представляет интереса, его можно отменить явным вызовом метода *s*.`cancel`.

Модуль `calendar`

Модуль `calendar` предоставляет функции для работы с календарем, включая функции, обеспечивающие вывод на печать текстового календаря для заданного месяца или года. По умолчанию в модуле `calendar` понедельник считается первым днем недели, а воскресенье — последним. Чтобы изменить это поведение, вызовите функцию `calendar.setfirstweekday`. Модуль `calendar` обрабатывает годы в диапазоне, используемом модулем `time`, в типичных случаях (по крайней мере) — в диапазоне 1970–2038.

Команда **python -m calendar** предлагает полезный интерфейс командной строки к функциональности данного модуля. Для получения короткой справки достаточно выполнить команду **python -m calendar -h**.

Модуль `calendar` предоставляет следующие функции.

calendar `calendar(year, w=2, l=1, c=6)`

Возвращает многострочную строку с календарем на год *year*, который отформатирован в три столбца, разделенных *c* пробелами. Аргумент *w* — ширина поля для каждой даты, выражаемая количеством символов. Длина каждой строки составляет `21*w+18+2*c`. Аргумент *l* — количество строк для каждой недели.

firstweekday `firstweekday()`

Возвращает текущую настройку для дня, которым начинается каждая неделя. По умолчанию, когда модуль `calendar` импортируется впервые, этот параметр равен `0`, что соответствует понедельнику.

isleap `isleap(year)`

Возвращает значение `True`, если год високосный, иначе — `False`.

leapdays `leapdays(y1, y2)`

Возвращает общее количество високосных дней в диапазоне годов `range(y1,y2)` (не забывайте о том, что год *y2* не включается в диапазон).

month `month(year, month, w=2, l=1)`

Возвращает многострочную строку календаря для месяца *month* года *year* по одной строке на неделю плюс две строки заголовков. Аргумент *w* — ширина поля для каждой даты; длина каждой строки составляет `7*w+6`. Аргумент *l* — количество строк для каждой недели.

monthcalendar `monthcalendar(year, month)`

Возвращает список списков целых чисел. Каждый подсписок относится к неделе. Дни, выходящие за пределы месяца, устанавливаются в `0`. Дни в пределах месяца устанавливаются в соответствующее число месяца от `1` и выше.

monthrange `monthrange(year, month)`

Возвращает два целых числа. Первое число — код дня недели для первого дня месяца *month* года *year*; второе число — количество дней в месяце. Коды дней недели изменяются в пределах от `0` (понедельник) до `6` (воскресенье). Номера месяцев изменяются в пределах от `1` до `12`.

prcal `prcal(year, w=2, l=1, c=6)`

Аналогичен вызову `print(calendar.calendar(year, w, l, c))`.

prmonth `prmonth(year, month, w=2, l=1)`

Аналогичен вызову `print(calendar.month(year, month, w, l))`.

setfirstweekday `setfirstweekday(`*weekday*`)`

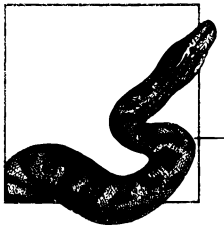
Устанавливает первый день каждой недели в код дня недели *weekday*. Коды дней недели изменяются в пределах от 0 (понедельник) до 6 (воскресенье). Модуль `calendar` предоставляет атрибуты `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` и `SUNDAY`, значениями которых являются целые числа от 0 до 6. Используйте эти атрибуты там, где подразумеваются дни недели (например, используйте `calendar.FRIDAY` вместо 4), чтобы сделать код более понятным и удобочитаемым.

timegm `timegm(`*tupletime*`)`

Выполняет операцию, обратную операции `time.gmtime`: принимает время в виде временного кортежа и возвращает время в виде числа с плавающей точкой, выражающего количество секунд, истекших от начала эпохи Unix.

weekday `weekday(`*year*`, `*month*`, `*day*`)`

Возвращает код дня недели для заданной даты. Коды дней недели изменяются в пределах от 0 (понедельник) до 6 (воскресенье). Номера месяцев изменяются в пределах от 1 (январь) до 12 (декабрь).



13

Управление процессом выполнения

Python непосредственно предоставляет, поддерживает и документирует многие из своих внутренних механизмов. Данный фактор может быть полезным для более глубокого понимания Python, что поможет вам организовать взаимодействие собственного кода с этими механизмами и в определенной степени управлять ими. Например, в разделе "Встроенные объекты Python" главы 6 рассказывалось о том, как Python делает видимыми встроенные модули. Эта глава посвящена обсуждению других продвинутых приемов работы с Python. В главе 16 рассматриваются специфические вопросы, касающиеся тестирования, отладки и профилирования программ. К кругу вопросов, связанных с управлением процессом выполнения, также относятся многопоточные и многозадачные вычисления (глава 14) и асинхронная обработка (глава 18).

Адаптация настроек узла и пользователя

Python предоставляет специальный "перехватчик" (расширение, "хук"), позволяющий каждому сайту (узлу) настраивать определенные аспекты поведения Python в начале каждого запуска. По умолчанию возможность настройки индивидуальных пользовательских параметров отключена, но Python определяет, каким образом программы, при запуске которых требуется выполнять пользовательский код, могут обеспечить явную поддержку такой настройки (редко используемая возможность).

Модули `site` и `sitecustomize`

Перед загрузкой основного сценария Python загружает стандартный модуль `site`. Если Python запускается с опцией `-S`, то модуль `site` не загружается. В этом случае запуск Python ускоряется, но на основной сценарий ложится дополнительная нагрузка по выполнению операций инициализации. Модуль `site` решает следующие задачи.

- Приводит список путей `sys.path` к стандартной форме (абсолютные пути без повторений).
- Интерпретирует каждый из файлов с расширением *.pth*, обнаруженных в домашнем каталоге Python, добавляя записи в `sys.path` и/или импортируя модули в соответствии с содержимым каждого *.pth*-файла.
- Добавляет встроенные модули, используемые для вывода информации в интерактивных сеансах (`exit`, `copyright`, `credits`, `license` и `quit`).
- Только в версии v2: устанавливает заданную по умолчанию кодировку Unicode в `'ascii'` (в версии v3 по умолчанию используется встроенная кодировка `'utf-8'`). В версии v2 исходный код модуля `site` включает два блока, и каждый из них защищается инструкциями `if 0:`, одна из которых устанавливает кодировку по умолчанию, зависящую от региональных настроек, а вторая полностью отключает любое заданное по умолчанию преобразование между строками Unicode и простыми строками. Вам предоставляется возможность редактировать содержимое файла *site.py* для выбора любого блока, но такое решение нельзя считать удачным, несмотря на то, что в одном из комментариев в этом файле сказано, что вы вправе экспериментировать, изменяя его содержимое.
- Только в версии v2: пытается импортировать модуль `sitecustomize` (если инструкция `import sitecustomize` возбуждает исключение `ImportError`, то модуль `site` перехватывает и игнорирует его). Это модуль, который может использоваться на каждом сайте для необязательной, специфической для данного сайта настройки сверх той, которая осуществляется модулем `site`. Лучше всего не изменять содержимое файла *site.py*, поскольку любое обновление или переустановка Python отменит внесенные изменения.
- По завершении выполнения модуля `sitecustomize` удаляет атрибут `sys.setdefaultencoding` из модуля `sys`, чтобы сделать невозможным изменение кодировки, используемой по умолчанию.

Индивидуальные настройки пользователей

Каждый сеанс интерактивной работы с интерпретатором Python начинается с выполнения сценария, указанного в переменной среды `PYTHONSTARTUP`. Вне интерактивных сеансов возможности автоматической настройки параметров для отдельных пользователей отсутствуют. Для запроса пользовательских настроек основной сценарий Python (только в версии v2) может явно выполнить инструкцию `import user`. При загрузке модуля `user`, входящего в стандартную библиотеку версии v2, прежде всего определяется домашний каталог данного пользователя, указанный в переменной среды `HOME` (а в случае неудачи — в переменной `HOMEPATH`, приоритет над которой (только в системах Windows) может иметь переменная `HOMEDRIVE`). Если найти домашний каталог в переменных среды не удается, то модуль `user` использует

текущий каталог. Если модуль `user` находит файл *.pythonrc.py* в указанном каталоге, он выполняет этот файл с помощью встроенной в версию Python v2 функции `execfile` в собственном глобальном пространстве имен пользователя.

Сценарии, не импортирующие модуль `user`, не выполняют файл *.pythonrc.py*; не выполняют его и сценарии в версии Python v3, поскольку в ней не определен модуль `user`. Разумеется, любой сценарий может свободно воспользоваться любым другим способом выполнения специфического для конкретного пользователя модуля во время своего запуска. Организация подобного рода специфических для приложения способов запуска, даже в версии v2, более распространена, чем импортирование модуля `user`. Типичный файл *.pythonrc.py*, загружаемый посредством инструкции `import user`, должен быть пригоден для использования с любым приложением, которое его загружает. От специализированных, специфических для приложения файлов, выполняемых при запуске, требуется лишь то, чтобы они следовали правилам, документированным для конкретного приложения.

Например, ваше приложение *MyApp.py* может документировать, что оно будет искать файл *.myapprc.py* в домашнем каталоге пользователя, указанном в переменной среды `HOME`, и загружать его в глобальное пространство имен основного сценария приложения, который может содержать следующий код.

```
import os
homedir = os.environ.get('HOME')
if homedir is not None:
    userscript = os.path.join(homedir, '.myapprc.py')
    if os.path.isfile(userscript):
        with open(userscript) as f:
            exec(f.read())
```

В этом случае сценарий пользовательских настроек *.myapprc.py*, если он имеется, должен решать задачи, специфические для конкретного пользователя приложения `MyApp`. Этот подход лучше того, который основан на использовании модуля `user`, и одинаково хорошо работает как в версии v3, так и в версии v2.

Функции завершения

Модуль `atexit` позволяет регистрировать функции завершения (т.е. функции, которые вызываются по завершении программы, подчиняясь принципу "последним пришел, первым ушел"). Функции завершения аналогичны обработчикам, устанавливаемым с помощью инструкций `try/finally` или `with` и предназначенным для очистки ресурсов. Однако функции завершения регистрируются в глобальном пространстве имен и вызываются по завершении всей программы, тогда как указанные обработчики устанавливаются лексически и вызываются по завершении конкретной инструкции `try` или `with`. Функции завершения и функции очистки ресурсов вызываются как при обычном, так и при аварийном завершении программы, но не тогда, когда программа завершается путем вызова функции `os._exit` (вот почему

вместо нее обычно вызывают функцию sys.exitinstead). Модуль atexit предоставляет описанную ниже функцию register.

register register(*func*,**args*,***kwds*)

Гарантирует, что по завершении работы программы будет вызвана функция *func*(**args*,***kwds*).

Динамическое выполнение и инструкция exec

Инструкция exec (в версии v3 — встроенная функция) может выполнить код, который вы читаете, генерируете или иным образом получаете во время выполнения программы. Инструкция exec динамически выполняет предоставленную ей инструкцию или набор инструкций. В версии v2 exec — это просто ключевое слово инструкции, имеющей следующий синтаксис:

```
exec code[ in globals[, locals]]
```

Параметр *code* может быть строкой, открытым объектом наподобие файла или объектом кода, а *globals* и *locals* — отображения. В версии v3 exec — встроенная функция, имеющая следующий синтаксис:

```
exec(code, globals=None, locals=None)
```

Параметр *code* может быть строкой, байтовой строкой или объектом кода, *globals* — словарем, а *locals* — любым отображением.

Если заданы как *globals*, так и *locals*, то они являются соответственно глобальным и локальным пространствами имен, в которых выполняется код. Если задан только аргумент *globals*, то exec использует *globals* в качестве обоих пространств имен. Если ни один из этих аргументов не задан, код выполняется в текущей области видимости.

Никогда не используйте функцию exec в текущей области видимости

Выполнение функции exec в текущей области видимости — очень плохая идея: это может привести к связыванию, повторному связыванию или откреплению любого глобального имени. Чтобы держать все под контролем, используйте функцию exec, если это вообще необходимо, лишь в отношении конкретных, явно определенных словарей.

Избегайте использования функции exec

От тех, кто работает с Python, часто приходится слышать вопрос: "Как установить переменную, имя которой я создаю на лету или читаю?" Вообще говоря, в случае глобальной переменной это можно сделать с помощью функции exec, но это плохая идея. Например, если имя переменной — varname, то вы могли бы подумать, что можно воспользоваться следующей инструкцией:

```
exec(varname + '= 23')
```

Никогда так не делайте! Применение функции `exec` в текущей области видимости лишает вас контроля над собственным пространством имен, из-за чего в код могут вкрасться трудно обнаруживаемые ошибки, что чрезвычайно усложнит понимание логики выполнения вашей программы. Храните "переменные", которые вам нужно устанавливать, не в виде переменных, а в виде записей в словаре, скажем, в словаре `mydict`. В нашем случае можно было бы поступить следующим образом:

```
exec(varname + '=23', mydict)
```

Подобный вариант уже не столь ужасен, как предыдущий, но это все еще неудачная идея. Хранение подобных "переменных" в словаре означает, что вам вовсе не нужна функция `exec` для установки их значений. Для этого достаточно использовать следующий код:

```
mydict[varname] = 23
```

Такой подход ускорит работу вашей программы и сделает ее код более понятным, непосредственным и элегантным. Существуют ситуации, в которых использование функции `exec` является оправданным, однако они встречаются крайне редко, — вместо этого достаточно использовать словари.

Старайтесь не использовать функцию exec

Используйте функцию `exec` только в том случае, если без нее действительно нельзя обойтись, хотя такие ситуации встречаются крайне редко. Наилучший выход чаще всего состоит в том, чтобы по возможности избегать использования функции `exec` и выбирать более специфические, хорошо контролируемые механизмы: функция `exec` ослабляет ваши возможности по управлению пространством имен своего кода, может отрицательно сказаться на производительности программы, стать причиной многочисленных трудно обнаруживаемых ошибок и создать серьезные угрозы для безопасности приложения.

Выражения

Функция `exec` может выполнить выражение, поскольку любое выражение одновременно является действительной инструкцией (носящей название *инструкция-выражение*). Однако Python игнорирует значение, возвращаемое инструкцией-выражением. Чтобы вычислить выражение и получить его значение, воспользуйтесь встроенной функцией `eval` (см. табл. 7.2).

Функция `compile` и объекты `code`

Чтобы создать объект программного кода, предназначенный для использования с функцией `exec`, следует вызвать встроенную функцию `compile`, указав в качестве последнего аргумента `'exec'` (см. табл. 7.2).

Объект кода *c* предоставляет ряд заслуживающих внимания атрибутов, доступных только для чтения, имена которых начинаются с префикса `'co_'`.

`co_argcount`
: Количество параметров функции, программный код которой представлен объектом *c* (`0`, если *c* не является объектом кода функции и был создан непосредственно функцией `compile`).

`co_code`
: Байтовая строка, содержащая байт-код *c*.

`co_consts`
: Кортеж констант, используемых байт-кодом в *c*.

`co_filename`
: Имя файла, из которого был скомпилирован объект *c* (строка, являющаяся вторым аргументом функции `compile`, если объект *c* был создан именно таким способом).

`co_firstlinenumber`
: Номер первой строки исходного кода (в файле `co_filename`), компиляцией которого был получен объект *c*, если *c* создавался компиляцией из файла.

`co_name`
: Имя функции, кодом которой является *c* (`'<module>'`, если *c* не является объектом кода функции и был создан непосредственно функцией `compile`).

`co_names`
: Кортеж всех идентификаторов, используемых в объекте *c*.

`co_varnames`
: Кортеж идентификаторов локальных переменных объекта *c*, начинающийся с имен параметров.

Большинство этих атрибутов используются только в целях отладки, но некоторые из них могут оказаться полезными для углубленной интроспекции, пример которой приведен далее.

Если вы начинаете со строки, в которой хранятся инструкции, то прежде всего скомпилируйте ее с помощью функции `compile`, а затем выполните результирующий объект программного кода с помощью функции `exec` — это лучше, чем сразу предоставить строку функции `exec` для компиляции и выполнения. Такое разделение операций позволяет отделить нахождение синтаксических ошибок от ошибок времени выполнения. Часто можно организовать вычисления таким образом, чтобы строка компилировалась только один раз, а объект кода выполнялся многократно, что ускоряет работу программы. Кроме того, стадия компилирования внутренне безопасна (как с `exec`, так и с `eval` связаны чрезвычайно высокие риски, если с их помощью выполняется код, которому нельзя доверять), что позволяет проверить объект кода

до его выполнения и тем самым свести риски к минимуму (хотя это и не позволяет избавиться от них полностью).

Объект программного кода имеет доступный только для чтения атрибут `co_names`, который представляет собой кортеж имен, используемых в коде. Предположим, вы хотите, чтобы пользователь вводил выражение, которое содержит только литеральные константы и операторы, а не вызовы функций или другие имена. Прежде чем вычислять выражение, вы можете убедиться в том, что введенная пользователем строка удовлетворяет этим ограничениям.

```
def safer_eval(s):
    code = compile(s, '<user-entered string>', 'eval')
    if code.co_names:
        raise ValueError('No names {!r} allowed in
                         expression {!r}'
                        .format(code.co_names, s))
    return eval(code)
```

Функция `safer_eval` вычисляет выражение, переданное ей в качестве аргумента *s*, только если она является синтаксически правильным выражением (в противном случае функция `compile` возбуждает исключение `SyntaxError`) и вообще не содержит никаких имен (в противном случае функция `safer_eval` явным образом возбуждает исключение `ValueError`). (Функция `safer_eval` ведет себя аналогично функции стандартной библиотеки `ast.literal_eval`, рассмотренной в разделе "Стандартный ввод" главы 10, но предоставляет более широкие возможности, поскольку допускает использование операторов.)

Иногда знание того, к каким именам будет обращаться код, может помочь оптимизировать подготовку словаря, который нужно передавать функции `exec` или `eval` в качестве пространства имен. Поскольку вы должны предоставить значения только для этих имен, можно сэкономить часть работы, не подготавливая другие записи. Предположим, например, что ваше приложение динамически принимает код от пользователя, причем действует соглашение, в соответствии с которым имена переменных, начинающиеся с префикса *`data_`*, относятся к файлам, находящимся в подкаталоге *`data`*, которые код, написанный пользователем, не должен читать явным образом. В свою очередь пользовательский код может выполнять вычисления и оставлять результаты в глобальных переменных с именами, начинающимися с префикса *`result_`*, которые ваше приложение может записывать обратно в виде файлов в подкаталог *`data`*. Благодаря указанному соглашению вы можете впоследствии переместить данные в другое место (например, в виде объектов BLOB в базу данных вместо файлов, хранящихся в подкаталоге), и это никак не скажется на коде, написанном пользователем. Ниже приведен пример эффективной реализации этих соглашений (в версии v3; в версии v2 вместо вызова `exec(user_code, datadict)` следует использовать инструкцию `exec user_code in datadict`).

```
def exec_with_data(user_code_string):
    user_code = compile(user_code_string, '<user code>', 'exec')
    datadict = {}
    for name in user_code.co_names:
        if name.startswith('data_'):
            with open('data/{}'.format(name[5:]),
                      'rb') as datafile:
                datadict[name] = datafile.read()
    exec(user_code, datadict)
    for name in datadict:
        if name.startswith('result_'):
            with open('data/{}'.format(name[7:]),
                      'wb') as datafile:
                datafile.write(datadict[name])
```

Никогда не используйте функцию exec или eval для выполнения кода, не заслуживающего доверия

В старых версиях Python предпринимались попытки предоставить инструменты категории "ограниченное выполнение", уменьшающие риски использования инструкций `exec` и `eval`, но эти инструменты никогда не были в состоянии обеспечить полную безопасность от атак изобретательных хакеров, в связи с чем было принято решение отказаться от них в текущих версиях Python. Если вам необходимо оградиться от подобных атак, воспользуйтесь преимуществами механизмов защиты своей операционной системы: выполняйте код, не заслуживающий доверия, в отдельном процессе, ограничивая его привилегии в доступных вам пределах (изучите механизмы, предоставляемые вашей ОС для этих целей, такие как `chroot`, `setuid` и `jail`), или выполняйте ненадежный код в отдельной виртуальной машине со строгими ограничениями. В качестве меры защиты от атак типа "отказ в обслуживании" основной процесс может осуществлять мониторинг отдельного процесса и прекращать его выполнение, если наблюдается резкий рост потребления ресурсов. Управление процессами рассмотрено в разделе "Выполнение других программ" главы 14.

Использование функций exec и eval в отношении ненадежного кода не является безопасным

Использование функции `exec_with_data` в отношении ненадежного кода вообще не является безопасным: если в качестве аргумента *`user_code_string`* ей передается строка, полученная способом, который не является *полностью* безопасным, то размеры ущерба, который это может нанести, фактически ничем не ограничены. К сожалению, это относится почти ко всем случаям использования как функции `exec`, так и `eval`, за исключением тех редких ситуаций, когда у вас есть возможность установить жесткие и строго контролируемые ограничения для выполняемого с их помощью кода, что было продемонстрировано на примере функции `safer_eval`.

Внутренние типы

Практическое использование некоторых из рассмотренных в этом разделе внутренних типов сопряжено с определенными трудностями. Корректное применение таких объектов для достижения желаемого результата требует изучения исходного кода на языке C (или Java, или C#), используемого вашей реализацией Python. Подобные средства "черной магии" требуются редко, за исключением тех случаев, когда это необходимо для создания универсальных инструментов разработки или решения других сложных задач подобного рода. При достаточно глубоком понимании принципов работы данных средств Python позволяет вам контролировать их, если это действительно необходимо. Поскольку Python предоставляет вашему коду возможность доступа ко многим разновидностям внутренних объектов, вы можете контролировать указанные средства с помощью кода на языке Python, пусть даже для этого и понадобится знание языка C (или Java, или C#).

Типы объектов

Встроенный тип `type` ведет себя подобно вызываемой фабрике объектов, которая возвращает объекты, являющиеся типами. Объекты типа не обязаны поддерживать никакие другие специальные операции, кроме сравнения на равенство и представления в виде строк. Однако большинство объектов типа являются вызываемыми и возвращают новые экземпляры типа при их вызове. Именно так работают такие встроенные типы, как `int`, `float`, `list`, `str`, `tuple`, `set` и `dict`. В частности, будучи вызванными без аргумента, они возвращают новый пустой экземпляр или, в случае чисел, экземпляр, равный `0`. Атрибуты модуля `types` являются встроенными типами, каждый из которых имеет одно или несколько имен. Например, в версии v2 оба типа, `types.DictType` и `types.DictionaryType`, ссылаются на тип `type({})`, также известный как `dict`. В версии v3 модуль `types` предоставляет только имена встроенных типов, которые не имеют встроенного имени (см. в главе 7). Многие типы объектов полезны не только тем, что их можно вызывать для генерации экземпляров, но и тем, что на их основе можно создавать новые типы путем наследования (см. раздел "Классы и экземпляры" в главе 3).

Тип объектов программного кода

Кроме использования встроенной функции `compile`, для получения объекта программного кода можно воспользоваться атрибутом `__code__` объекта функции или метода. (Для получения информации относительно атрибутов объектов кода обратитесь к разделу "Функция `compile` и объекты `code`".) Объект кода не является вызываемым объектом, но вы можете обернуть его вызываемой оболочкой путем повторного связывания с атрибутом `__code__` объекта функции с подходящим количеством параметров.

```
def g(x): print('g', x)
code_object = g.__code__
def f(x): pass
f.__code__ = code_object
f(23)     # вывод: g 23
```

Объекты кода, не имеющие параметров, также могут использоваться совместно с функциями `exec` и `eval`. Чтобы создать новый объект, вызовите объект типа, который хотите инстанциализировать. Однако непосредственное создание объектов кода требует использования многих параметров. О том, как это можно сделать, рассказано в неофициальной документации на сайте Stack Overflow (`https://stackoverflow.com/questions/16064409/how-to-create-a-code-object-in-python`), но почти всегда вместо этого лучше вызвать функцию `compile`.

Тип `frame`

Функция `_getframe`, которая содержится в модуле `sys`, возвращает объект фрейма (кадра) стека вызовов Python. Объект фрейма обладает атрибутами, которые предоставляют информацию о коде, выполняющемся во фрейме, и состоянии выполнения. Модули `traceback` и `inspect` облегчают доступ к этой информации и ее отображение, особенно когда обрабатывается исключение. В главе 16 приводится больше информации о фреймах и трассировочной информации и рассматривается модуль `inspect`, реализующий наилучший способ выполнения подобной интроспекции.

Сборка мусора

Обычно механизм сборки мусора Python работает прозрачно в автоматическом режиме, но вам предоставляется возможность частичного управления этим процессом. Общий принцип заключается в том, что Python отбирает каждый объект *x* спустя некоторое время после того, как он становится недостижимым, т.е. тогда, когда не остается ни одной цепочки ссылок, начинающейся с локальной переменной выполняющейся функции или глобальной переменной загруженного модуля, которая вела бы к *x*. Обычно объект *x* становится недостижимым, если вообще не существует ссылок на *x*. Кроме того, группа объектов может быть недостижимой, если объекты ссылаются друг на друга, но не существует глобальных или локальных переменных, которые ссылались бы на любой из них, пусть даже косвенно (в подобных ситуациях говорят о *циклических ссылках*).

В классическом Python для каждого объекта *x* поддерживается так называемый *счетчик ссылок*, с помощью которого ведется подсчет количества ссылок, указывающих на *x*. Когда значение счетчика ссылок для объекта *x* уменьшается до `0`, CPython немедленно помечает этот объект для удаления. Функция `getrefcount` модуля `sys` получает в качестве аргумента любой объект и возвращает значение его счетчика ссылок (равное по крайней мере `1`, поскольку функция `getrefcount` сама ссылается

на исследуемый ею объект). Другие версии Python, такие как Jython или IronPython, полагаются на другие механизмы сборки мусора, предоставляемые платформой, на которой они выполняются (например, JVM или MSCLR). Поэтому модули `gc` и `weakref` применимы только в CPython.

Если механизм сборки мусора Python отбирает объект *x* и на этот объект вообще отсутствуют ссылки, то Python финализирует его (т.е. вызывает метод `x.__del__()`) и освобождает занимаемую им память для использования в других целях. Если *x* хранит ссылки на другие объекты, Python удаляет эти ссылки, что, в свою очередь, может сделать соответствующие объекты недостижимыми и подлежащими сборке в качестве мусора.

Модуль gc

Модуль `gc` реализует функциональность сборщика мусора Python и обрабатывает только недостижимые объекты, участвующие в циклических ссылках. В случае циклических ссылок объекты ссылаются друг на друга, поддерживая положительными значения счетчиков ссылок всех объектов, образующих цикл. При этом внешние ссылки, указывающие на любой объект из набора взаимно ссылающихся объектов, отсутствуют. В результате группа в целом, известная как *циклический мусор*, оказывается недостижимой и потому подлежит процедуре сборки мусора. Поиск циклического мусора занимает определенное время, и модуль `gc` помогает контролировать, когда именно программа должна выполнять этот затратный по времени поиск. Функциональность сборки циклического мусора по умолчанию включена с использованием разумных значений ряда параметров, заданных по умолчанию. Однако, импортируя модуль `gc` и вызывая его функции, вы можете отключить эту функциональность, изменить ее параметры и/или получить точные сведения о том, что делается в этом отношении.

Модуль `gc` предоставляет описанные ниже функции, используя которые вы сможете управлять запуском сборщика мусора. Иногда эти функции позволяют обнаруживать утечки памяти, обусловленные недостижимыми объектами, что облегчает выявление объектов, которые в действительности хранят ссылки на эти объекты.

collect `collect()`

Принудительно запускает полную сборку циклического мусора.

disable `disable()`

Приостанавливает автоматическую периодическую сборку циклического мусора.

enable `enable()`

Возобновляет периодическую сборку циклического мусора, ранее приостановленную с помощью функции `disable`.

garbage Доступный только для чтения атрибут, предоставляющий список недостижимых объектов, которые не могут быть удалены. Это происходит тогда, когда объект связан циклическими ссылками и имеет специальный метод `__del__`, поскольку в таком случае не существует очевидного безопасного способа, который Python мог бы использовать для финализации подобных объектов.

get_debug `get_debug()`

Возвращает строку, представляющую состояние флагов отладки сборщика мусора, установленных с помощью функции `set_debug`.

get_objects `get_objects()`

Возвращает список всех объектов, в данный момент контролируемых сборщиком циклического мусора.

get_referrers `get_referrers(*objs)`

Возвращает список всех контейнерных объектов, в данный момент контролируемых сборщиком циклического мусора, которые ссылаются на объекты, указанные в качестве аргументов.

get_threshold `get_threshold()`

Возвращает кортеж из трех элементов (*thresh0*, *thresh1*, *thresh2*), представляющих пороговые значения для запуска сборщика мусора, установленные с помощью функции `set_threshold`.

isenabled `isenabled()`

Возвращает значение `True`, если в данный момент времени механизм сборки циклического мусора включен; в противном случае возвращает значение `False`.

set_debug `set_debug(flags)`

Устанавливает флаги отладки для механизма сборки мусора. Аргумент *flags* представляет собой целое число, сформированное с помощью побитовой операции ИЛИ (`|`) из перечисленных ниже констант, определенных в модуле `gc`.

`DEBUG_COLLECTABLE`

Задает вывод информации об утилизируемых объектах, обнаруженных в процессе сборки мусора.

`DEBUG_INSTANCES`

Если установлен также флаг `DEBUG_COLLECTABLE` или `DEBUG_UNCOLLECTABLE`, то будет выводиться информация об обнаруженных в процессе сборки мусора объектах, являющихся экземплярами классов старого стиля (только в версии v2).

`DEBUG_LEAK`

Набор флагов отладки, вынуждающих механизм сборки мусора выводить всю информацию, которая может пригодиться при

диагностике утечки памяти. То же самое, что объединение с помощью побитовой операции ИЛИ всех остальных констант (за исключением константы DEBUG_STATS, которая служит для других целей).

DEBUG_OBJECTS

Если установлен также флаг DEBUG_COLLECTABLE или DEBUG_UNCOLLECTABLE, то будет выводиться информация об обнаруженных в процессе сборки мусора объектах, не являющихся экземплярами классов старого стиля (только в версии v2).

DEBUG_SAVEALL

Задает сохранение всех утилизируемых объектов в списке gc.garbage (в нем также всегда сохраняются объекты, которые не утилизируются), что может пригодиться для диагностики утечек памяти.

DEBUG_STATS

Задает вывод статистических данных в процессе сборки мусора, что может пригодиться для настройки пороговых значений.

DEBUG_UNCOLLECTABLE

Задает вывод информации об обнаруженных в процессе сборки мусора объектах, которые не утилизируются.

set_threshold set_threshold(*thresh0*[,*thresh1*[,*thresh2*]])

Устанавливает пороговые значения, управляющие частотой запуска механизма сборщика мусора. Значение *thresh0*, равное 0, отключает сборку мусора. Сборка мусора — сложная специальная тема, и детальное рассмотрение используемого в Python подхода (а следовательно, и смысла пороговых значений), основанного на понятии поколений объектов, выходит за рамки данной книги. Для получения более подробной информации по этому вопросу обратитесь к онлайн-документации.

Если вы знаете, что в вашей программе отсутствуют циклические ссылки, или если для вас недопустимо, чтобы на критическом участке программы возникла задержка, вызванная сборкой мусора, отключите автоматическую сборку мусора, вызвав функцию gc.disable(). Впоследствии вы сможете вновь включить сборку мусора, вызвав функцию gc.enable(). Проверить текущее состояние механизма сборки мусора можно с помощью вызова функции gc.isenabled(), возвращающей значение True или False. Вы можете сами выбирать, когда именно допустимо затратить время на сборку мусора, вызывая функцию gc.collect(), которая принудительно запускает этот процесс для немедленного выполнения. Критический в отношении времени выполнения код можно обернуть оболочкой примерно следующего вида.

```
import gc
gc_was_enabled = gc.isenabled()
if gc_was_enabled:
    gc.collect()
```

```
    gc.disable()
# Вставьте сюда критический в отношении времени
# выполонения код
if gc_was_enabled:
    gc.enable()
```

Модуль `gc` включает более сложную и редко используемую функциональность, которая охватывает два круга задач. Функции `get_threshold` и `set_threshold` вместе с флагом отладки `DEBUG_STATS` обеспечивают возможность тонкой настройки поведения механизма сборки мусора с целью оптимизации производительности программы. Остальная часть функциональности, предлагаемой модулем `gc`, облегчает диагностику утечек памяти в программе. Несмотря на то что во многих случаях модуль `gc` способен самостоятельно устранять утечки памяти (при условии, что вы избегаете определять метод `__del__` в своих классах, поскольку его наличие может блокировать утилизацию циклического мусора), ваша программа будет работать быстрее, если вы будете стараться прежде всего не допускать создания циклического мусора.

Модуль `weakref`

Тщательное проектирование программы часто помогает избежать образования циклических ссылок. Однако иногда возникает необходимость в том, чтобы объекты располагали информацией друг о друге, и в этом случае избавление от взаимных ссылок потребовало бы изменения и усложнения структуры программы. Например, контейнер содержит ссылки на свои элементы, однако часто не помешает, чтобы объекту также было известно, какому контейнеру он принадлежит. В результате образуются циклические ссылки: благодаря наличию взаимных ссылок контейнер и объект поддерживают друг друга в активном состоянии, даже если другим объектам о них ничего не известно. Эта проблема решается с помощью слабых ссылок, позволяющих объектам ссылаться друг на друга, но не поддерживать в активном состоянии.

Слабая ссылка — это специальный объект *w*, который ссылается на другой объект *x*, не инкрементируя его счетчик ссылок. Если счетчик ссылок объекта *x* уменьшается до `0`, то Python финализирует и собирает *x* как мусор, извещая *w* о прекращении существования *x*. Теперь слабая ссылка *w* может либо исчезнуть, либо быть помеченной как недействительная контролируемым способом. В любой момент времени объект *w* ссылается либо на тот же объект *x*, что и при создании *w*, либо вообще ни на что: слабая ссылка никогда не перенаправляется на другой объект. Не все типы объектов могут выступать в качестве целевого объекта *x* слабой ссылки *w*, но классы, экземпляры и функции годятся для этого.

Ниже описаны функции и типы модуля `weakref`, предназначенные для создания слабых ссылок и управления ими.

`getweakrefcount` `getweakrefcount(x)`

Возвращает значение `len(getweakrefs(x))`.

getweakrefs `getweakrefs(x)`

Возвращает список всех слабых ссылок и прокси-объектов, указывающих на *x*.

proxy `proxy(x[,f])`

Возвращает прокси-объект *p* типа `ProxyType` (`CallableProxyType`, если *x* — вызываемый объект), содержащий слабую ссылку на объект *x*. В большинстве контекстов использование *p* равносильно использованию *x*, за исключением того, что в случае использования объекта *p* после уничтожения объекта *x* Python возбуждает исключение `ReferenceError`. Объект *p* не хешируемый (таким образом, его нельзя использовать в качестве ключа в словаре), даже если объект *x* является таковым. Если задан аргумент *f*, то он должен быть вызываемым объектом, принимающим один аргумент и выступающим в качестве функции обратного вызова, финализирующей объект *p* (т.е. непосредственно перед финализацией *x* Python вызывает *f*(*p*)). Функция *f* выполняется сразу же после того, как объект *x* становится недостижимым из объекта *p*.

ref `ref(x[,f])`

Возвращает слабую ссылку *w* типа `ReferenceType`, указывающую на объект *x*. Объект *w* является вызываемым объектом, не принимающим аргументов: вызов *w*() возвращает *x*, если *x* все еще активен; в противном случае вызов *w*() возвращает значение `None`. Объект *w* является хешируемым, если объект *x* хешируемый. Для слабых ссылок допустимы операции отношения равенства (==, !=), но не порядка (<, >, <=, >=). Две слабые ссылки *x* и *y* считаются равными, если их целевые объекты находятся в активном состоянии и равны или если `x is y` равно `True`. Если задан аргумент *f*, то он должен быть вызываемым объектом, принимающим один аргумент и выступающим в качестве функции обратного вызова, финализирующей объект *w* (т.е. непосредственно перед финализацией *x* Python вызывает *f*(*w*)). Функция *f* выполняется сразу же после того, как объект *x* становится недостижимым из объекта *w*.

WeakKeyDictionary `class WeakKeyDictionary(adict={})`

Объект *d* типа `WeakKeyDictionary` — это отображение, использующее слабые ссылки на свои ключи. Если счетчик ссылок ключа *k* в *d* снижается до 0, то элемент *d*[*k*] исчезает. Аргумент `adict` используется для инициализации отображения.

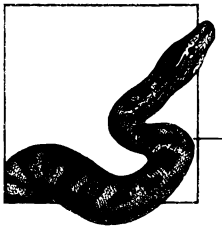
WeakValueDictionary `class WeakValueDictionary(adict={})`

Объект *d* типа `WeakKeyValue` — это отображение, использующее слабые ссылки на свои значения. Если счетчик ссылок значения *v* в *d* снижается до 0, то все элементы, такие, что *d*[*k*] равно *v*, исчезают. Аргумент `adict` используется для инициализации отображения.

Класс `WeakKeyDictionary` обеспечивает недеструктивный способ ассоциирования дополнительных данных с хешируемыми объектами, не сопровождающийся изменением этих объектов. Класс `WeakValueDictionary` обеспечивает недеструктивный способ записи временных ассоциаций между объектами и создание кешей. Во всех этих случаях использование слабого отображения вместо словаря позволяет добиться того, что объект, являющийся во всех остальных отношениях утилизируемым, не будет оставаться в активном состоянии лишь по той причине, что он используется в отображении.

В качестве типичного примера можно привести класс, отслеживающий свои экземпляры, но не оставляющий их в активном состоянии лишь потому, что он отслеживает их.

```
import weakref
class Tracking(object):
    _instances_dict = weakref.WeakValueDictionary()
    def __init__(self):
        Tracking._instances_dict[id(self)] = self
    @classmethod
    def instances(cls): return cls._instances_dict.values()
```



14

Потоки и процессы

Поток — это отдельно выполняемый набор инструкций, использующий глобальное состояние (память) совместно с другими потоками. Создается впечатление, будто потоки выполняются одновременно, хотя обычно они выполняются поочередно на одном процессоре (ядре). Овладеть навыками работы с потоками нелегко, и многопоточные программы трудно поддаются тестированию и отладке. Однако, как говорится во врезке "Что использовать: многопоточное, многопроцессное или асинхронное программирование?", корректное использование многопоточности иногда позволяет повысить быстродействие программы по сравнению с традиционным однопоточным подходом. В этой главе рассматриваются средства, предлагаемые Python для работы с потоками, включая модули `threading`, `queue` и `concurrent`.

Процесс — это экземпляр выполняющейся программы. Операционная система защищает процессы от взаимного влияния. Процессы, нуждающиеся в обмене данными, должны явно организовать такой обмен посредством механизмов *межпроцессного взаимодействия* (IPC). Процессы могут сообщаться посредством файлов (см. главу 10) и баз данных (см. главу 11). Общий способ, с помощью которого процессы могут обмениваться информацией с использованием механизмов хранения данных, таких как файлы или базы данных, состоит в том, что один процесс записывает данные, а другой читает их. В этой главе рассматриваются модули `subprocess` и `multiprocessing` стандартной библиотеки Python, а также части модуля `os`, связанные с процессами, включая простое межпроцессное взаимодействие посредством *каналов*, а также механизм кросс-платформенного межпроцессного взаимодействия, известный как *отображение файлов в памяти*, который предоставляется программам модулем `mmap`.

Сетевые механизмы хорошо приспособлены к IPC, поскольку они работают с процессами, выполняющимися на различных узлах сети, а не на одном и том же узле. Модуль `multiprocessing` предоставляет механизмы, пригодные для осуществления IPC по сети. В главе 17 рассмотрены низкоуровневые сетевые механизмы, обеспечивающие основу для реализации IPC. Высокоуровневые механизмы *распределенных*

вычислений, такие как CORBA, DCOM/COM+, EJB, SOAP, XML-RPC и .NET, могут реализовывать IPC более простыми способами, как локальными, так и удаленными. Распределенные вычисления в этой книге не рассматриваются.

Что использовать: многопоточное, многопроцессное или асинхронное программирование?

Во многих ситуациях наилучшим ответом будет "Ничего из вышеперечисленного". Каждый из этих подходов в лучшем случае может служить лишь целям оптимизации, причем (как обсуждается в разделе "Оптимизация" главы 16) нередко ненужной или преждевременной. Все они подвержены ошибкам, трудно поддающимся тестированию и отладке; по возможности придерживайтесь однопоточного подхода.

Если вам действительно требуется оптимизация и ваша программа интенсивно использует операции ввода-вывода (в том смысле, что *б*ольшая часть времени тратится на выполнение этих операций), то асинхронное программирование (глава 18) обеспечит наилучшее быстродействие, при условии что вы сможете сделать свои операции ввода-вывода неблокирующими. Если же вы никак не можете обойтись без блокирующих операций ввода-вывода, занимающих основную часть времени, то модуль `threading` может помочь вашей программе работать быстрее.

Если ваша программа интенсивно использует процессор (в том смысле, что *б*ольшая часть времени тратится на выполнение вычислений), то в случае CPython модуль `threading`, как правило, не позволит улучшить производительность программы. Причина заключается в том, что CPython использует *глобальную блокировку интерпретатора* (GIL), в связи с чем в каждый конкретный момент времени может выполняться только один поток кода на языке Python (то же относится к PyPy). Однако Jython и Iron-Python не используют GIL. C-расширение может "отменить GIL", выполняя затратную в отношении времени операцию. В частности, библиотека NumPy (раздел "NumPy" в главе 15) прибегает к этому при выполнении операций с массивами. Как следствие, если ваша программа интенсивно задействует процессор посредством вызовов, потребляющих значительное количество процессорного времени, в NumPy или другом аналогичным образом оптимизированном расширении, то модуль `threading` может способствовать повышению быстродействия вашей программы на многопроцессорном или многоядерном компьютере.

Если ваша программа интенсивно задействует процессор, выполняя код, написанный исключительно на языке Python, и вы используете CPython или PyPy на многопроцессорном или многоядерном компьютере, то повысить производительность поможет модуль `multiprocessing`. Особенно это относится к ситуациям, когда у вас есть возможность выполнять программу параллельно на нескольких компьютерах, в идеале — подключенных к быстродействующей сети. Однако в этом случае лучше использовать подходы и пакеты, обсуждаемые на вики-сайте Python (`https://wiki.python.org/moin/ParallelProcessing`), которые в этой книге не рассматриваются.

Потоки в Python

Python предлагает многопоточность на таких платформах, поддерживающих потоки, как Win32, Linux и другие разновидности Unix. Действие называют *атомарным*, если гарантируется, что между началом и завершением его выполнения не произойдет переключения потоков. На практике в CPython операции, которые *кажутся* атомарными (например, операции простого присваивания), большей частью атомарные, если выполняются над встроенными типами (в то же время операции расширенного и многократного присваивания таковыми не являются). Однако полагаться на атомарность операций — в целом не лучшая идея. Вы можете работать с экземпляром пользовательского класса, а не встроенного типа, поэтому попутно могут выполняться неявные вызовы кода на языке Python, делающие несостоятельными предположения об атомарном характере операций. Расчет на атомарность, зависящую от реализации, может привязать ваш код к конкретной реализации, препятствуя последующим обновлениям. Вместо того чтобы полагаться на атомарность, лучше воспользоваться средствами синхронизации, рассмотрению которых посвящена остальная часть главы.

Python предлагает многопоточность двух видов. Прежний модуль `_thread` (называвшийся `thread` в версии v2) содержит низкоуровневую функциональность и не рекомендуется к непосредственному использованию в коде; в данной книге модуль `_thread` не рассматривается. Вместо этого рекомендуется использовать созданный поверх него модуль `threading`. При проектировании многопоточных систем основной проблемой является координация выполнения нескольких потоков. Модуль `threading` предоставляет несколько объектов синхронизации. Альтернативный вариант заключается в использовании модуля `queue`, весьма полезного в плане синхронизации потоков, поскольку он предлагает синхронизированные потокобезопасные типы, удобные для обмена данными между потоками и координации их выполнения. Пакет `concurrent`, который будет рассмотрен вслед за модулем `multiprocessing`, предоставляет унифицированный интерфейс для решения задач межпоточной коммуникации и координации, что может быть реализовано посредством пулов потоков или процессов.

Модуль `threading`

Модуль `threading` содержит функциональность для работы с потоками. Применяемый в нем подход аналогичен тому, который используется в Java, но объекты блокировок и состояний моделируются как отдельные объекты (в Java эта функциональность является частью каждого объекта), а потоками нельзя непосредственно управлять извне (т.е. возможности присваивания приоритетов, группирования, уничтожения или прекращения выполнения потоков отсутствуют). Все предоставляемые модулем `threading` объекты атомарные.

Модуль threading содержит следующие классы, предназначенные для работы с потоками: Thread, Condition, Event, Lock, RLock, Semaphore, BoundedSemaphore и Timer (а также, только в версии v3, класс Barrier). Он также предоставляет функции, описание которых приводится ниже.

active_count `active_count()`

Возвращает целое число, представляющее количество объектов Thread, активных в настоящее время (но не тех, выполнение которых закончилось или не было начато).

current_thread `current_thread()`

Возвращает количество объектов Thread для вызывающего потока. Если вызывающий поток не был создан модулем threading, то current_thread создает и возвращает полуфиктивный объект Thread с ограниченной функциональностью.

enumerate `enumerate()`

Возвращает список всех объектов Thread, активных в настоящее время (но не тех, выполнение которых закончилось или не было начато).

stack_size `stack_size([size])`

Возвращает размер стека (в байтах), используемого для новых потоков; 0 означает "системное значение по умолчанию". Передаваемое в необязательном аргументе *size* значение будет использоваться для вновь создаваемых потоков (на платформах, которые поддерживают установку размера стека потоков). На допустимые значения этого аргумента налагаются ограничения, зависящие от платформы: он может быть равным 32768 (или еще более высокому минимальному значению на некоторых платформах) и (на некоторых платформах) быть кратным 4096. Передача значения 0 всегда допустима и означает "использовать системное значение по умолчанию". В случае передачи значения аргумента *size*, недопустимого для текущей платформы, функция stack_size возбуждает исключение ValueError.

Объекты Thread

Экземпляр *t* класса Thread моделирует поток. Можно либо передать объекту *t* при его создании функцию, которая будет использоваться в качестве основной функции данного объекта, либо создать подкласс Thread и переопределить метод run (также можно переопределить метод __init__, но при этом не следует переопределять другие методы). Вновь созданный объект *t* не готов к немедленному выполнению; чтобы подготовить его к этому (активизировать), следует вызвать метод *t*.start(). Ставший активным объект *t* прекращает выполнение, как только завершается выполнение его основной функции (будь-то обычным способом или в результате распространения исключения). Объект *t* класса Thread может быть *демоном*, т.е. выполнение программы на языке Python может завершиться, даже если объект *t* все еще остается активным, в то время как обычные (не являющиеся демонами) потоки не

дают программе завершиться, пока все они не завершатся. Описание конструктора, методов и свойств, предоставляемых классом Thread, приведено ниже.

Thread `class Thread(`*`name`*`=None, `*`target`*`=None, `*`args`*`=(), `*`kwargs`*`={})`

Всегда вызывайте Thread с именованными аргументами: количество и порядок задания параметров в будущем могут измениться, но гарантируется, что имена параметров останутся неизменными. Если вы инициализируете сам класс Thread, передайте аргумент *target*: метод `t.run` вызывает функцию `target(*args,**kwargs)`. Если вы расширяете класс Thread и переопределяете метод run, то передавать аргумент *target* не следует. В любом случае выполнение потока не начнется до тех пор, пока вы не вызовете метод `t.start()`. Аргумент *name* определяет имя потока *t*. Если оно задано равным None, то Thread генерирует уникальное имя для *t*. Если подкласс *T* класса Thread переопределяет метод `__init__`, то метод *T*.`__init__` должен вызвать метод `Thread.__init__` для объекта `self`, прежде чем вызвать любой другой метод Thread.

daemon `t.daemon`

daemon — это свойство, принимающее значение True, если *t* — демон (т.е. если процесс может завершиться, даже когда объект *t* все еще остается активным; при этом завершается также поток *t*); в противном случае daemon принимает значение False. Первоначально *t* является демоном тогда и только тогда, когда создавший его поток является демоном. Вы можете установить значение `t.daemon` только до вызова метода `t.start`. Объект *t* устанавливается в качестве потока-демона при присваивании значения свойству `t.daemon` тогда и только тогда, когда присваиваемым значением является True.

is_alive `t.is_alive()`

Возвращает значение True, если объект *t* активен (т.е. если метод `t.start` был выполнен, но выполнение метода `t.run` еще не завершилось). В противном случае метод `is_alive` возвращает значение False.

name `t.name`

Аргумент *name* — это свойство, возвращающее имя объекта *t*. Присваивание ему значения повторно привязывает имя объекта *t* (имя *name* предназначено для упрощения отладки; оно не обязано быть уникальным среди потоков).

join `t.join(`*`timeout`*`=None)`

Приостанавливает вызывающий поток (который не должен быть объектом *t*) до завершения *t* (если *t* уже завершился, то вызывающий поток не приостанавливается). Аргумент *timeout* обсуждается в разделе "Параметры timeout". Метод `t.join` можно вызывать только после вызова метода `t.start`. Вы можете вызвать метод `t.join` более одного раза.

run `t.run()`

Метод run — это метод, который выполняет основную функцию объекта *t*. Подклассы Thread могут переопределять метод run. Если метод run не переопределен, он вызывает объект *target*, переданный конструктору при создании объекта *t*. Не вызывайте метод `t.run` непосредственно — это задача метода `t.start`.

start `t.start()`

Метод `start` активизирует объект `t` и организует выполнение метода `t.run` в отдельном потоке. Вы должны вызывать метод `t.start` только один раз для любого заданного объекта потока `t`. При повторном вызове этого метода он возбудит исключение.

Объекты синхронизации потоков

Модуль `threading` предоставляет несколько примитивов синхронизации — типов, которые позволяют потокам обмениваться сообщениями и координировать свою работу. Каждый примитивный тип имеет свою специфическую сферу применения.

Примитивы синхронизации потоков могут вам не понадобиться

Если вы обходитесь без глобальных переменных, значения которых изменяются и к которым получают доступ несколько потоков, то модуль `queue` (раздел "Модуль `queue`"), равно как и модуль `concurrent` (раздел "Модуль `concurrent.futures`"), часто может обеспечить требуемую координацию работы потоков. В разделе "Архитектура многопоточной программы" показано, как объекты `Queue` позволяют создавать многопоточные программы с простой и эффективной архитектурой без явного использования примитивов синхронизации.

Параметры `timeout`

Примитивы синхронизации `Condition` и `Event` предоставляют методы `wait`, которые принимают необязательный аргумент *timeout*. Метод `join` объекта `Thread` также принимает необязательный аргумент `timeout`, равно как и методы `acquire` объектов блокировки. Значению этого аргумента `None` (значение по умолчанию) соответствует обычное поведение блокировки (выполнение вызывающего потока прекращается до тех пор, пока не выполнится требуемое условие). Во всех остальных случаях он принимает значение в виде числа с плавающей точкой, означающего предельную длительность ожидания в секундах (это число может иметь дробную часть, поэтому длительность определяемого им временного интервала может быть любой, даже очень малой). По истечении интервала ожидания (тайм-аута) вызывающий поток возвращается в состояние выполнения, даже если требуемое условие не выполнено. В этом случае ожидающий метод возвращает значение `False` (иначе метод возвращает значение `True`). Аргумент *timeout* позволяет проектировать системы, способные справляться со случайными аномалиями в некоторых потоках, повышая робастность системы. Однако использование тайм-аута может замедлить выполнение программы: если этот фактор имеет существенное значение, проведите соответствующие измерения быстродействия программы.

Объекты Lock и RLock

Классы блокировок Lock и RLock предоставляют одни и те же три метода. Описание сигнатур и семантики этих методов на примере экземпляра *L* объекта Lock приведено ниже.

acquire *L*.acquire(*blocking*=True)

Если аргумент *blocking* равен True, метод acquire блокирует объект *L*. Если *L* уже заблокирован, то вызывающий поток приостанавливается и ожидает, пока *L* не разблокируется, после чего блокирует *L*. Даже если вызывающий поток является тем же потоком, который в последний раз заблокировал *L*, он все равно приостанавливается и ожидает, пока другой поток не освободит *L*. Если аргумент *blocking* равен False и *L* разблокирован, то метод acquire блокирует *L* и возвращает значение True. Если аргумент *blocking* равен False и *L* заблокирован, то метод acquire не оказывает никакого воздействия на объект *L* и возвращает значение False.

locked *L*.locked()

Возвращает значение True, если *L* заблокирован; в противном случае возвращает значение False.

release *L*.release()

Разблокирует объект *L*, который должен быть заблокированным. Если *L* заблокирован, то метод *L*.release может быть вызван любым потоком, а не только тем, который заблокировал *L*. Если на *L* блокируются несколько потоков (т.е. каждый из них вызвал метод *L*.acquire, обнаружил, что объект *L* заблокирован, и ожидает его разблокирования), то метод release пробуждает произвольный поток. Поток, вызвавший метод release, не приостанавливается: он остается в состоянии готовности и продолжает выполнение.

Часто более удобной оказывается семантика объекта *r* класса RLock (особенно в случае специфических архитектур, в которых требуется, чтобы одни потоки могли освобождать блокировки, захваченные другими потоками). RLock — это *реентерабельная блокировка*, т.е. такая, что если объект *r* заблокирован, то он запоминает, какой поток является его *потоком-владельцем* (т.е. какой поток заблокировал его и, в случае класса RLock, является единственным потоком, который может его разблокировать). Повторный вызов метода *r*.acquire потоком-владельцем не сопровождается блокированием; в этом случае всего лишь инкрементируется внутренний счетчик объекта *r*. В аналогичной ситуации с объектом Lock поток блокировался бы до освобождения блокировки другим потоком. Рассмотрим, например, следующий фрагмент кода.

```
lock = threading.RLock()
global_state = []
def recursive_function(some, args):
    with lock: # захватывает блокировку, гарантируя ее
               # освобождение при завершении
    ...modify global_state...
```

```
    if more_changes_needed(global_state):
        recursive_function(other, args)
```

Если бы переменная *lock* была экземпляром `threading.Lock`, то функция `recursive_function` заблокировала бы свой вызывающий поток, рекурсивно вызывая саму себя: инструкция `with`, обнаружив, что блокировка уже захвачена (даже если это было сделано тем же потоком), должна была бы блокироваться и ждать... Если используется блокировка `threading.RLock`, подобная проблема не возникает: в этом случае, поскольку блокировка уже захвачена *тем же потоком*, она всего лишь инкрементирует свой внутренний счетчик при ее повторном захвате, и выполнение продолжится.

Объект *r* класса `RLock` разблокируется, только если он освобождается столько раз, сколько был захвачен. Тип `RLock` полезен, если необходимо гарантировать исключительный доступ к объекту в условиях, когда одни методы объекта вызывают другие. Каждый из методов может захватить в начале выполнения и освободить при завершении один и тот же экземпляр `RLock`. Одним из способов обеспечить гарантированное освобождение блокировки является использование инструкции `try/finally` (раздел "Инструкция `try/finally`" в главе 5). Обычно лучше использовать инструкцию `with` (раздел "Инструкция `with`" в главе 3): все блокировки, объекты состояния и семафоры — менеджеры контекста, поэтому для получения экземпляра любого из этих типов (неявно — с блокированием) можно использовать его непосредственно в инструкции `with`, и он будет гарантированно освобожден при выходе из блока `with`.

Объекты `Condition`

Объект состояния *c* класса `Condition` обертывает объект *L* класса `Lock` или `RLock`. Описание конструктора и методов, предоставляемых классом `Condition`, приведено ниже.

`Condition`

`class Condition(`*lock*`=None)`

Создает и возвращает новый объект *c* класса `Condition` с объектом блокировки *L*, передаваемым с помощью аргумента *lock*. Если аргумент *lock* равен `None`, то *L* устанавливается во вновь создаваемый объект `RLock`.

`acquire`, `release`

c`.acquire(`*blocking*`=1)`

c`.release()`

Эти методы всего лишь вызывают соответствующие методы *L*. Поток не должен вызывать для *c* никакой другой метод, если он не захватил блокировку *L*.

`notify`, `notify_all`

c`.notify()`
c`.notify_all()`

Метод `notify` пробуждает один из потоков, ожидающих изменения состояния объекта *c*. Вызывающий поток должен владеть блокировкой *L*, прежде чем вызывать метод *c*`.notify()`, и метод `notify` не освобождает *L*. Пробужденный поток не переходит

в состояние готовности к немедленному выполнению до тех пор, пока не сможет вновь захватить блокировку *L*. Поэтому обычно вызывающий поток вызывает метод `release` после вызова метода `notify`. Метод `notify_all` аналогичен методу `notify`, но пробуждает все ожидающие потоки, а не только один.

`wait` `c.wait(`*timeout*`=None)`

Метод `wait` освобождает блокировку *L*, а затем приостанавливает вызывающий поток до тех пор, пока другой поток не вызовет для объекта *c* метод `notify` или `notify_all`. Вызывающий поток должен владеть блокировкой *L*, прежде чем вызывать метод `c.wait()`. Аргумент *timeout* рассмотрен в разделе "Параметры `timeout`". После пробуждения потока, будь то в результате вызова метода `notify` или по истечении тайм-аута, он готов к выполнению, когда вновь захватывает блокировку *L*. Если метод `wait` возвращает значение `True` (означающее нормальный выход, а не по тайм-ауту), то вызывающий поток всегда вновь захватывает блокировку *L*.

Обычно объект *c* класса `Condition` регулирует доступ к некоторому глобальному состоянию *s*, совместно используемому потоками. Если поток должен ждать изменения состояния *s*, то он должен выполнять цикл.

```
with c:
    while not is_ok_state(s):
        c.wait()
    do_some_work_using_state(s)
```

В то же время каждый поток, изменяющий *s*, вызывает метод `notify` (или `notify_all`, если он должен пробудить все ожидающие потоки, а не только один) при каждом изменении *s*.

```
with c:
    do_something_that_modifies_state(s)
    c.notify() # or, c.notify_all()
# в вызове c.release() нет необходимости; это делается
# автоматически при выходе из блока 'with'
```

Объект *c* всегда должен захватываться и освобождаться при каждом использовании методов *c*: выполнение этого требования с помощью инструкции `with` уменьшает вероятность возникновения ошибок при использовании экземпляров `Condition`.

Объекты `Event`

Объекты событий класса `Event` позволяют приостанавливать и переводить в состояние ожидания любое количество потоков. Все потоки, ожидающие установки внутреннего флага объекта *e* класса `Event`, переходят в состояние готовности, как только любой другой поток вызовет метод `e.set()`. Объект события *e* имеет флаг, индицирующий наступление события; его начальным значением при создании объекта *e* является `False`. Следовательно, объект `Event` похож на упрощенный объект

Condition. Объекты Event удобно использовать в качестве сигналов одноразовых изменений, но они слишком хрупкие для более общих применений. В частности, вызовы метода e.clear() подвержены ошибкам. Класс Event предоставляет следующие методы.

Event `class Event()`

Создает и возвращает новый объект *e* класса Event, флаг которого установлен в False.

clear `e.clear()`

Устанавливает флаг объекта *e* в False.

is_set `e.is_set()`

Возвращает значение флага объекта *e*, True или False.

set `e.set()`

Устанавливает флаг объекта *e* в True. Все потоки, ожидающие установки флага объекта *e*, если таковые имеются, переходят в состояние готовности к выполнению.

wait `e.wait(`*timeout*`=None)`

Если флаг объекта *e* имеет значение True, то метод wait немедленно возвращает управление. В противном случае он приостанавливает работу вызывающего потока до тех пор, пока другой поток не вызовет метод set. Аргумент *timeout* рассмотрен в разделе "Параметры timeout".

Объекты Semaphore

Семафоры (также известные как *семафоры со счетчиками*) — это обобщение блокировок. Состояние объекта класса Lock характеризуется значениями True и False. Состояние объекта *s* класса Semaphore — это число (счетчик), которое может принимать значения от 0 до некоторого значения *n*, устанавливаемого при создании *s*, включительно. Семафоры могут использоваться для управления фиксированным пулом ресурсов (например, 4 принтера или 20 сокетов), хотя часто для этих целей надежнее использовать объекты класса Queue. Класс BoundedSemaphore обладает сходными свойствами, но возбуждает исключение ValueError, если состояние превышает начальное значение: во многих случаях такое поведение может служить полезным индикатором наличия ошибки в коде.

Semaphore, BoundedSemaphore
`class Semaphore(`*n*`=1)`
`class BoundedSemaphore(`*n*`=1)`

Вызов Semaphore создает и возвращает объект семафора *s* с начальным значением счетчика, равным *n*. Ограниченный семафор BoundedSemaphore аналогичен семафору Semaphore, за исключением того, что метод *s*.release() возбуждает исключение ValueError, если значение счетчика становится больше *n*. Объект семафора *s* предоставляет следующие методы.

acquire `s.acquire(`*blocking*`=True)`

Если значение счетчика объекта *s* больше нуля, метод `acquire` уменьшает значение счетчика на *1* и возвращает значение `True`. Если значение счетчика равно *0* и аргумент *blocking* равен `True`, метод `acquire` приостанавливает вызывающий поток и ожидает, пока другой поток не вызовет метод `s.release`. Если счетчик объекта *s* равен *0* и аргумент *blocking* равен *False*, метод `acquire` немедленно возвращает значение `False`.

release `s.release()`

Если значение счетчика объекта *s* больше нуля, но ни один объект не ожидает изменения состояния объекта *s*, то метод `release` увеличивает значение счетчика на `1`. Если значение счетчика равно `0` и некоторые потоки ожидают изменения состояния объекта *s*, то метод `release` оставляет значение счетчика равным `0` и пробуждает один из ожидающих потоков, выбранный произвольно. Поток, вызвавший метод `release`, не приостанавливается; он остается в состоянии готовности и продолжает выполнение, как обычно.

Объекты `Timer`

Объект класса `Timer` вызывает указанный объект во вновь создаваемом потоке через определенное время. Описание конструктора и методов, предоставляемых классом `Timer`, приведено ниже.

Timer `class Timer(`*interval*`, `*callable*`, `*args*`=(), `*kwargs*`={})`

Создает объект *t*, который вызывает объект *callable* спустя *interval* секунд после запуска (аргумент *interval* — это число с плавающей точкой, которое может иметь дробную часть).

cancel `cancel()`, `t.cancel()`

Останавливает таймер и отменяет выполнение заданного действия, если объект *t* все еще находится в состоянии ожидания (еще не вызвал свой вызываемый объект), когда вы вызываете метод `cancel`.

start `start()`, `t.start()`

Запускает таймер *t*.

Класс `Timer` расширяет класс `Thread`, добавляя атрибуты *function*, *interval*, *args* и *kwargs*.

Объект `Timer` является "одноразовым" — *t* вызывает свой объект только один раз. Ниже приведен простой способ организации периодического вызова объекта через каждые *interval* секунд.

```
class Periodic(threading.Timer):
  def __init__(self, interval, callable, args=(), kwargs={}):
    self.callable = callable
    threading.Timer.__init__(self, interval, self._f, args,
                             kwargs)
  def _f(self, *args, **kwargs):
    Periodic(self.interval, self.callable, args, kwargs).start()
    self.callable(*args, **kwargs)
```

Объекты `Barrier` (только в версии v3)

Класс `Barrier` — это примитив синхронизации (барьер выполнения), позволяющий организовать ожидание для определенного количества потоков, пока все они не достигнут некоторой точки в ходе своего выполнения, прежде чем каждый из них сможет возобновить работу. В частности, если поток вызывает метод `b.wait()`, он блокируется до тех пор, пока тот же вызов для объекта *b* не будет выполнен заданным количеством потоков. Когда это происходит, все потоки, блокируемые объектом *b*, освобождаются.

Описание конструктора, методов и свойств, предоставляемых классом `Barrier`, приведено ниже.

`Barrier` `class Barrier(num_threads, action=None, timeout=None)`

Аргумент *`action`* — вызываемый объект, не имеющий аргументов: если он передан, то вызывается в одном из заблокированных потоков, когда все они разблокируются. Аргумент *`timeout`* рассмотрен в разделе "Параметры `timeout`".

`abort` `abort()`

Вызов метода `b.abort()` переводит объект *b* класса `Barrier` в сброшенное (broken) состояние в том смысле, что любой поток, ожидающий синхронизации посредством объекта *b*, возбуждает исключение `threading.BrokenBarrierException` (это же исключение возбуждается и при любом последующем вызове `b.wait()`). Обычно метод `abort()` используется в ситуациях, когда некоторому ожидающему потоку необходимо преждевременно прекратить выполнение во избежание блокирования всей программы.

`broken` `broken`

Возвращает значение `True`, если *b* находится в сброшенном состоянии; в противном случае возвращает значение `False`.

`n_waiting` `n_waiting`

Количество потоков, ожидающих синхронизации посредством объекта *b*.

`parties` `parties`

Значение, переданное конструктору *b* в качестве аргумента *`num_threads`*.

reset `reset()`

Возвращает объект *b* в начальное пустое (сброшенное) состояние. Однако любой активный поток, ожидающий синхронизации посредством объекта *b*, возобновляет работу, возбуждая при этом исключение `threading.BrokenBarrierException`.

wait `wait()`

Первые `b.parties-1` потоков, вызывающих метод `b.wait()`, блокируются. Если количество потоков, блокируемых объектом *b*, равно `b.parties-1` и один дополнительный поток вызывает метод `b.wait()`, то все блокированные потоки возобновляют работу. Вызов `b.wait()` возвращает целые числа, разные для каждого возобновляемого потока и принадлежащие диапазону `range(b.parties)`, в произвольном порядке. Потоки могут использовать это возвращаемое значение для определения своих дальнейших действий (хотя для этого во многих случаях достаточно передать аргумент *action* конструктору `Barrier`, что гораздо проще).

Класс `threading.Barrier` существует лишь в версии v3. В версии v2 вы можете реализовать его самостоятельно, как показано в примере на сайте Stack Overflow (`https://stackoverflow.com/questions/26622745/implementing-barrier-in-python2-7`).

Локальное хранилище потока

Модуль `threading` предоставляет класс `local`, который поток может использовать для получения *локального хранилища потока* (thread-local storage — TLS), также называемого *данными потока*. Экземпляр *L* класса `local` может иметь произвольные именованные атрибуты, которые разрешается устанавливать и получать, и сохраняет их в словаре `L.__dict__`, к которому вы также можете получать доступ. Объект *L* полностью потокобезопасен, т.е. при попытке установки и получения атрибутов *L* одновременно несколькими потоками никаких проблем возникать не будет. Что самое важное, каждый из потоков, обращающихся к *L*, видит набор разрозненных атрибутов, и изменения, внесенные в одном потоке, не оказывают влияния на другие потоки. Соответствующий пример приведен ниже.

```
import threading
L = threading.local()
print('in main thread, setting zop to 42')
L.zop = 42
def targ():
  print('in subthread, setting zop to 23')
  L.zop = 23
  print('in subthread, zop is now ', L.zop)
t = threading.Thread(target=targ)
t.start()
t.join()
print('in main thread, zop is now ', L.zop)
```

```
# вывод:
# in main thread, setting zop to 42
# in subthread, setting zop to 23
# in subthread, zop is now 23
# in main thread, zop is now 42
```

TLS упрощает написание кода, предназначенного для выполнения несколькими потоками, поскольку вы можете использовать одно и то же пространство имен (экземпляр `threading.local`) в нескольких потоках, не опасаясь того, что потоки будут мешать друг другу.

Модуль queue

Модуль `queue` (`Queue` в версии v2) предоставляет различные типы очередей, поддерживающие возможность доступа из множества потоков, с одним основным классом, двумя подклассами и двумя классами исключений.

`Queue` `class Queue(maxsize=0)`

`Queue`, основной класс модуля `queue`, реализует очередь, работающую по принципу "первым пришел, первым ушел" (FIFO — каждый раз извлекается элемент, который был добавлен раньше других). Соответствующие методы обсуждаются в разделе "Методы экземпляров `Queue`". Если аргумент *`maxsize`*>0, то новый экземпляр *`q`* класса `Queue` считается полностью заполненным, когда *`q`* имеет *`maxsize`* элементов. Поток, вставляющий элемент с использованием опции *`block`*, когда очередь *`q`* полностью заполнена, приостанавливается до тех пор, пока другой поток не извлечет элемент из очереди. Если *`maxsize`*<=0, то очередь *`q`* никогда не считается заполненной, и ее размер ограничен лишь доступным объемом памяти, как в случае обычных контейнеров Python.

`LifoQueue` `class LifoQueue(maxsize=0)`

`LifoQueue` — это подкласс `Queue`. Единственным различием между ними является то, что очередь типа `LifoQueue` работает по принципу "последним пришел, первым ушел" (LIFO), т.е. каждый раз извлекается элемент, который был добавлен позже других.

`PriorityQueue` `class PriorityQueue(maxsize=0)`

`PriorityQueue` — это подкласс `Queue`. Единственным различием между ними является то, что `PriorityQueue` реализует *очередь с приоритетом* — каждый раз извлекается элемент, имеющий наименьшее значение в очереди. Поскольку в данном случае отсутствуют соотношения типа *`key=argument`*, то обычно в качестве элементов очереди используют пары (*`priority, payload`*), причем значения с более низким значением приоритета извлекаются раньше других.

Empty — `Empty` — это исключение, которое возбуждается при вызове метода `q.get(False)`, если очередь `q` пустая.

Full — `Full` — это исключение, которое возбуждается при вызове метода `q.put(x, False)`, если очередь `q` заполнена.

Методы экземпляров Queue

Экземпляр `q` класса `Queue` (или любого его подкласса) предоставляет следующие потокобезопасные методы, атомарность которых гарантируется.

empty `q.empty()`

Возвращает значение `True`, если очередь `q` пустая; в противном случае возвращает значение `False`.

full `q.full()`

Возвращает значение `True`, если очередь `q` заполнена; в противном случае возвращает значение `False`.

get, get_nowait `q.get(block=True, timeout=None)`

Если аргумент `block` равен `False`, то метод `get` удаляет элемент из очереди `q` и возвращает его, если в очереди имеются элементы; в противном случае возбуждается исключение `Empty`. Если аргумент `block` равен `True`, а аргумент `timeout` — `None`, то метод `get` удаляет элемент из очереди `q` и возвращает его, приостанавливая вызывающий поток в случае необходимости до тех пор, пока элемент не станет доступным. Если аргумент `block` равен `True`, а аргумент `timeout` не равен `None` (в этом случае он должен иметь значение в виде неотрицательного числа, которое может включать дробную часть, представляющую доли секунды), то метод `get` ожидает не более чем `timeout` секунд (если по истечении тайм-аута доступные элементы все еще отсутствуют, возбуждается исключение `Empty`). Вызов `q.get_nowait()` аналогичен вызову `q.get(False)`, который, в свою очередь, аналогичен вызову `q.get(timeout=0.0)`. Метод `get` удаляет элементы из очереди и возвращает их в том же порядке, в каком они помещались в очередь с помощью метода `put` (FIFO), если `q` — непосредственный экземпляр класса `Queue`. Если же `q` — экземпляр класса `LifoQueue`, то элементы удаляются из очереди и возвращаются в соответствии с принципом LIFO, а если `q` — экземпляр класса `PriorityQueue`, то извлекается и возвращается наименьший из элементов очереди.

put, put_nowait `q.put(item, block=True, timeout=None)`

Если аргумент `block` равен `False`, то метод `put` добавляет элемент `item` в очередь `q` при условии, что она не заполнена; в противном случае метод `put` возбуждает исключение `Full`. Если аргумент `block` равен `True`, а аргумент `timeout` равен `None`, то метод `put` добавляет элемент `item` в очередь `q`, в случае необходимости

приостанавливая вызывающий поток до тех пор, пока в очереди не появится место для нового элемента. Если аргумент *block* равен `True`, а аргумент *timeout* не равен `None` (в этом случае он должен иметь значение в виде неотрицательного числа, которое может включать дробную часть, представляющую доли секунды), то метод `put` ожидает не более чем *timeout* секунд (если по истечении тайм-аута очередь *q* все еще остается заполненной, то метод `put` возбуждает исключение `Full`). Вызов `q.put_nowait(item)` аналогичен вызову `q.put(item, False)`, который, в свою очередь, аналогичен вызову `q.put(item, timeout=0.0)`.

qsize `q.qsize()`

Возвращает текущее количество элементов в очереди *q*.

Кроме того, объект *q* поддерживает внутренний скрытый счетчик незавершенных заданий, начальным значением которого является нуль. Каждый вызов метода `get` увеличивает значение счетчика на единицу. Когда рабочий поток завершает обработку задания, значение счетчика уменьшается на единицу за счет вызова метода `q.task_done()`. Для синхронизации с событием "все задания выполнены" следует выполнить вызов `q.join()`: метод `join` продолжает выполнение вызывающего потока, когда счетчик незавершенных заданий обнуляется. Если значение счетчика не равно нулю, то метод `q.join()` блокирует вызывающий поток и снимает блокировку впоследствии, когда счетчик обнулится.

Вы не обязаны использовать методы `join` и `task_done`, если предпочитаете координировать работу потоков другими способами, но эти методы обеспечивают простой и полезный подход к координации работы потоков с использованием класса `Queue`.

Класс `Queue` может служить неплохим примером реализации идиомы EAFP ("Проще просить прощение, чем получить разрешение"), которая обсуждалась в разделе "Стратегии контроля ошибок" главы 5. Вследствие многопоточности результаты, возвращаемые каждым из немутирующих методов объекта *q* (`empty`, `full`, `qsize`), могут быть лишь ориентировочными. Если другой поток изменит объект *q*, то за промежуток времени между моментом получения потоком информации от немутирующего метода и до момента, когда поток воздействует на эту информацию, ситуация может измениться. Поэтому не стоит возлагать надежды на идиому LBYL ("Хорошенько осмотрись, прежде чем прыгать"), а возня с блокировками для исправления ситуации в основном приведет к бесполезной трате усилий. Просто старайтесь избегать хрупкого LBYL-кода и вместо, например, следующего кода:

```
if q.empty():
    print('no work to perform')
else:
    x = q.get_nowait()
    work_on(x)
```

используйте более простой и надежный подход EAFP:

```
try:
    x = q.get_nowait()
except queue.Empty:
    print('no work to perform')
else:
    work_on(x)
```

Модуль multiprocessing

Модуль multiprocessing содержит функции и классы, с помощью которых можно писать код так, как если бы он был основан на использовании модуля threading, но с распределением работы между процессами, а не потоками. В модуле multiprocessing определены класс Process (аналогичный классу threading.Thread) и классы примитивов синхронизации (BoundedSemaphore, Condition, Event, Lock, RLock, Semaphore и, только в версии v3, Barrier, каждый из которых аналогичен одноименному классу из модуля threading, а также классы Queue и JoinableQueue, аналогичные классу queue.Queue). Эти классы упрощают переделку кода, написанного под модуль threading, в версию, приспособленную для использования модуля multiprocessing. Нужно только уделить внимание различиям, которые анализируются в разделе "Различия между модулями Multiprocessing и Threading".

Обычно лучше избегать разделения состояния между процессами: вместо этого используйте обмен сообщениями между процессами с помощью очередей. Однако для тех редких ситуаций, в которых вам действительно требуется использование разделяемого состояния, модуль multiprocessing предлагает классы, обеспечивающие доступ к разделяемой памяти (Value и Array), а также более гибкий (включающий координацию работы различных компьютеров в сети), хотя и требующий бо́льших накладных расходов подход, основанный на использовании подкласса Process — класса Manager, предназначенного для хранения произвольных данных и позволяющий другим процессам манипулировать этими данными посредством прокси-объектов. Разделение состояний обсуждается в разделе "Разделение состояний: классы Value, Array и Manager".

Если вы пишете новый многозадачный код, а не портируете код, первоначально написанный как многопоточный, то часто можно воспользоваться другим подходом, предлагаемым модулем multiprocessing. В частности, во многих случаях удается упростить код, используя класс Pool, который обсуждается в разделе "Пул процессов".

Существуют также другие эффективные подходы, основанные на использовании объектов Connection, создаваемых с помощью функции-фабрики Pipe или обернутых объектами Client и Listener. Несмотря на то что эти подходы обеспечивают бо́льшую гибкость, их применение может характеризоваться повышенной сложностью, и в данной книге они не рассматриваются. Для получения более подробной информации, касающейся модуля multiprocessing, следует обратиться к онлайн-документации

(https://docs.python.org/3/library/multiprocessing.html) и сторонним онлайн-руководствам (https://pymotw.com/2/multiprocessing/index.html).

Различия между модулями Multiprocessing и Threading

Код, первоначально написанный в расчете на использование модуля threading, нетрудно переписать так, чтобы вместо модуля threading использовался модуль multiprocessing, однако при этом необходимо учитывать различия между модулями.

Структурные различия

Все объекты, которыми обмениваются процессы (например, посредством очереди или аргумента функции *target* конструктора экземпляра Process), сериализуются с помощью модуля pickle (см. раздел "Модули pickle и cPickle" в главе 11). Следовательно, такой обмен возможен лишь для объектов, которые могут быть сериализованы подобным образом. Кроме того, размер сериализованной байтовой строки не может превышать примерно 32 Мбайт (в зависимости от платформы), иначе будет возбуждено исключение. Поэтому на размер объектов, которыми могут обмениваться процессы, налагаются определенные ограничения.

В частности, в Windows дочерние процессы должны быть в состоянии импортировать в качестве модуля основной сценарий, который их породил. Поэтому вы обязательно должны обеспечивать защиту всего высокоуровневого кода основного сценария (в том смысле, что он не должен вновь выполняться дочерними процессами) с помощью обычной идиомы if __name__ == '__main__' (см. раздел "Основная программа" в главе 6).

В случае принудительного завершения процесса (например, посредством сигнала) в то время, когда он использует очередь или примитив синхронизации, он не сможет надлежащим образом очистить ресурсы, связанные с данной очередью или примитивом. В результате этого состояние очереди или примитива может разрушиться, что приведет к возникновению ошибок в других процессах, пытающихся использовать эти объекты.

Класс Process

Класс multiprocessing.Process весьма напоминает класс threading.Thread, но дополнительно предлагает следующие атрибуты и методы.

authkey authkey

Ключ авторизации процесса в виде байтовой строки, который инициализируется случайным числом, предоставляемым функцией os.urandom(). Впоследствии его можно изменить, если в этом возникнет необходимость. Поскольку ключ авторизации используется в более сложных случаях применения для проверки подлинности посредством процедуры рукопожатия, в данной книге он не рассматривается.

exitcode `exitcode`

Имеет значение `None`, если процесс еще не завершился. В противном случае содержит код завершения процесса: целое число, равное `0` в случае успешного завершения, `>0` — в случае аварийного завершения, `<0` — если выполнение процесса было принудительно завершено.

pid `pid`

Имеет значение `None`, если выполнение процесса еще не началось. В противном случае содержит идентификатор процесса, установленный операционной системой.

terminate `terminate()`

Принудительно завершает процесс без предоставления ему возможности выполнить завершающий код, например освободить ресурсы очередей и примитивов синхронизации. Остерегайтесь возможного возникновения ошибок, если процесс использует очереди или примитивы синхронизации.

Различия в очередях

Класс `multiprocessing.Queue` очень напоминает класс `queue.Queue`, за исключением того, что экземпляр `q` класса `multiprocessing.Queue` *не* предоставляет методы `join` и `task_done`. Если методы `q` возбуждают исключения вследствие истечения тайм-аутов, они возбуждают экземпляры исключений `queue.Empty` или `queue.Full`. В модуле `multiprocessing` отсутствуют классы, эквивалентные классам очередей `LifoQueue` и `PriorityQueue`.

Класс `multiprocessing.JoinableQueue` предоставляет методы `join` и `task_done`, которые, однако, имеют семантические отличия по сравнению с их аналогами из класса `queue.Queue`: в случае экземпляра `q` класса `multiprocessing.JoinableQueue` процесс, вызывающий метод `q.get`, *должен* вызвать метод `q.task_done`, когда он завершает обработку данной порции работы (в отличие от необязательной возможности сделать это, как в случае использования класса `queue.Queue`).

Все объекты, которые вы помещаете в очередь модуля `multiprocessing`, должны допускать сериализацию с помощью модуля `pickle`. Между моментом выполнения вызова метода `q.put` и моментом, когда объект станет доступным для получения с помощью метода `q.get`, может возникнуть небольшая задержка. Наконец, не забывайте о том, что преждевременное (вследствие краха или получения сигнала) завершение процесса, использующего объект `q`, может оставить `q` в состоянии, делающем невозможным его использование другими процессами.

Различия в примитивах синхронизации

В модуле `multiprocessing` метод `acquire` примитивов синхронизации классов `BoundedSemaphore`, `Lock`, `RLock` и `Semaphore` имеет сигнатуру `acquire(`*`block`*`=True, `*`timeout`*`=None)`. Семантика аргумента *`timeout`* рассмотрена в разделе "Параметры `timeout`".

Разделение состояний: классы `Value`, `Array` и `Manager`

Чтобы обеспечить использование разделяемой памяти для хранения одиночного примитивного значения, доступного для нескольких процессов, модуль `multiprocessing` предоставляет класс `Value`. Хранение примитивных значений в массивах фиксированного размера обеспечивается классом `Array`. Для обеспечения большей гибкости (включая возможность хранения непримитивных значений и их совместного использования различными системами, соединенными сетью, но не имеющими разделяемой памяти) за счет увеличения накладных расходов модуль `multiprocessing` предоставляет класс `Manager`, являющийся подклассом класса `Process`.

Класс `Value`

Конструктор класса `Value` имеет следующую сигнатуру.

`Value` `Value(`*`typecode`*`, *`*`args`*`, `*`lock`*`=True)`

Аргумент *`typecode`* — это строка, определяющая примитивный тип значения, как в случае модуля `array`, описанного в табл. 15.3. (Кроме того, *`typecode`* может определять также тип из модуля `ctypes`, рассмотренного в разделе "Модуль `ctypes`" главы 24, но необходимость в этом возникает лишь в редких случаях.) Аргумент *`args`* передается конструктору типа: следовательно, *`args`* либо отсутствует (и в этом случае примитив инициализируется значением, используемым по умолчанию и обычно равным `0`), либо является одиночным значением, которое используется для инициализации примитива.

Если аргумент *`lock`* задан равным `True` (значение по умолчанию), то конструктор `Value` создает и использует новую блокировку для защиты экземпляра. Однако в аргументе *`lock`* можно передать существующий экземпляр `Lock` или `RLock`. Кроме того, возможна передача значения *`lock`*`=False`, но это можно рекомендовать лишь в редких случаях: если вы так поступите, то экземпляр не будет защищен (и, таким образом, не будет синхронизироваться между процессами) и не будет иметь метода `get_lock`. Если вы передаете значение, то *обязаны* использовать форму именованного аргумента: *`lock=значение`*.

Экземпляр *v* класса `Value` предоставляет следующие атрибуты и метод.

`get_lock` `get_lock()`

Возвращает (но не захватывает и не освобождает) блокировку, защищающую *v*.

`value` `value`

Атрибут, доступный для чтения и записи, который используется для установки и получения примитивного значения, хранимого в *v*.

Чтобы гарантировать атомарность операций, выполняемых над примитивным значением объекта *v*, их следует защитить с помощью инструкции `with v.get_lock():`. В качестве типичного примера можно привести операцию присваивания.

```
with v.get_lock():
    v.value += 1
```

Однако в случае выполнения любым другим процессом незащищенной операции над тем же примитивным значением, пусть даже это будет такая атомарная операция, как простое присваивание наподобие `v.value = x`, "все ставки аннулируются": защищенная и незащищенная операции могут вогнать систему в *состояние гонки*. Поэтому принимайте все меры к тому, чтобы обеспечить безопасность операций: если хотя бы одна из операций, выполняемых над `v.value`, не является атомарной (и потому нуждается в защите путем помещения ее в блок `v.get_lock():`), защищайте все операции, выполняемые над значением `v`, помещая их в подобные блоки.

Класс `Array`

Конструктор класса `Array` имеет следующую сигнатуру.

`Array` `Array(`*`typecode`*`, `*`size_or_initializer`*`, `*`lock`*`=True)`

Массив фиксированного размера для хранения примитивных значений (все элементы относятся к одному и тому же примитивному типу). Аргумент *`typecode`* — это строка, определяющая примитивный тип значения, как в случае модуля `array`, описанного в табл. 15.3. (Кроме того, *`typecode`* может определять также тип из модуля `ctypes`, рассмотренного в разделе "Модуль `ctypes`" главы 24, но необходимость в этом возникает лишь в редких случаях.) Аргументом *`size_or_initializer`* может быть итерируемый объект, используемый для инициализации массива. Им также может быть целое число, используемое в качестве размера массива (в этом случае каждый элемент массива инициализируется значением `0`).

Если аргумент *`lock`* задан равным `True` (значение по умолчанию), то конструктор `Array` создает и использует новую блокировку для защиты экземпляра. Однако в аргументе *`lock`* можно передать существующий экземпляр `Lock` или `RLock`. Кроме того, возможна передача значения *`lock`*`=False`, но это можно рекомендовать лишь в редких случаях: если вы так поступите, то экземпляр не будет защищен (и, таким образом, не будет синхронизироваться между процессами) и не будет иметь метода `get_lock`. Если вы передаете значение, то *обязаны* использовать форму именованного аргумента: *`lock=значение`*.

Экземпляр *`a`* класса `Array` предоставляет следующий метод.

`get_lock` `get_lock()`

Возвращает (но не захватывает и не освобождает) блокировку, защищающую *`a`*.

Для получения и изменения значений при работе с экземпляром *`a`* можно использовать индексы и срезы. Поскольку размер *`a`* фиксирован, то при присваивании значения срезу вы должны использовать итерируемый объект того же размера, что и срез, которому он присваивается. Экземпляр *`a`* также является итерируемым объектом.

В тех особых случаях, когда экземпляр *`a`* создается с использованием аргумента *`typecode`*, равного `'c'`, вы можете обратиться к атрибуту `a.value` для получения содержимого *`a`* в виде байтовой строки, а также присвоить `a.value` значение в виде байтовой строки, длина которой не превышает `len(a)`. Если *`s`* — байтовая строка, причем `len(s)<len(a)`, то `a.value = s` означает `a[:len(s)+1] = s+b'\0'`. Эта

запись отражает представление символьных строк в языке C, которые всегда завершаются байтом 0.

```
a = multiprocessing.Array('c', b'four score and seven')
a.value = b'five'
print(a.value)   # выводит b'five'
print(a[:])      # выводит b'five\x00score and seven'
```

Класс Manager

Класс `multiprocessing.Manager` — это подкласс класса `multiprocessing.Process` с теми же самыми методами и атрибутами. Дополнительно он предоставляет методы, предназначенные для создания экземпляров любого из примитивов многозадачной синхронизации, а также классы `Queue`, `dict`, `list` и `Namespace`, последний из которых — это класс, позволяющий устанавливать и получать произвольные именованные атрибуты. Каждый из этих методов носит название класса, экземпляры которого он создает, и возвращает прокси-объект для доступа к соответствующему экземпляру, который может использоваться любым процессом для вызова методов (в том числе специальных методов наподобие тех, которые предназначены для индексирования экземпляров `dict` или `list`) экземпляра, принадлежащего процессу менеджера.

Большинство операторов и обращений к методам и атрибутам экземпляров, представляемых прокси-объектами, передается экземплярам, однако *операторы сравнения* не передаются. Для выполнения сравнений следует использовать локальную копию объекта, представляемого прокси-объектом.

```
p = some_manager.list()
p[:] = [1, 2, 3]
print(p == [1, 2, 3])       # выводит False, поскольку для
                            # сравнения используется p
print(list(p) == [1, 2, 3]) # выводит True, поскольку для
                            # сравнения используется
                            # локальная копия
```

Конструктор класса `Manager` не принимает аргументов. Существуют эффективные способы адаптации подклассов `Manager`, обеспечивающие возможность взаимодействия с другими процессами (включая те, которые выполняются на других компьютерах, входящих в состав сети) и предоставляющие другой набор методов для создания прокси-объектов, однако в данной книге они не рассматриваются. Во многих случаях вполне хватает одного простого подхода к использованию класса `Manager`, который заключается в передаче создаваемых им прокси-объектов другим процессам, обычно с помощью очередей или в виде аргументов функции *`target`* конструктора `Process`.

Предположим, например, что имеется некая длительно выполняющаяся и интенсивно задействующая CPU функция *`f`*, которая принимает аргумент в виде строки и возвращает некий результат. Мы хотим, чтобы в случае набора исходных строк эту функцию можно было использовать для получения словаря, который содержал

бы эти строки в качестве ключей для доступа к соответствующим значениям. Чтобы можно было следить за тем, в каких процессах выполняется функция *f*, мы выводим идентификатор процесса перед вызовом *f*. Ниже приведен один из возможных способов решения этой задачи.

Листинг 14.1

```
import multiprocessing as mp

def f(s):
    """Выполняется длительное время, после чего
       возвращает результат."""
    import time, random
    time.sleep(random.random()*2)     # имитировать замедленность
                                      # вычислений
    return s+s    # те или иные вычисления

def runner(s, d):
    print(os.getpid())
    d[s] = f(s)

def make_dict(set_of_strings):
    mgr = mp.Manager()
    d = mgr.dict()
    workers = []
    for s in set_of_strings:
        p = mp.Process(target=runner, args=(s, d))
        p.start()
        workers.append(p)
    for p in workers:
        p.join()
    return dict(d)
```

Пул процессов

В реальной жизни следует остерегаться создания неограниченного количества рабочих процессов, как это было сделано в листинге 14.1. Выигрыш в производительности проявляется лишь до тех пор, пока количество процессов не сравняется с количеством ядер CPU данного компьютера (которое можно определить с помощью функции `multiprocessing.cpu_count()`) или не будет оставаться близким к нему, в зависимости от платформы и загруженности процессора другими заданиями. Превышение оптимального количества процессов приводит лишь к существенному увеличению накладных расходов без каких-либо дополнительных положительных эффектов.

Как следствие, общепринятая практика заключается в том, чтобы с самого начала использовать ограниченный пул рабочих процессов, которые должны выполнять всю работу. Класс `multiprocessing.Pool` берет на себя управление этими процессами от вашего имени.

Класс Pool

Конструктор класса Pool имеет следующую сигнатуру.

Pool Pool(*processes*=None, *initializer*=None, *initargs*=(), *maxtasksperchild*=None)

Создает и возвращает экземпляр *p* класса Pool. Аргумент *processes* задает количество процессов в пуле. По умолчанию он принимается равным значению, возвращаемому функцией cpu_count(). Если аргумент *initializer* не равен None, то он представляет функцию, вызываемую в начале каждого процесса из пула с аргументами *initargs* в качестве аргументов: *initializer*(**initargs*). Если аргумент *maxtasksperchild* не равен None, то он представляет максимальное количество задач, выполняемых каждым процессом из пула. После выполнения процессом из пула этого количества задач он завершается, а вместо него запускается и присоединяется к пулу новый процесс. Если аргумент *maxtasksperchild* равен None (значение по умолчанию), то время жизни процессов совпадает со временем жизни пула.

Экземпляр *p* класса Pool предоставляет следующие методы (все они должны вызываться только в процессе, создавшем экземпляр *p*).

apply apply(*func*, *args*=(), *kwds*={})

Выполняет в одном произвольно выбираемом рабочем процессе вызов функции *func*(**args*, ***kwds*), дожидается ее завершения и возвращает результат выполнения *func*.

apply_async apply_async(*func*, *args*=(), *kwds*={}, *callback*=None)

Запускает в одном произвольно выбираемом рабочем процессе функцию *func*(**args*, ***kwds*) и, не дожидаясь ее завершения, немедленно возвращает экземпляр класса AsyncResult (раздел "Класс AsyncResult"), который в конечном счете предоставляет результат выполнения *func*, когда он становится доступным. Если аргумент *callback* не равен None, то он представляет функцию, вызываемую (в отдельном потоке того процесса из пула, который вызвал метод apply_async) с единственным аргументом в виде результата, возвращенного функцией *func*, когда он становится доступным. Функция, задаваемая с помощью аргумента *callback*, должна выполняться быстро, иначе она заблокирует процесс. Кроме того, эта функция может изменить свой аргумент, если он изменяемый, а возвращаемое ею значение несущественно.

close close()

Предотвращает передачу последующих задач пулу процессов. Рабочие процессы завершаются только после выполнения ими незавершенных задач.

imap imap(*func*, *iterable*, *chunksize*=1)

Возвращает итератор, вызывающий функцию *func* поочередно для каждого элемента итерируемого объекта *iterable*. Аргумент *chunksize* определяет количество последовательных элементов,

передаваемых каждому процессу. В случае очень длинных объектов *iterable* большие значения *chunksize* могут способствовать повышению производительности. Если аргумент *chunksize* равен 1 (значение по умолчанию), то возвращаемый итератор имеет метод next (даже в версии v3, где каноническим именем данного метода итератора является __next__), который может принимать необязательный аргумент *timeout* (значение с плавающей точкой, выражающее длительность таймаута в секундах) и возбуждает исключение multiprocessing.TimeoutError, если результат не был получен по истечении *timeout* секунд.

imap_unordered imap_unordered(*func*, *iterable*, *chunksize*=1)

То же, что и imap, за исключением того, что результаты возвращаются в произвольном порядке (иногда это может улучшить производительность, если очередность получения результатов при итерировании вам безразлична).

join join()

Ожидает, пока все рабочие процессы не завершат работу. Прежде чем вызывать метод join, следует вызвать метод close или terminate.

map map(*func*, *iterable*, *chunksize*=1)

Вызывает функцию *func* поочередно для каждого элемента итерируемого объекта *iterable* в рабочих процессах из пула. Ожидает, пока все они не завершат работу, и возвращает список результатов. Аргумент *chunksize* определяет количество последовательных элементов, передаваемых каждому процессу. В случае очень длинных объектов *iterable* большие значения *chunksize* могут способствовать повышению производительности.

map_async map_async(*func*, *iterable*, *chunksize*=1, *callback*=None)

Вызывает функцию *func* для каждого элемента итерируемого объекта *iterable* в рабочих процессах из пула. Не ожидая завершения любого из них, немедленно возвращает экземпляр AsyncResult (раздел "Класс AsyncResult"), который в конечном счете предоставляет список результатов выполнения *func*, когда он становится доступным. Если аргумент *callback* не равен None, то он представляет функцию и вызывается (в отдельном потоке процесса, вызвавшего метод map_async) с единственным аргументом в виде списка результатов работы функции *func*, когда он становится доступным. Функция *callback* должна выполняться быстро, иначе она заблокирует процесс. Кроме того, эта функция может изменить свой аргумент, если он изменяемый, а возвращаемое ею значение несущественно.

terminate terminate()

Немедленно прекращает выполнение всех рабочих процессов из пула, не дожидаясь, пока они завершат свою работу.

Для сравнения ниже приведен пример выполнения той же задачи, что и в листинге 14.1, но с использованием класса `Pool`.

```
import multiprocessing as mp

def f(s):
    """Выполняется длительное время, после чего
       возвращает результат."""
    import time, random
    time.sleep(random.random()*2)     # имитировать замедленность
                                      # вычислений
    return s+s    # те или иные вычисления

def runner(s):
    print(os.getpid())
    return s, f(s)

def make_dict(set_of_strings):
    with mp.Pool() as pool:
        d = dict(pool.imap_unordered(runner, set_of_strings))
        return d
```

Класс `AsyncResult`

Методы `apply_async` и `map_async` класса `Pool` возвращают экземпляр класса `AsyncResult`. Экземпляр *r* класса `AsyncResult` предоставляет следующие методы.

`get` `get(`*timeout*`=None)`

Блокируется и возвращает результат, когда он становится доступным, или повторно возбуждает исключение, сгенерированное в процессе вычисления результата. Если аргумент *timeout* не равен `None`, то он является числом с плавающей точкой, представляющим длительность тайм-аута в секундах. Если по истечении *timeout* секунд результат не будет готов, метод `get` возбуждает исключение `multiprocessing.TimeoutError`.

`ready` `ready()`

Не блокируется. Возвращает значение `True`, если результат готов, в противном случае возвращает значение `False`.

`successful` `successful()`

Не блокируется. Возвращает значение `True`, если результат готов и в процессе вычислений не было возбуждено исключение. Возвращает значение `False`, если в процессе выполнения вычислений было возбуждено исключение. Если результат еще не готов, метод `successful` возбуждает исключение `AssertionError`.

`wait` `wait(`*timeout*`=None)`

Блокируется и ожидает, пока результат не будет готов. Если аргумент `timeout` не равен `None`, то он является числом с плавающей точкой, представляющим длительность тайм-аута в секундах. Если по истечении *timeout* секунд результат не будет готов, то метод `get` возбуждает исключение `multiprocessing.TimeoutError`.

Модуль concurrent.futures

Пакет concurrent содержит единственный модуль: futures. Модуль concurrent.futures входит лишь в стандартную библиотеку версии v3. Чтобы использовать его в версии v2, вам следует загрузить и установить ретроподдержку с помощью команды **pip2 install futures** (или, что эквивалентно, **python2 -m pip install futures**).

Модуль concurrent.futures предоставляет два класса: ThreadPoolExecutor (использующий рабочие потоки) и ProcessPoolExecutor (использующий рабочие процессы), которые реализуют один и тот же абстрактный интерфейс: Executor. Для инстанциализации пула любого рода следует вызвать соответствующий класс с одним аргументом, max_workers, указывающим, сколько потоков или процессов должен содержать пул. Этот аргумент можно опустить, и тогда система сама выберет значение данного параметра (однако для инстанциализации класса ThreadPoolExecutor в версии v2 с ретроподдержкой модуля futures требуется явно задать аргумент max_workers).

Экземпляр *e* класса Executor предоставляет следующие методы.

map map(*func*, **iterables*, *timeout*=None)

Возвращает итератор *it*, элементами которого являются результаты функции *func*, вызываемой поочередно с одним аргументом из каждого объекта *iterables* (используя несколько рабочих потоков или процессов для параллельного выполнения функции *func*). Если аргумент *timeout* не равен None, то он является числом с плавающей точкой, представляющим длительность тайм-аута в секундах. Если по истечении *timeout* секунд вызов next(it) не возвращает результат, то метод map возбуждает исключение concurrent.futures.TimeoutError. В версии v3 вы можете указать (именованный) аргумент *chunksize*: в случае класса ThreadPoolExecutor он игнорируется, но в случае класса ProcessPoolExecutor позволяет указать количество элементов каждого из итерируемых объектов *iterables*, передаваемых каждому рабочему процессу.

shutdown shutdown(*wait*=True)

Предотвращает дальнейшие вызовы методов map и submit. Если аргумент *wait* равен True, то метод shutdown блокируется до завершения всех запланированных для выполнения фьючерсов. Если он равен False, то метод shutdown немедленно возвращает управление. В любом случае процесс не завершается до тех пор, пока не выполнятся все запланированные фьючерсы.

submit submit(*func*, **a*, ***k*)

Гарантирует выполнение функции *func*(**a*, ***k*) в одном из произвольно выбираемых процессов или потоков из пула. Метод submit не блокируется, а возвращает экземпляр класса Future.

Любой экземпляр класса Executor является также менеджером контекста и поэтому пригоден для использования с инструкцией with (причем вызов метода __exit__ аналогичен вызову shutdown(*wait*=True)).

Ниже приведен пример, в котором решается та же задача, что и в листинге 14.1, но с использованием подхода на основе параллелизма выполнения.

```
import concurrent.futures as cf

def f(s):
    """Выполняется длительное время, после чего
       возвращает результат."""
    ...

def runner(s):
    return s, f(s)

def make_dict(set_of_strings):
    with cf.ProcessPoolExecutor() as e:
        d = dict(e.map(runner, set_of_strings))
    return d
```

Метод `submit` класса `Executor` возвращает экземпляр класса `Future`. Экземпляр *f* класса `Future` предоставляет методы, описанные в табл. 14.1.

Таблица 14.1. Методы класса `Future`

Метод	Описание
`add_done_callback`	`add_done_callback(`*`func`*`)` Добавляет вызываемый объект *func* в *f*. Объект *func* вызывается с фьючерсом *f* в качестве единственного аргумента при отмене или завершении *f*
`cancel`	`cancel()` Делает попытку отменить вызов. Возвращает значение `False`, если вызов выполняется и не может быть отменен; в противном случае возвращает значение `True`
`cancelled`	`cancelled()` Возвращает значение `True`, если вызов был успешно отменен; в противном случае возвращает значение `False`
`done`	`done()` Возвращает значение `True`, если вызов выполнен или успешно отменен
`exception`	`exception(`*`timeout`*`=None)` Возвращает исключение, возбужденное вызовом, или значение `None`, если исключение не было возбуждено. Если аргумент *timeout* не равен `None`, то он является числом с плавающей точкой, представляющим длительность тайм-аута в секундах. Если по истечении *timeout* секунд вызов не будет завершен, то метод `exception` возбуждает исключение `concurrent.futures.TimeoutError`. Если вызов отменен, возбуждается исключение `concurrent.futures.CancelledError`

Метод	Описание
result	`result(`*timeout*`=None)` Возвращает результат вызова. Если аргумент *timeout* не равен None, то он является числом с плавающей точкой, представляющим длительность тайм-аута в секундах. Если по истечении *timeout* секунд вызов не завершается, то метод result возбуждает исключение concurrent.futures.TimeoutError. Если вызов отменен, возбуждается исключение concurrent.futures.CancelledError
running	`running()` Возвращает значение True, если вызов выполняется и не может быть отменен

Модуль concurrent.futures также предоставляет две функции.

as_completed `as_completed(`*fs*`, `*timeout*`=None)`

Возвращает итератор *it* для итерирования по экземплярам Future, являющимся элементами итератора *fs*. Если в *fs* имеются дубликаты элементов, то каждый элемент возвращается только один раз. Итератор *it* возвращает по одному выполненному фьючерсу за раз по мере их завершения. Если аргумент *timeout* не равен None, то он является числом с плавающей точкой, представляющим длительность тайм-аута в секундах, и если по истечении *timeout* секунд после возврата очередного фьючерса ни один новый фьючерс не может быть возвращен, то метод as_completed возбуждает исключение concurrent.futures.Timeout.

wait `wait(`*fs*`, `*timeout*`=None, `*return_when*`=ALL_COMPLETED)`

Ожидает поступления экземпляров класса Future, являющихся элементами итератора *fs*. Возвращает именованный кортеж из двух множеств. Первое множество, done, содержит незавершенные фьючерсы (т.е. фьючерсы, которые были выполнены или отменены к моменту возврата управления методом wait). Второе множество, not_done, содержит незавершенные фьючерсы.

Если аргумент *timeout* не равен None, то он является числом с плавающей точкой, представляющим максимальную длительность тайм-аута до возврата управления (если аргумент *timeout* равен None, то метод wait возвращает управление только в случае, если удовлетворяется условие *return_when*, независимо от того, сколько времени потребуется для того, чтобы это произошло).

Аргумент *return_when* управляет тем, когда именно метод wait возвращает управление. Он должен принимать значение в виде одной из трех констант, предоставляемых модулем concurrent.futures, которые описаны ниже.

ALL_COMPLETED

Возврат управления осуществляется только после выполнения или отмены всех фьючерсов.

FIRST_COMPLETED

Возврат управления осуществляется после выполнения или отмены любого из фьючерсов.

FIRST_EXCEPTION

Возврат управления осуществляется после возбуждения любым из фьючерсов исключения. В случае отсутствия исключений эквивалентна константе ALL_COMPLETED.

Архитектура многопоточной программы

Всегда старайтесь организовать работу многопоточных программ таким образом, чтобы обработка любых объектов или подсистем, являющихся внешними по отношению к программе (таких, как файл, база данных, графический пользовательский интерфейс или сетевое соединение), выполнялась единственным потоком. Работа нескольких потоков с одним и тем же объектом возможна, но зачастую чревата различными проблемами.

Если ваша многопоточная программа должна работать с неким внешним объектом, отведите для этого поток, используя объект Queue, из которого поток, взаимодействующий с внешним объектом, будет получать запросы, отправляемые другим потоком. Тогда поток, взаимодействующий с внешним объектом, сможет возвращать результаты, помещая их в один или несколько объектов Queue. В приведенном ниже примере показано, как упаковать эту архитектуру в общий, повторно используемый класс в предположении, что каждая порция работы, выполняемой над внешней подсистемой, может быть представлена вызываемым объектом. (Разбирая примеры, не забывайте о том, что в версии v2 модуль queue называется Queue. Имя класса Queue в этом модуле записывается с прописной буквы Q в обеих версиях, v2 и v3.)

```
import threading, queue
class ExternalInterfacing(threading.Thread):
    def __init__(self, external_callable, **kwds):
        threading.Thread.__init__(self, **kwds) # можно было бы
                                                # использовать
                                                # 'super'
        self.daemon = True
        self.external_callable = external_callable
        self.work_request_queue = queue.Queue()
        self.result_queue = queue.Queue()
        self.start()
    def request(self, *args, **kwds):
        """Вызывается другими потоками как аргумент
           external_callable"""
        self.work_request_queue.put((args,kwds))
        return self.result_queue.get()
    def run(self):
```

```
        while True:
            a, k = self.work_request_queue.get()
            self.result_queue.put(self.external_callable(*a, **k))
```

Как только инстанциализирован объект *ei* класса `ExternalInterfacing`, любой другой поток может вызывать метод `ei.request` так, как если бы он вызывал *external_callable*, не прибегая к этому механизму (с параметрами или без параметров, в зависимости от обстоятельств). Преимуществом механизма `ExternalInterfacing` является то, что все вызовы *external_callable* сериализуются. Это означает, что они выполняются только одним потоком (объектом потока, связанным с *ei*) в определенной очередности, без перекрытия и перехода в состояние гонки (создающее трудности для отладки программ, когда все зависит от того, какой поток начнет выполняться раньше) и возникновения каких-либо других аномалий, возможных в подобных ситуациях.

Если возникает необходимость в совместной сериализации нескольких вызываемых объектов, можно передавать вызываемый объект в качестве части рабочего запроса, а не при инициализации класса `ExternalInterfacing`, тем самым обеспечивая бо́льшую общность. Именно такой более общий подход продемонстрирован в следующем примере.

```
import threading, queue
class Serializer(threading.Thread):
    def __init__(self, **kwds):
        threading.Thread.__init__(self, **kwds) # можно было бы
                                                # использовать
                                                # 'super'
        self.daemon = True
        self.work_request_queue = queue.Queue()
        self.result_queue = queue.Queue()
        self.start()
    def apply(self, callable, *args, **kwds):
        """Вызывается другими потоками как аргумент callable"""
        self.work_request_queue.put((callable, args, kwds))
        return self.result_queue.get()
    def run(self):
        while True:
            callable, args, kwds = self.work_request_queue.get()
            self.result_queue.put(callable(*args, **kwds))
```

Как только инстанциализирован объект *ser* класса `Serializer`, любой другой поток может вызывать метод *ser*.`apply`(*external_callable*) так, как если бы он вызывал *external_callable*, не прибегая к этому механизму (с параметрами или без параметров, в зависимости от обстоятельств). Механизм `Serializer` обладает теми же преимуществами, что и механизм `ExternalInterfacing`, за исключением того, что все вызовы одного и того же или различных вызываемых объектов, обернутых единственным экземпляром *ser*, теперь сериализуются.

Пользовательский интерфейс всей программы является внешней подсистемой, и поэтому с ним должен работать только один поток, а именно — основной поток программы (что является обязательным условием в случае некоторых наборов инструментов для создания интерфейса пользователя и рекомендуется даже в тех случаях, когда поступать так необязательно). Следовательно, поток `Serializer` для данных целей не подходит. Вместо этого работать с пользовательским интерфейсом должен только основной поток программы, который распределяет фактическую нагрузку между рабочими потоками, принимающими запросы с помощью одного объекта `Queue` и возвращающими результаты с помощью другого. Набор рабочих потоков обычно называют *пулом потоков*. Как показано в следующем примере, все рабочие потоки должны разделять одну очередь запросов и одну очередь результатов, поскольку основной поток — единственный, который отправляет рабочие запросы и собирает результаты.

```
import threading
class Worker(threading.Thread):
    IDlock = threading.Lock()
    request_ID = 0
    def __init__(self, requests_queue, results_queue, **kwds):
        threading.Thread.__init__(self, **kwds)
        self.daemon = True
        self.work_request_queue = requests_queue
        self.result_queue = results_queue
        self.start()
    def perform_work(self, callable, *args, **kwds):
        """Вызывается основным потоком как аргумент callable,
           но без возвращаемого значения"""
        with self.IDlock:
            Worker.request_ID += 1
            self.work_request_queue.put(
                (Worker.request_ID, callable, args, kwds))
            return Worker.request_ID
    def run(self):
        while True:
            request_ID, callable, a, k =
                                self.work_request_queue.get()
            self.result_queue.put((request_ID, callable(*a, **k)))
```

Основной поток создает две очереди, а затем инстанциализирует рабочие потоки следующим образом.

```
import queue
requests_queue = queue.Queue()
results_queue = queue.Queue()
for i in range(number_of_workers):
    worker = Worker(requests_queue, results_queue)
```

Всякий раз, когда основной поток должен поручить выполнение работы другому потоку (выполнить некоторый вызываемый объект, для получения результатов

работы которого может потребоваться существенное время), основной поток вызывает метод *worker*.perform_work(*callable*) так, как если бы он вызывал данный объект, не прибегая к этому механизму (с параметрами или без параметров, в зависимости от обстоятельств). Однако метод perform_work не возвращает результаты вызова. Вместо результатов основной поток получает идентификатор *id* рабочего запроса. Если основному потоку требуются результаты, он может получить их с помощью этого идентификатора, поскольку они помечаются им. Преимущество данного механизма заключается в том, что основной поток не блокируется, ожидая завершения продолжительных вычислений, и сразу же готов вновь вернуться к выполнению своих основных обязанностей — обслуживанию пользовательского интерфейса.

Основной поток должен позаботиться о проверке состояния очереди results_queue, поскольку именно в ней в конечном счете появляется результат выполнения каждого рабочего запроса, помеченный соответствующим идентификатором *id*, после того, как рабочий поток, принявший этот запрос из очереди, завершит вычисление результата. Как именно основной поток организует наблюдение за событиями пользовательского интерфейса и за результатами, поступающими из рабочих потоков в очередь результатов, зависит от инструментального набора пользовательского интерфейса или, в случае текстового интерфейса пользователя, от платформы, на которой выполняется программа.

Широко применяемая, хотя и не всегда оптимальная, общая стратегия заключается в том, что основной поток *опрашивает* (периодически проверяет) состояние очереди результатов. На большинстве Unix-подобных платформ такой опрос может осуществляться с помощью функции alarm модуля signal. Инструментальный набор TkinterGUI предоставляет метод after, пригодный для выполнения опроса. Некоторые инструментальные наборы и платформы позволяют использовать более эффективные стратегии (позволяющие рабочим потокам уведомлять основной поток о помещении какого-либо результата в очередь результатов), но не существует общедоступного, кросс-платформенного и кросс-инструментального способа реализации таких стратегий. Поэтому в приведенном ниже иллюстративном примере события пользовательского интерфейса игнорируются, а работа имитируется вычислением случайных выражений несколькими рабочими потоками с различным временем задержки, тем самым завершая предыдущий пример.

```
import random, time
def make_work():
    return '{} {} {}'.format(random.randrange(2,10),
        random.choice(('+', '-', '*', '/', '%', '**')),
        random.randrange(2,10))
def slow_evaluate(expression_string):
    time.sleep(random.randrange(1,5))
return eval(expression_string)
workRequests = {}
def showResults():
    while True:
```

```
        try: id, results = results_queue.get_nowait()
        except queue.Empty: return
        print('Result {}: {} -> {}'.format(
            id, work_requests[id], results))
        del work_requests[id]
for i in range(10):
    expression_string = make_work()
    id = worker.perform_work(slow_evaluate, expression_string)
    work_requests[id] = expression_string
    print('Submitted request {}: {}'
                                  .format(id, expression_string))
    time.sleep(1)
    showResults()
while work_requests:
    time.sleep(1)
    showResults()
```

Окружение процесса

Операционная система предоставляет каждому процессу *P* *окружение* — набор переменных, именами которых являются строки (чаще всего представляемые идентификаторами, записанными в верхнем регистре) и содержимым которых также являются строки. Переменные среды, оказывающие влияние на выполнение операций в Python, рассмотрены в разделе "Переменные среды" главы 1. Командные оболочки операционных систем предлагают способы проверки и изменения свойств окружения при помощи команд оболочки и других средств, упомянутых в разделе "Переменные среды" главы 1.

Автономность окружений процессов

Окружение любого процесса *P* определяется при запуске *P*. После этого только сам процесс *P* может изменить свое окружение. Изменения окружения *P* влияют только на процесс *P*: окружения не являются инструментом межпроцессного взаимодействия (IPC). Что бы ни делал процесс *P*, он не сможет повлиять ни на окружение своего родительского процесса (процесса, который запустил *P*), ни на окружения дочерних процессов, запущенных *ранее* из *P* и выполняющихся в данный момент, ни на процессы, не связанные с процессом *P*. Обычно дочерние процессы процесса *P* получают копию окружения *P* в качестве своего начального окружения. В этом узком смысле окружение *P* влияет на дочерние процессы, запущенные процессом *P* после изменения своего окружения.

Модуль *os* поддерживает атрибут `environ` — отображение, представляющее окружение текущего процесса. Атрибут `os.environ` инициализируется из окружения процесса при запуске Python. Изменение `os.environ` обновляет окружение текущего

процесса, если платформа поддерживает подобные обновления. Ключи и значения в os.environ должны быть строками. В Windows (но не на Unix-подобных платформах) os.environ неявно преобразуются в верхний регистр. В качестве примера ниже показано, как можно попытаться определить используемую вами оболочку или командный процессор.

```
import os
shell = os.environ.get('COMSPEC')
if shell is None: shell = os.environ.get('SHELL')
if shell is None: shell = 'an unknown command processor'
print('Running under', shell)
```

Если программа на языке Python изменяет свое окружение (например, с помощью присваивания os.environ['X']='Y'), то это никак не повлияет на окружение оболочки или командного процессора, запустившего данную программу. Как уже подчеркивалось, в Python, как и в любом другом языке программирования, изменения окружения процесса сказываются лишь на самом процессе, но не на других процессах, выполняющихся в данный момент.

Выполнение других программ

Для выполнения других программ можно использовать функции из модуля os или (на более высоком и обычно более предпочтительном уровне абстракции) модуль subprocess.

Выполнение других программ с помощью модуля os

Обычно для запуска других процессов из программы лучше всего использовать модуль subprocess, описанный в разделе "Модуль Subprocess". Однако модуль os также предлагает для этого несколько инструментов, которые в редких случаях могут обеспечить более простые решения.

Проще всего запустить другую программу с помощью функции os.system, хотя она не предоставляет никаких способов управления внешней программой. Модуль os также содержит ряд функций, имена которых начинаются с префикса exec. Эти функции позволяют осуществлять детальный контроль над выполнением внешней программы. Программа, выполняемая одной из функций exec, заменяет текущую программу (т.е. интерпретатор Python) в том же процессе. Поэтому на практике функции exec в основном используют в случае платформ, позволяющих процессу дублировать самого себя путем ветвления (т.е. на Unix-подобных платформах). Функции модуля os, имена которых начинаются с префиксов spawn и popen, занимают промежуточное положение в смысле простоты использования и предлагаемых возможностей: они кросс-платформенные и не столь простые, как системные, но все же достаточно простые для того, чтобы их можно было использовать для многих целей.

Функции `exec` и `spawn` выполняют заданный исполняемый файл, используя предоставленные им путь к исполняемому файлу, соответствующие аргументы и необязательное отображение с переменными среды. Функции `system` и `popen` выполняют команду, представленную строкой, которая передается новому экземпляру командной оболочки, используемой по умолчанию данной платформой (обычно */bin/sh* в случае Unix и *cmd.exe* в случае Windows). *Команда* — более общее понятие, чем *исполняемый файл*, поскольку она может включать функциональность оболочки (каналы, перенаправление, встроенные утилиты оболочки), используя синтаксис оболочки, специфический для данной платформы. Модуль `os` предоставляет следующие функции.

execl, execle, execlp, execv, execve, execvp, execvpe

`execl(path, *args), execle(path, *args), execlp(path, *args), execv(path, args), execve(path, args, env), execvp(path, args), execvpe(path, args, env)`

Эти функции запускают исполняемый файл (программу), заданный строкой *path*, заменяя текущую программу (т.е. интерпретатор Python) в текущем процессе. Различия в именах функций (часть имени вслед за префиксом *exec*) определяют различия в управлении тремя аспектами поиска и выполнения новой программы.

- Должен ли аргумент *path* определять полный путь к исполняемому файлу программы, или может ли функция принимать имя файла в качестве аргумента *path* и осуществлять поиск исполняемого файла в нескольких каталогах, как это делает операционная система? Функции `execlp`, `execvp` и `execvpe` могут принимать аргумент *path*, представляющий лишь имя файла, а не полный путь к нему. В этом случае функция осуществляет поиск исполняемого файла с указанным именем в каталогах, перечисленных в атрибуте `os.environ['PATH']`. Другие функции требуют, чтобы аргумент *path* представлял полный путь к исполняемому файлу новой программы.
- Представляются ли аргументы, передаваемые новой программе, в виде единой последовательности *args* или в виде отдельных аргументов функции? Функции, имена которых начинаются с префикса `execv`, поддерживают единственный аргумент *args*, являющийся последовательностью аргументов, предназначенных для использования новой программой. Функциям, имена которых начинаются с префикса `execl`, передаются аргументы новой программы в виде отдельных аргументов (в частности, функция `execle` использует свой последний аргумент в качестве окружения для новой программы).
- Определяется ли окружение новой программы явно задаваемым отображением в виде аргумента *env* функции или неявно посредством атрибута `os.environ`? Функции `execle`, `execve` и `execvpe` поддерживают аргумент *env*, являющийся отображением (ключи и значения которого должны быть строками), которое новая программа будет использовать в качестве среды, тогда как другие функции используют для этой цели атрибут `os.environ`.

Каждая из функций `exec` использует первый элемент *args* в качестве имени новой программы, подлежащей выполнению (например, `argv[0]` в случае программы `main` на языке C). Собственно аргументы, передаваемые новой программе, определяются срезом *args*`[1:]`.

popen `popen(cmd, mode='r', buffering=-1)`

Выполняет заданную строкой *cmd* команду в новом процессе *P* и возвращает файловый объект *f*, который обертывает канал стандартного ввода или стандартного вывода процесса *P* (в зависимости от аргумента *mode*). Аргументы *mode* и *buffering* имеют тот же смысл, что и соответствующие аргументы функции *open*, рассмотренной в разделе "Создание объекта "файла" с помощью метода `io.open`" главы 10. Если аргумент *mode* равен `'r'` (значение по умолчанию), то объект *f* доступен только для чтения и обертывает стандартный вывод процесса *P*. Если аргумент *mode* равен `'w'`, то объект *f* доступен только для записи и обертывает стандартный ввод процесса *P*.

Ключевым отличием объекта *f* от других файловых объектов является поведение метода `f.close`. Метод `f.close()` ожидает завершения процесса *P* и в случае его успешного завершения возвращает значение `None`, как при обычном поведении методов `close` файловых объектов. В то же время, если операционная система связывает с завершением процесса *P* целочисленный код ошибки *c*, указывающий на то, что завершение процесса *P* не было успешным, то метод `f.close()` возвращает код ошибки *c*. В случае систем Windows код ошибки *c* является целым числом со знаком, представляющим код возврата из дочернего процесса.

spawnv, spawnve `spawnv(mode, path, args)`
`spawnve(mode, path, args, env)`

Эти функции выполняют программу, указанную с помощью аргумента *path*, в новом процессе *P*, причем аргументы передаются в виде последовательности *args*. Функция `spawnve` использует отображение *env* в качестве окружения процесса *P* (ключи и значения которого должны быть строками), тогда как функция `spawnv` использует для этих целей атрибут `os.environ`. На Unix-подобных платформах существуют другие разновидности функции `os.spawn`, соответствующие разновидностям функции `os.exec`, но на платформах Windows существуют только две такие функции: `spawnv` и `spawnve`.

Аргументом *mode* должен быть один из двух атрибутов, предоставляемых модулем `os`: атрибут `os.P_WAIT` указывает на то, что вызывающий процесс ожидает завершения нового процесса, тогда как атрибут `os.P_NOWAIT` указывает на то, что вызывающий процесс продолжает выполнение одновременно с новым процессом. Если аргументом *mode* является атрибут `os.P_WAIT`, то функция возвращает код завершения *c* процесса *P*: значение `0` указывает на успешное завершение, тогда как значение *c*, меньшее или равное `0`, указывает на то, что выполнение процесса *P* было преждевременно прервано сигналом, а значение *c*, большее `0`, указывает на нормальное, но не являющееся успешным, завершение. Если аргументом `mode` является атрибут `os.P_NOWAIT`, то функция возвращает идентификатор ID процесса *P* (или, в случае Windows, дескриптор процесса *P*). Не существует кросс-платформенных способов использования идентификатора ID или дескриптора процесса *P*. К числу платформозависимых способов их использования (не рассматриваемых в данной книге) относятся функция `os.waitpid` на Unix-подобных платформах и независимый пакет расширений PyWin32 на платформах Windows. Например, ваша интерактивная программа может предоставить пользователю

возможность редактировать текстовый файл, который программа будет читать и использовать. В этом случае вы должны иметь предварительно определенный полный путь к предпочтительному для пользователя текстовому редактору, например `c:\\windows\\notepad.exe` на платформах Windows или `/usr/bin/vim` на Unix-подобных платформах. Предположим, что строка, содержащая этот путь, находится в переменной *editor*, а путь к текстовому файлу, возможность редактирования которого вы хотите предоставить пользователю, содержится в переменной *textfile*:

```
import os
os.spawnv(os.P_WAIT, editor, [editor, textfile])
```

Первый элемент аргумента *args* передается ответвляемой программе в качестве "имени, используемого для вызова программы". Большинство программ игнорирует этот элемент, поэтому обычно в него можно помещать любую строку. Однако на тот случай, если программа редактора ожидает получения этого специального первого аргумента, наиболее простым и эффективным подходом является передача той же самой строки *editor*, которая используется в качестве второго аргумента функции `os.spawnv`.

system `system(cmd)`

Выполняет команду, заданную строкой *cmd*, в новом процессе и возвращает значение `0`, если новый процесс завершается успешно. В случае неуспешного завершения нового процесса система возвращает целочисленный код ошибки, отличный от `0`. (Какой именно код может быть возвращен, зависит от выполняемой команды: общепринятого стандарта для этого значения не существует.)

Обратите внимание на то, что метод `popen` признан устаревшим в версии v2 (но не удален из нее), а затем заново реализован в версии v3 в качестве простой обертки вокруг функции `subprocess.Popen`.

Модуль `subprocess`

Модуль `subprocess` предоставляет весьма обширный класс `Popen`, который поддерживает множество различных методов, позволяющих одним программам выполнять другие.

Popen `class Popen(args, bufsize=0, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=False, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0)`

Класс `Popen` запускает подпроцесс для выполнения другой программы, а также создает и возвращает объект *p*, представляющий этот подпроцесс. Подробные характеристики того, как должен выполняться подпроцесс, определяются обязательным аргументом *args* и множеством необязательных именованных аргументов. Если в процессе создания подпроцесса возбуждается исключение (до запуска другой программы), то класс `Popen` повторно возбуждает это исключение в вызывающем процессе с добавлением атрибута `child_traceback`, представляющего объект трассировки Python для данного процесса. Обычно

таким исключением служит экземпляр класса OSError (или, возможно, класса TypeError или ValueError, указывающего на то, что классу Popen был передан аргумент, имеющий корректный тип или значение).

Что и как выполнять: *args, executable, shell*

Аргумент *args* — это последовательность (как правило, список) строк: первый элемент представляет путь к программе, подлежащей выполнению, а последующие элементы, если таковые имеются, являются аргументами, передаваемыми программе (*args* также может быть просто строкой, если вы не передаете аргументы). Аргумент *executable*, если он не равен None, перекрывает аргумент *args* в части определения того, какую программу выполнять. Если аргумент *shell* равен True, то *executable* определяет, какую оболочку следует использовать для выполнения подпроцесса. Если аргумент *shell* равен True, а аргумент *executable* равен None, то используемой оболочкой является /bin/sh на Unix-подобных системах (в случае Windows используется командный процессор os.environ['COMSPEC']).

Файлы подпроцесса: *stdin, stdout, stderr, bufsize, universal_newlines, close_fds*

Аргументы *stdin*, *stdout* и *stderr* определяют используемые подпроцессом файлы стандартного ввода, вывода и ошибок соответственно. Каждый из них может представлять собой объект PIPE, который создает канал ввода/вывода подпроцесса, иметь значение None, означающее, что подпроцесс использует тот же файл, что и данный ("родительский") процесс, или быть объектом файла (или дескриптором файла), открытого в соответствующем режиме (для чтения — в случае стандартного ввода и для записи — в случае стандартного вывода и потока ошибок). Аргумент *stderr* также может иметь значение STDOUT, указывающее на то, что подпроцесс использует для стандартного потока ошибок тот же файл, что и его стандартный вывод. Аргумент *bufsize* управляет буферизацией данных файлов (если они еще не были открыты до этого) с использованием семантики одноименного аргумента функции open, рассмотренной в разделе "Создание объекта "файла" с помощью метода io.open" главы 10 (значение по умолчанию, равное 0, означает "небуферизованный"). Если аргумент *universal_newlines* равен True, то стандартные потоки *stdout* и *stderr* (если они еще не были открыты до этого) открываются в режиме "универсальный символ перевода строки" ('rU'), рассмотренном в разделе "Аргумент *mode*" главы 10. Если аргумент *close_fds* равен True, то все остальные файлы (кроме файлов стандартного ввода, вывода и ошибок) закрываются в подпроцессе перед тем, как будет выполняться программа или командная оболочка подпроцесса.

Другие аргументы: *preexec_fn, cwd, env, startupinfo, creationflags*

Если аргумент *preexec_fn* не равен None, то он должен быть функцией или другим вызываемым объектом, который вызывается в подпроцессе непосредственно перед выполнением в нем программы или оболочки (только в Unix-подобных системах, в которых этот вызов осуществляется после вызова fork и перед вызовом exec).

Если аргумент *cwd* не равен None, то он должен быть строкой, представляющей путь к существующему каталогу. Текущий рабочий каталог заменяется в подпроцессе на *cwd* перед выполнением в подпроцессе программы или оболочки.

Если аргумент *env* не равен None, то он должен быть отображением (как правило, словарем) с ключами и значениями в виде строк и полностью определяет окружение для нового процесса.

Аргументы *startupinfo* и *creationflags* предназначены только для Windows и передаются вызову CreateProcess Win32 API, используемому для создания подпроцесса в специфических для Windows целях (не рассматриваются в данной книге, которая фокусируется на кросс-платформенных применениях Python).

Атрибуты экземпляров subprocess.Popen

Экземпляр *p* класса Popen предоставляет следующие атрибуты.

args
: (Только в версии v3.) Аргумент *args* конструктора Popen (строка или последовательность строк).

pid
: Идентификатор подпроцесса.

returncode
: Значение None указывает на то, что подпроцесс еще не завершился. В противном случае это целое число: 0 — успешное завершение, >0 — завершение с кодом ошибки, <0 — выполнение подпроцесса было прервано сигналом.

stderr, stdin, stdout
: Если соответствующим аргументом Popen был subprocess.PIPE, то каждый из этих атрибутов является объектом файла, обертывающим соответствующий канал. В противном случае каждый из этих атрибутов равен None. Во избежание возможных взаимоблокировок применяйте метод communicate объекта *p* вместо выполнения операций чтения/записи с использованием этих объектов файлов.

Методы экземпляров subprocess.Popen

Экземпляр *p* класса Popen предоставляет следующие методы.

communicate *p*.communicate(*input*=None)
: Передает строку *input* в качестве входных данных на стандартный ввод подпроцесса (если *input* не равно None), затем считывает данные из стандартного вывода и стандартного потока ошибок процесса в хранящиеся в памяти строки *so* и *se*, пока не исчерпаются данные, и, наконец, ожидает завершения подпроцесса и возвращает пару (двухэлементный кортеж) (so, se). В версии v3 также принимает необязательный аргумент *timeout*.

poll *p*.poll()
: Проверяет, завершился ли процесс, а затем возвращает *p*.returncode.

wait *p*.wait()

Ожидает завершения процесса, а затем возвращает *p*.returncode. В версии v3 также принимает необязательный аргумент *timeout*.

Модуль mmap

Модуль mmap поддерживает отображаемые в памяти файловые объекты. Объект mmap ведет себя подобно строке байтов, поэтому его часто можно использовать вместо байтовых строк. Однако между ними существуют следующие различия:

- объект mmap не предоставляет методы строкового объекта;
- объект mmap изменяемый, тогда как строковые объекты неизменяемые;
- объект mmap также соответствует открытому файлу и ведет себя подобно объекту файла Python (см. раздел "Объекты, подобные файлам, и полиморфизм" в главе 10).

Объект *m* класса mmap можно индексировать или извлекать из него срезы, получая байтовые строки. Поскольку объект *m* изменяемый, также возможно присваивание ему значений по индексу или срезу. Однако в случае присваивания значения срезу *m* правая часть инструкции присваивания должна быть байтовой строкой той же длины, что и срез, которому она присваивается. Поэтому многие полезные приемы, работающие в случае присваивания значений срезам списков (см. раздел "Изменение списка" в главе 3), неприменимы в случае срезов mmap.

Модуль mmap предоставляет функцию-фабрику с несколько отличающимися версиями для Unix-подобных систем и Windows.

mmap *Версия для Windows:*

mmap(*filedesc*, *length*, *tagname*='', *access*=None, *offset*=None)

Версия для Unix:

mmap(*filedesc*, *length*, *flags*=MAP_SHARED, *prot*=PROT_READ | PROT_WRITE, *access*=None, offset=0)

Создает и возвращает объект *m* типа mmap, отображающий в память первые *length* байт файла, определяемого целочисленным дескриптором *filedesc*. Обычно *filedesc* должен быть дескриптором файла, открытого как для чтения, так и для записи (на Unix-подобных платформах — за исключением случаев, когда аргумент *prot* требует только чтения или только записи). (Дескрипторы файлов обсуждаются в разделе "Операции над файловыми дескрипторами" в главе 10.) Чтобы получить объект *m* класса mmap для объекта файла *f* Python, следует использовать вызов m=mmap.mmap(*f*.fileno(), *length*). Аргумент *filedesc* может принимать значение -1 для анонимного отображения в память.

На платформах Windows все отображения в память допускают чтение и запись и разделяются процессами, поэтому все процессы, использующие отображение файла в память, могут видеть изменения, создаваемые другими процессами. Кроме того (только на платформах Windows), функции mmap можно передать строку *tagname*,

определяющую имя отображения в памяти. Именование отображений позволяет связывать с одним и тем же файлом несколько отображений в памяти, но необходимость в этом возникает лишь в редких случаях. Вызов функции mmap лишь с двумя аргументами обладает тем преимуществом, что обеспечивает переносимость кода между Windows и Unix-подобными платформами.

Только на Unix-подобных платформах функции mmap можно передавать атрибут mmap.MAP_PRIVATE в качестве аргумента *flags* для получения частной копии отображения, используемой только данным процессом с применением механизма *копирования при записи* (copy-on-write). Значение mmap.MAP_SHARED, используемое по умолчанию, позволяет получить отображение, совместно используемое всеми процессами, так что изменения отображения, создаваемые одним процессом, видят другие процессы (точно так же, как в Windows). Для получения отображения, допускающего только чтение, но не запись, можно передать атрибут mmap.PROT_READ в качестве аргумента *prot*. Передача атрибута mmap.PROT_WRITE позволяет получить отображение, которое можно только записывать, но не читать. Используя оператор побитового ИЛИ (|), можно передать объединенное значение mmap.PROT_READ|mmap.PROT_WRITE для получения отображения, допускающего как чтение, так и запись.

Вместо аргументов *flags* и *prot* можно передать именованный аргумент *access* (одновременная передача аргумента *access* и любого из аргументов *flags* и *prot* вызывает ошибку). Аргумент *access* может принимать одно из значений ACCESS_READ (объект доступен только для чтения), ACCESS_WRITE (объект доступен для чтения и сквозной записи — значение по умолчанию для Windows), или ACCESS_COPY (объект доступен для чтения и записи с копированием при записи).

Именованный аргумент *pass* определяет смещение (в байтах) начала отображения относительно начала файла. Аргумент *offset* должен указываться в виде положительного целого числа, кратного значению mmap.ALLOCATIONGRANULARITY (или, в случае Unix, mmap.PAGESIZE).

Методы объектов mmap

Объект *m* класса mmap предоставляет следующие методы.

close `m.close()`

Закрывает файл *m*.

find `m.find(sub, start=0, end=None)`

Возвращает наименьший индекс *i*>=*start*, такой, что *sub*==*m*[*i*:*i*+len(*sub*)] (и *i*+len(*sub*)-1<=*end*, если вы передаете *end*). Если такого индекса *i* не существует, то *m*.find возвращает -1. Это поведение аналогично поведению метода find строковых объектов, описанного в табл. 8.1.

flush `m.flush([offset, n])`

Гарантирует, что все изменения, внесенные в объект *m* в памяти, будут записаны в файл *m*. До тех пор, пока не вызван метод *m*.flush, нельзя быть уверенным в том, что состояние файла отражает текущее состояние объекта *m*. Аргумент *offset* определяет смещение начального байта, а аргумент *n* —

диапазон среза, запись которого в файл *m* гарантируется. Можно передать оба аргумента или ни одного: вызов метода `m.flush` только с одним из этих аргументов считается ошибкой.

move `m.move(dstoff, srcoff, n)`

Подобен операции присваивания срезов `m[dstoff:dstoff+n]=m[srcoff:srcoff+n]`, но может работать быстрее. Исходный и целевой срезы могут перекрываться. За исключением подобного перекрывания, метод `move` не влияет на исходный срез (т.е. метод `move` *копирует* байты, но не перемещает их, несмотря на название метода).

read `m.read(n)`

Читает и возвращает строку *s*, содержащую вплоть до *n* байтов, начиная с текущей позиции в файле, а затем перемещает указатель файла *m* на `len(s)` байтов. Если количество байтов между текущей позицией и концом файла меньше *n*, то возвращается доступное количество байтов. В частности, если текущей позицией в файле является конец файла, то метод `m.read` возвращает пустую строку `''`.

read_byte `m.read_byte()`

Возвращает байтовую строку длиной `1`, содержащую один байт в текущей позиции указателя файла *m*, а затем увеличивает указатель файла на `1`. Вызов `m.read_byte()` аналогичен методу `m.read(1)`. Однако, если указатель файла *m* находится в конце файла, то вызов `m.read(1)` возвращает пустую строку `''`, тогда как вызов `m.read_byte()` возбуждает исключение `ValueError`.

readline `m.readline()`

Читает и возвращает одну строку из файла *m*, начиная от текущей позиции указателя файла до следующего символа `'\n'` включительно (или до конца файла в отсутствие символов `'\n'`), а затем увеличивает указатель файла таким образом, чтобы он указывал позицию, непосредственно следующую за только что прочитанными байтами. Если указатель файла *m* находится в конце файла *m*, то метод `readline` возвращает пустую строку `''`.

resize `m.resize(n)`

Изменяет длину отображения *m* в памяти таким образом, что `len(m)` было равно *n*. Не влияет на размер файла *m*. Длина отображения *m* и размер файла являются независимыми величинами. Чтобы установить длину отображения равной размеру файла, следует выполнить вызов `m.resize(m.size())`. Если длина *m* превышает размер файла, то отображение *m* дополняется нулевыми байтами (`\x00`).

rfind `rfind(sub, start=0, end=None)`

Возвращает наибольший индекс *i*>=*start*, такой, что *sub*==`m[i:i+len(sub)]` (и *i*+`len(sub)`-1<=*end*, если вы передаете *end*). Если такого индекса *i* не существует, то вызов `m.rfind` возвращает `-1`. Это поведение аналогично поведению метода `rfind` строковых объектов, описанного в табл. 8.1.

seek *m*.seek(*pos*, *how*=0)

Устанавливает указатель файла *m* в позицию, смещенную от начала отсчета на целое количество байтов *pos*. Аргумент *how* указывает начало отсчета (позицию 0): если *how* равно 0, то началом отсчета служит начало файла, если *how* равно 1 — текущая позиция указателя, если *how* равно 2 — конец файла. Если метод seek пытается установить для указателя файла m отрицательное или положительное смещение, выходящее за пределы длины файла, то возбуждается исключение ValueError.

size *m*.size()

Возвращает длину (количество байтов) файла *m*, а не самого отображения *m*. Для получения длины отображения в памяти *m* следует использовать вызова len(*m*).

tell *m*.tell()

Возвращает текущую позицию указателя файла *m* в виде смещения (в байтах) от начала файла *m*.

write *m*.write(*str*)

Записывает содержащиеся в *str* байты в файл *m*, начиная с текущей позиции указателя файла поверх имеющихся байтов, а затем перемещает указатель на len(*str*) байтов. Если между текущей позицией указателя и концом файла *m* недостаточно места для размещения по крайней мере len(*str*) байтов, то метод write возбуждает исключение ValueError.

write_byte *m*.write_byte(*byte*)

Записывает аргумент *byte*, который должен быть целым числом в версии v3 и односимвольной байтовой строкой в версии v2, в отображение *m*, начиная с текущей позиции указателя файла *m*, поверх имеющихся байтов, а затем увеличивает указатель файла на 1. Если *x* — односимвольная байтовая строка в версии v2, то вызов *m*.write_byte(*x*) аналогичен вызову *m*.write(*x*). Однако, если указатель файла *m* находится в конце файла, то вызов m.write_byte(x) не выполняет никаких действий, в то время как вызов *m*.write(*x*) возбуждает исключение ValueError. Обратите внимание на противоположный характер поведения пары методов write и write_byte по сравнению с парой методов read и read_byte в конце файла: методы write и read_byte возбуждают исключение ValueError, тогда как методы read и write_byte — не возбуждают.

Использование объектов mmap для межпроцессного взаимодействия

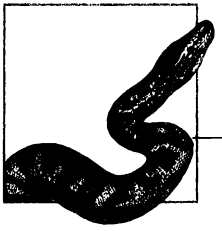
Способ, с помощью которого процессы обмениваются между собой сообщениями, используя модуль mmap, напоминает использование файлов для реализации межпроцессного взаимодействия (IPC): один процесс записывает данные, тогда как другой читает их. Поскольку в основе объекта класса mmap находится файл, некоторые из ваших процессов могут использовать операции ввода-вывода, работая

непосредственно с файлом (см. раздел "Модуль io" в главе 10), в то время как другие процессы могут использовать модуль mmap для доступа к тому же файлу. Вы можете использовать либо модуль mmap, либо файловые операции ввода-вывода, исходя из соображений удобства: в отношении функциональности и производительности оба этих подхода примерно эквивалентны. В качестве примера ниже приведена простая программа, в которой для создания содержимого файла в виде последней строки, введенной пользователем в интерактивном режиме, используются файловые операции ввода-вывода.

```
fileob = open('xxx','w')
while True:
    data = input('Enter some text:')
    fileob.seek(0)
    fileob.write(data)
    fileob.truncate()
    fileob.flush()
```

Ниже приведен пример другой простой программы, которая, если выполнить ее в том же каталоге, где выполняется предыдущая программа, использует модуль mmap (а также функцию time.sleep, описанную в табл. 12.2) для того, чтобы каждую секунду проверять изменение содержимого файла, и выводит новое содержимое.

```
import mmap, os, time
mx = mmap.mmap(os.open('xxx',os.O_RDWR), 1)
last = None
while True:
    mx.resize(mx.size())
    data = mx[:]
    if data != last:
        print(data)
        last = data
    time.sleep(1)
```



15 Математические вычисления

Для выполнения некоторых математических вычислений вам будет вполне достаточно использовать операторы (см. раздел "Операции над числами" в главе 3) и встроенные функции (см. раздел "Встроенные функции" в главе 7). Кроме того, Python предоставляет дополнительные модули, поддерживающие численную обработку данных: `math` и `cmath` (раздел "Модули `math` и `cmath`"), `operator` (раздел "Модуль `operator`"), `random` (раздел "Модуль `random`"), `fractions` (раздел "Модуль `fractions`") и `decimal` (раздел "Модуль `decimal`"). В разделе "Модуль `gmpy2`" будет приведена краткая характеристика стороннего модуля `gmpy2`, дополнительно расширяющего возможности выполнения численных расчетов в Python. Такие расчеты часто требуют обработки числовых *массивов* (раздел "Обработка массивов"), в связи с чем в данной главе также обсуждаются модуль стандартной библиотеки `array` и популярное стороннее расширение NumPy.

Модули `math` и `cmath`

Модуль `math` содержит математические функции, предназначенные для работы с числами с плавающей точкой, а модуль `cmath` — эквивалентные функции для работы с комплексными числами. Например, вызов `math.sqrt(-1)` возбуждает исключение, однако вызов `cmath.sqrt(-1)` возвращает результат в виде комплексного числа `1j`.

Как и в случае любого другого модуля, самый непосредственный способ использования этих функций, обеспечивающий лучшую удобочитаемость, заключается в том, чтобы импортировать нужный модуль, скажем `math`, в начале кода с помощью инструкции `import math` и впоследствии выполнять явные вызовы функций наподобие `math.sqrt()`. Но если вызовы хорошо известных математических функций, содержащихся в используемых модулях, встречаются довольно часто, то в виде исключения из общего правила считается допустимым добавить в начало кода инструкции импорта вида `from math import *`, что позволит в дальнейшем использовать более короткие вызовы наподобие `sqrt()`.

Оба модуля содержат два атрибута со значениями в виде чисел с плавающей точкой, которые представляют фундаментальные математические константы, `e` и `pi`, а также ряд функций, описанных в табл. 15.1.

Таблица 15.1. Функции модулей `math` и `cmath`

Функция	Описание	Модуль
acos, asin, atan, cos, sin, tan	`acos(x)` Возвращают арккосинус, арксинус, арктангенс, косинус, синус и тангенс аргумента `x` соответственно (углы выражаются в радианах)	`math` и `cmath`
acosh, asinh, atanh, cosh, sinh, tanh	`acosh(x)` Возвращают гиперболические арккосинус, арксинус, арктангенс, косинус, синус и тангенс аргумента `x` соответственно (углы выражаются в радианах)	`math` и `cmath`

atan2	`atan2(y, x)`
	Аналогична `atan(y/x)`, за исключением того, что в `atan2` учитываются знаки обоих аргументов.
	`>>> import math`
	`>>> math.atan(-1./-1.)`
	`0.78539816339744828`
	`>>> math.atan2(-1., -1.)`
	`-2.3561944901923448`
	Если `x` равно `0`, то `atan2` возвращает значение `pi/2`, тогда как при попытке деления на `x` было бы возбуждено исключение `ZeroDivisionError`
		Только `math`

ceil	`ceil(x)` Возвращает значение `float(i)`, где `i` — наименьшее целое число, такое, что `i>=x`	Только `math`
e	Математическая константа `e` (`2.718281828459045`)	`math` и `cmath`
exp	`exp(x)` Возвращает значение `e**x`	`math` и `cmath`
erf	`erf(x)` Возвращает функцию ошибки `x`, используемую в статистических расчетах	Только `math`
fabs	`fabs(x)` Возвращает абсолютную величину `x`	Только `math`

Функция	Описание	Модуль
factorial	`factorial(x)` Возвращает факториал `x`. Если `x` имеет отрицательное или нецелочисленное значение, то возбуждает исключение `ValueError`	Только `math`
floor	`floor(x)` Возвращает значение `float(i)`, где `i` — наибольшее целое число, такое, что `i<=x`	Только `math`
fmod	`fmod(x, y)` Возвращает число с плавающей точкой `r`, знак которого совпадает со знаком `x`, такое, что `r==x-n*y` для некоторого целого числа `n` и `abs(r)<abs(y)`. Аналогична операции `x%y`, за исключением того, что в тех случаях, когда `x` и `y` имеют разные знаки, `x%y` имеет тот же знак, что и `y`, а не тот же знак, что и `x`	Только `math`
fsum	`fsum(iterable)` Возвращает результат в виде числа с плавающей точкой, представляющего сумму значений, содержащихся в итерируемой последовательности `iterable`, с большей точностью, чем функция `sum`	Только `math`
frexp	`frexp(x)` Возвращает пару `(m, e)`, содержащую "мантиссу" (строго говоря, значащую часть числа со знаком) и экпоненту аргумента `x`, где `m` — число с плавающей точкой, а `e` — целое число, такое, что `x==m*(2**e)` и `0.5<=abs(m)<1`, за исключением того, что `frexp(0)` возвращает `(0.0, 0)`	Только `math`
gcd	`gcd(x, y)` Возвращает наибольший общий делитель `x` и `y`. Если `x` и `y` оба равны нулю, то возвращает значение `0`	Только `math` (только в версии v3)
hypot	`hypot(x, y)` Возвращает значение `sqrt(x*x+y*y)`	Только `math`
inf	Значение с плавающей точкой, представляющее положительную бесконечность, аналогичное тому, которое возвращает вызов `float('inf')`	Только `math`
isclose	`isclose(x, y, rel_tol=1e-09, abs_tol=0.0)` Возвращает значение `True`, если `x` и `y` приблизительно равны между собой в пределах относительной погрешности `rel_tol` и с минимальным значением абсолютной погрешности `abs_tol`. В противном случае возвращает значение `False`.	`math` и `cmath` (только в версии v3)

Функция	Описание	Модуль
	Значению относительной погрешности `rel_tol`, используемому по умолчанию, соответствуют различия в 9-м знаке после десятичной точки. Значение `rel_tol` должно быть больше `0`. Аргумент `abs_tol` используется для сравнения значений, близких к нулю: его значение должно быть равным по крайней мере `0.0`. Никакое значение (включая `NaN`) не может считаться близким к `NaN`; каждое из значений `-inf` и `inf` считается близким лишь к самому себе. За исключением поведения в отношении значений `+/-inf`, поведение функции `isclose` описывается следующим образом: `abs(x-y) <= max(rel_tol * max(abs(x), abs(y)), abs_tol)`  **Не используйте оператор == для сравнения чисел с плавающей точкой** Учитывая приблизительный характер арифметики чисел с плавающей точкой, проверка равенства двух таких чисел `x` и `y` может иметь смысл лишь в редких случаях: самые незначительные отличия в способах их получения могут приводить к не поддающимся контролю различиям в их значениях. Избегайте использования операции `x==y`; используйте вместо нее функцию `math.isclose(x, y)`	
isfinite	`isfinite(x)` Возвращает значение `True`, если `x` (в `cmath` — как действительная, так и мнимая часть `x`) не является ни бесконечностью, ни `NaN`; в противном случае возвращает значение `False`	`math` и `cmath` (только в версии v3)
isinf	`isinf(x)` Возвращает значение `True`, если `x` (в `cmath` — либо действительная, либо мнимая часть `x`) является положительной или отрицательной бесконечностью; в противном случае возвращает значение `False`	`math` и `cmath`
isnan	`isnan(x)` Возвращает значение `True`, если `x` (в `cmath` — либо действительная, либо мнимая часть `x`) — `NaN`; в противном случае возвращает значение `False`	`math` и `cmath`
ldexp	`ldexp(x, i)` Возвращает значение `x*(2**i)` (`i` должно быть целочисленным значением; если `i` — значение с плавающей точкой, `ldexp` возбуждает исключение `TypeError`)	Только `math`

Функция	Описание	Модуль
log	`log(x)` Возвращает натуральный логарифм *x*	`math` и `cmath`
log10	`log10(x)` Возвращает логарифм *x* по основанию 10. Также имеется функция `log2(x)` (только в модуле `math` и только в версии v3), которая возвращает логарифм *x* по основанию 2	`math` и `cmath`
modf	`modf(x)` Возвращает пару `(f, i)`, содержащую дробную и целую части *x* в виде двух целых чисел с тем же знаком, что и *x*, таких, что `i==int(i)` и `x==f+i`	Только `math`
nan	Значение с плавающей точкой "не число" (not a number, NaN), аналогичное значению `float('nan')`	Только `math`
pi	Математическая константа π, равная `3.141592653589793`	`math` и `cmath`
phase	`phase(x)` Возвращает фазу *x* в виде значения с плавающей точкой в диапазоне (-π, π). Аналогична `math.atan2(x.imag, x.real)`. См. раздел "Conversions to and from polar coordinates" онлайн-документации Python	Только `cmath`
polar	`polar(x)` Возвращает представление *x* в полярных координатах в виде пары `(r, phi)`, где *r* — модуль *x*, а *phi* — фаза *x*. Результат аналогичен паре `(abs(x), cmath.phase(x))`. См. раздел "Conversions to and from polar coordinates" онлайн-документации Python	Только `cmath`
pow	`pow(x,y)` Возвращает значение `x**y`	Только `math`
sqrt	`sqrt(x)` Возвращает значение квадратного корня из *x*	`math` и `cmath`
trunc	`trunc(x)` Возвращает значение *x*, усеченное до целого числа	Только `math`

Не забывайте о том, что в силу особенностей внутреннего представления значений с плавающей точкой в компьютере они никогда не являются абсолютно точными. Это продемонстрировано в следующем примере, в котором также показано, чем может быть полезна функция `isclose`.

```
>>> f = 1.1 + 2.2 - 3.3              # интуиция подсказывает, что
                                     # f должно быть равно 0
>>> f==0
False
>>> f
4.440892098500626e-16
>>> math.isclose(0,f,abs_tol=1e-15) # abs_tol для сравнений
                                     # вблизи 0
True
>>> g = f-1
>>> g
-0.9999999999999996               # почти равно -1
>>> math.isclose(-1,g)            # значение по умолчанию подходит
                                  # для данного сравнения
True
>>> isclose(-1,g,rel_tol=1e-15) # но вы можете задать точность,
True
>>> isclose(-1,g,rel_tol=1e-16) # включая более высокую точность
False
```

Модуль operator

Модуль operator предоставляет функции, эквивалентные операторам Python. Эти функции оказываются удобными в тех случаях, когда вызываемые объекты должны сохраняться, передаваться в качестве аргументов и возвращаться в качестве результатов работы функции. Предлагаемые модулем operator функции называются так же, как и соответствующие специальные методы (см. раздел "Специальные методы" в главе 4). Каждая такая функция допускает использование двух имен, одно из которых содержит префикс и суффикс в виде двойного символа подчеркивания, а второе не имеет: например, обе функции, operator.add(a,b) и operator.__add__(a,b), возвращают значение a+b. В версии v3[1] была добавлена поддержка матричного умножения для инфиксного оператора @, но вы должны реализовать его, определив собственные методы __matmul__(), __rmatmul__() и/или __imatmul__() (это требование действовало на момент выхода книги). Впрочем, использование оператора @ (но пока что не оператора @=) для матричного умножения поддерживается пакетом NumPy.

Некоторые из функций, предоставляемых модулем operator, приведены в табл. 15.2.

Таблица 15.2. Функции, предоставляемые модулем operator

Метод	Сигнатура	Поведение
abs	abs(*a*)	abs(*a*)
add	add(*a* , *b*)	*a* + *b*

[1] А именно, в Python 3.5.

Метод	Сигнатура	Поведение
and_	and_(*a* , *b*)	*a* & *b*
concat	concat(*a* , *b*)	*a* + *b*
contains	contains(*a* , *b*)	*b* in *a*
countOf	countOf(*a* , *b*)	*a* .count(*b*)
delitem	delitem(*a* , *b*)	del *a*[*b*]
delslice	delslice(*a* , *b* , *c*)	del *a*[*b*:*c*]
div	div(*a* , *b*)	*a* / *b*
eq	eq(*a* , *b*)	*a* == *b*
floordiv	floordiv(*a* , *b*)	*a* // *b*
ge	ge(*a* , *b*)	*a* >= *b*
getitem	getitem(*a* , *b*)	*a* [*b*]
getslice	getslice(*a* , *b* , *c*)	*a* [*b* : c]
gt	gt(*a* , *b*)	*a* > *b*
indexOf	indexOf(*a* , *b*)	*a* .index(*b*)
invert, inv	invert(*a*), inv(*a*)	~ *a*
is	is(*a* , *b*)	*a* is *b*
is_not	is_not(*a* , *b*)	*a* is not *b*
le	le(*a* , *b*)	*a* <= *b*
lshift	lshift(*a* , *b*)	*a* << *b*
lt	lt(*a* , *b*)	*a* < *b*
matmul	matmul(*m1*, *m2*)	*m1* @ *m2*
mod	mod(*a* , *b*)	*a* % *b*
mul	mul(*a* , *b*)	*a* * *b*
ne	ne(*a* , *b*)	*a* != *b*
neg	neg(*a*)	- *a*
not_	not_(*a*)	not *a*
or_	or_(*a* , *b*)	*a* | *b*
pos	pos(*a*)	+ *a*
repeat	repeat(*a* , *b*)	*a* * *b*
rshift	rshift(*a* , *b*)	*a* >> *b*
setitem	setitem(*a* , *b* , *c*)	*a* [*b*]= *c*
setslice	setslice(*a* , *b* , *c* , *d*)	*a* [*b* : *c*]= *d*
sub	sub(*a* , *b*)	*a* - *b*
truediv	truediv(*a* , *b*)	*a*/*b* # "true" div -> no truncation
truth	truth(*a*)	not not *a*, bool(*a*)
xor	xor(*a* , *b*)	*a* ^ *b*

Модуль operator также предоставляет две высокоуровневые функции. Их возвращаемыми результатами являются функции, которые можно передавать в качестве именованного аргумента *key*= методу sort списков, встроенной функции sorted, функции itertools.groupby() и другим встроенным функциям, таким как min и max.

attrgetter attrgetter(*attr*)

Возвращает вызываемый объект *f*, такой, что *f*(*o*) — это то же самое, что getattr(*o*, *attr*). Строка *attr* может включать точки (.), и в этом случае вызываемый результат функции attrgetter повторно вызывает функцию getattr. Например, operator.attrgetter('a.b') эквивалентно lambda o: getattr(getattr(o, 'a'), 'b').

attrgetter(**attrs*)

Если функция attrgetter вызвана с несколькими аргументами, то результирующий вызываемый объект извлекает каждый из именованных атрибутов и возвращает результирующий кортеж значений.

itemgetter itemgetter(*key*)

Возвращает вызываемый объект *f*, такой, что f(*o*) — это то же самое, что getitem(o, *key*).

itemgetter(**keys*)

Если функция attrgetter вызвана с несколькими аргументами, то результирующий вызываемый объект извлекает каждый из элементов, определяемых ключами, и возвращает результирующий кортеж значений. Предположим, что имеется список списков L, в котором каждый подсписок содержит по крайней мере три элемента: вы хотите отсортировать L на месте по третьему элементу каждого подсписка, причем подсписки, в которых третьи элементы равны, сортируются по их первым элементам. Проще всего это можно сделать таким образом:

```
import operatorя
L.sort(key=operator.itemgetter(2, 0))
```

Случайные и псевдослучайные числа

Модуль random стандартной библиотеки генерирует псевдослучайные числа с различными распределениями. В базовом генераторе равномерно распределенных псевдослучайных чисел используется алгоритм вихря Мерсенна с периодом 2**19937-1.

Физически случайные и криптографически стойкие случайные числа

Несмотря на то что псевдослучайные числа, предоставляемые модулем random, отлично подходят для многих целей, они не обладают криптографической стойкостью. Если вам необходимы случайные числа более высокого качества, воспользуйтесь функцией os.urandom (т.е. функцией из модуля os, а *не* из модуля random) или

создайте экземпляр класса SystemRandom, содержащегося в модуле random (который сам вызовет функцию os.urandom).

urandom urandom(*n*)

Возвращает *n* случайных байтов, прочитанных из физических источников случайных битов, таких как /dev/urandom в старых выпусках Linux. В версии v3 используйте системный вызов getrandom() в Linux 3.17 и выше. (В OpenBSD 5.6 и выше теперь используется C-функция getrandom().) Работая в Windows, используйте такие источники криптографически стойких случайных чисел, как CryptGenRandom API. Если подходящий источник отсутствует в текущей системе, функция urandom возбуждает исключение NotImplementedError.

Также имеется альтернативный источник физически случайных чисел: http://www.fourmilab.ch/hotbits.

Модуль random

Все функции модуля random являются методами одного скрытого глобального экземпляра класса random.Random. Экземпляры Random можно инстанциализировать явно для получения нескольких генераторов, не имеющих общего состояния. Явную инстанциализацию рекомендуется применять в тех случаях, когда случайные числа требуются нескольким потокам (потоки рассматриваются в главе 14). Другой возможный подход предполагает инстанциализацию класса SystemRandom в ситуациях, когда требуются высококачественные случайные числа (раздел "Физически случайные и криптографически стойкие случайные числа"). В данном разделе документированы наиболее часто используемые функции, предоставляемые модулем random.

choice choice(*seq*)

Возвращает случайный элемент из последовательности *seq*.

getrandbits getrandbits(*k*)

Возвращает целое число >=0 с *k* случайными битами, как функция randrange(2**k) (но работает быстрее и не испытывает проблем при больших значениях *k*).

getstate getstate()

Возвращает кешируемый и сериализуемый с помощью модуля pickle объект *S*, представляющий текущее состояние генератора. Впоследствии объект *S* может быть передан функции setstate для восстановления состояния генератора.

jumpahead jumpahead(*n*)

Продвигает состояние генератора, как если бы было сгенерировано *n* случайных чисел. Эта операция выполняется быстрее, чем генерирование и игнорирование *n* случайных чисел.

randint randint(*start*, *stop*)

Возвращает случайное целое число *i* из равномерного распределения, такое, что *start*<=*i*<=*stop*. Оба граничных значения включаются в диапазон: это совершенно не свойственно Python, поэтому обычно предпочтительнее использовать функцию randrange.

random random()

Возвращает случайное число с плавающей точкой *r* из однородного распределения 0<=*r*<1.

randrange randrange([*start*,] *stop*[, *step*])

Подобна вызову choice(range(*start*, *stop*, *step*)), но работает быстрее.

sample sample(*seq*, *k*)

Возвращает новый список, *k* элементов которого являются уникальными элементами, извлеченными случайным образом из последовательности *seq*. Элементы этого списка располагаются в случайном порядке, поэтому любой его срез в равной мере является действительной случайной выборкой. Последовательность seq может содержать повторяющиеся элементы. В этом случае каждое вхождение элемента является кандидатом для включения в выборку, и выборка также может включать подобные дубликаты.

seed seed(*x*=None)

Инициализирует состояние генератора. Аргументом *x* может быть любой хешируемый объект. Если *x* равен None и если первым был загружен модуль random, то аргумент *seed* использует текущее время системы (или некоторый платформозависимый источник случайности, если таковой имеется) для получения затравочного значения. Обычно аргумент *x* является целым числом величиной вплоть до 27814431486575. Большие значения *x* также допустимы, но они могут приводить к тем же состояниям генератора, что и меньшие значения.

setstate setstate(*S*)

Восстанавливает состояние генератора. Аргумент *S* должен быть результатом предыдущего вызова функции getstate (допускается, чтобы этот вызов был выполнен в другой программе или при прошлом запуске этой программы, если объект *S* был корректно передан или сохранен и восстановлен).

shuffle shuffle(*alist*)

Перемешивает на месте элементы изменяемой последовательности *alist*.

uniform uniform(*a*, *b*)

Возвращает случайное число с плавающей точкой *r* из однородного распределения, такого, что *a*<=r<*b*.

Кроме того, модуль random предоставляет несколько других функций, которые генерируют псевдослучайные числа с плавающей точкой из других распределений

(бета, гамма, экспоненциальное, Гаусса, Парето и др.), используя внутренний вызов функции `random.random` в качестве источника случайности.

Модуль `fractions`

Модуль `fractions` предоставляет класс рациональных чисел `Fraction`, экземпляры которого могут создаваться на основе двух целых чисел, другого рационального числа или строки. Вы можете передать конструктору пару целых чисел (возможно, со знаком): числитель и знаменатель. Если знаменатель равен `0`, то возбуждается исключение `ZeroDivisionError`. Строка может быть задана в виде `'3.14'` или же включать числитель (возможно, со знаком), символ косой черты и знаменатель, например `'-22/7'`. Кроме того, класс `Fraction` поддерживает создание экземпляров на основе экземпляров `decimal.Decimal`, а также на основе чисел с плавающей точкой (хотя в последнем случае, учитывая ограниченную точность таких чисел, вы можете получить не тот результат, которого ожидали). Экземпляры класса `Fraction` имеют свойства `numerator` и `denominator`.

Приведение дробей к наименьшему общему знаменателю

Класс `Fraction` приводит дроби к наименьшему общему знаменателю. Например, инструкция `f = Fraction(226, 452)` создает экземпляр `f`, эквивалентный тому, который создается с помощью вызова `Fraction(1, 2)`. Значения числителя и знаменателя, переданные конструктору `Fraction`, невозможно восстановить из созданного экземпляра.

```
from fractions import Fraction
>>> Fraction(1,10)
Fraction(1, 10)
>>> Fraction(Decimal('0.1'))
Fraction(1, 10)
>>> Fraction('0.1')
Fraction(1, 10)
>>> Fraction('1/10')
Fraction(1, 10)
>>> Fraction(0.1)
Fraction(3602879701896397, 36028797018963968)
>>> Fraction(-1, 10)
Fraction(-1, 10)
>>> Fraction(-1,-10)
Fraction(1, 10)
```

Класс `Fraction` также поддерживает несколько методов, в том числе метод `limit_denominator`, позволяющий создать приближенное значение числа с плавающей точкой, выраженное рациональной дробью. Например, вызов `Fraction(0.0999)`.

`limit_denominator(10)` возвращает `Fraction(1, 10)`. Экземпляры `Fraction` не являются изменяемыми объектами и могут служить ключами в словарях и элементами множеств, а также использоваться в арифметических операциях с другими числами. Более подробную информацию о модуле `fractions` можно получить в онлайн-документации (`https://docs.python.org/3/library/fractions.html`).

Кроме того, как в версии v2, так и в версии v3, модуль `fractions` предоставляет функцию `gcd`, которая работает точно так же, как и функция `math.gcd` (существующая лишь в версии v3), описанная в табл. 15.1.

Модуль `decimal`

Тип `float` в Python — это представление двоичного числа с плавающей точкой, обычно соответствующее стандарту IEEE 754 и аппаратно реализуемое в современных компьютерах. Краткое практическое введение в арифметику чисел с плавающей точкой содержится в статье Дэвида Голдберга *Every Computer Scientist Should Know about Floating-Point Arithmetic* (`https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`). Статья на ту же тему, сфокусированная на специфике Python, является частью онлайн-руководства (`https://docs.python.org/3/tutorial/floatingpoint.html`). Еще один великолепный краткий обзор доступен в Интернете (`http://www.lahey.com/float.htm`).

Часто, особенно в финансовых расчетах, предпочтительнее использовать *десятичные* числа с фиксированной точкой. Python предоставляет их реализацию, соответствующую стандарту IEEE 854 для основания 10, в модуле `decimal` стандартной библиотеки. Этот модуль снабжен отличной документацией в обеих версиях, v2 и v3: в ней вы найдете полную справочную информацию, ссылки на соответствующие стандарты, практическое руководство и рекомендации по работе с модулем `decimal`. Мы же рассмотрим лишь небольшое подмножество функциональности модуля `decimal`, которое соответствует наиболее часто используемым частям этого модуля.

Модуль `decimal` предоставляет класс `Decimal` (неизменяемые экземпляры которого являются десятичными числами), классы исключений, а также классы и функции для работы с *арифметическим контекстом*, который определяет такие характеристики, как точность и округление, и вычислительные аномалии которого (такие, как деление на нуль, переполнение, переполнение снизу и др.) в случае их возникновения возбуждают исключения. В контексте, используемом по умолчанию, точность составляет 28 десятичных цифр, округление является "наполовину четным" (результат округляется до ближайшего представляемого десятичного числа; если результат располагается в точности посредине между двумя такими числами, то он округляется до того числа, последняя цифра которого является четной), а аномалиями, возбуждающими исключения, являются следующие: недопустимая операция, деление на нуль и переполнение.

Чтобы создать десятичное число, следует вызвать класс `Decimal` с одним аргументом, которым может быть целое число, число с плавающей точкой или кортеж. Если

задано число с плавающей точкой, то его двоичное представление преобразуется без потерь в точный десятичный эквивалент (при таком преобразовании может потребоваться использовать 53 и более цифр).

```
from decimal import Decimal
df = Decimal(0.1)
df
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

Если вас не устраивает такое поведение, можете передать число в виде строки.

```
ds = Decimal(str(0.1)) # or, directly, Decimal('0.1')
ds
Decimal('0.1')
```

Не составляет труда написать функцию-фабрику, которую можно использовать для экспериментов, особенно интерактивных, с десятичными числами.

```
def dfs(x):
return Decimal(str(x))
```

Теперь `dfs(0.1)` означает то же самое, что и `Decimal(str(0.1))` или `Decimal('0.1')`, но такая запись гораздо лаконичнее.

Вы также можете использовать метод `quantize` класса `Decimal` для создания нового десятичного числа путем округления аргумента до указанного количества значащих цифр.

```
dq = Decimal(0.1).quantize(Decimal('.00'))
dq
Decimal('0.10')
```

Если вы начнете с кортежа, то вам потребуется предоставить три аргумента: знак (`0` — для положительных, `1` — для отрицательных), кортеж цифр и целочисленную экспоненту (показатель степени).

```
pidigits = (3, 1, 4, 1, 5)
Decimal((1, pidigits, -4))
Decimal('-3.1415')
```

Как только получены экземпляры `Decimal`, их можно использовать в операциях сравнения, в том числе с числами с плавающей точкой (используя для этой цели функцию `math.isclose`), сериализовать и десериализовать их с помощью модуля `pickle`, а также использовать в качестве ключей в словарях и элементах множеств. Они также могут участвовать в арифметических операциях между собой и с целыми числами, но не с числами с плавающей запятой (во избежание непредвиденной потери точности результатов), что продемонстрировано в следующем примере.

```
>>> a = 1.1
>>> d = Decimal('1.1')
>>> a == d
```

```
False
>>> math.isclose(a, d)
True
>>> a + d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
  'decimal.Decimal' and 'float'
>>> d + Decimal(a) # new decimal constructed from a
Decimal('2.20000000000000000888178419700') # whoops
>>> d + Decimal(str(a)) # convert a to decimal with str(a)
Decimal('2.20')
```

В онлайн-документации даны полезные советы касательно применения десятичных чисел в финансовых расчетах и тригонометрических вычислениях (`https://docs.python.org/2.7/library/decimal.html#recipes`). Также там приведен список часто задаваемых вопросов (FAQ).

Модуль gmpy2

Модуль `gmpy2` — это C-расширение Python, которое поддерживает библиотеки GMP, MPFR и MPC и обеспечивает ускоренное выполнение арифметических операций над числами кратной точности (числа, точность представления которых ограничена лишь доступным объемом памяти). Основная ветвь разработки `gmpy2` поддерживает потокобезопасные контексты. Модуль `gmpy2` доступен для загрузки из каталога PyPI (`https://pypi.python.org/pypi/gmpy2`).

Обработка массивов

Для представления массивов можно использовать списки (см. раздел "Списки" в главе 3), а также модуль `array` стандартной библиотеки (раздел "Модуль `array`"). Массивами можно манипулировать с помощью циклов, срезов, генераторов списков, итераторов, выражений-генераторов (все они рассмотрены в главе 3), встроенных функций, таких как `map`, `reduce` и `filter` (все они рассмотрены в разделе "Встроенные функции" главы 7), и модулей стандартной библиотеки, таких как `itertools` (рассмотрен в разделе "Модуль `itertools`" главы 7). Если вам нужны лишь простые одномерные массивы, можете ограничиться модулем `array`. Однако в случае крупных числовых массивов указанные выше функции работают медленнее и менее удобны в использовании по сравнению со сторонними расширениями, такими как NumPy и SciPy (рассмотрены в разделе "Расширения для работы с числовыми массивами"). Для анализа данных или задач моделирования больше всего подходит библиотека Pandas, реализованная поверх низкоуровневой библиотеки NumPy.

Модуль array

Модуль array предоставляет одноименный тип, экземпляры которого, как и списки, являются изменяемыми последовательностями. Массив array *a* — это одномерная последовательность, элементами которой могут быть только символы или только числа определенного типа, заданного при создании *a*.

По сравнению со списком массив array.array обладает тем преимуществом, что он позволяет экономить память, выделяемую для хранения объектов одного и того же (числового или символьного) типа. Объект *a* типа array имеет доступный только для чтения атрибут *a*.typecode, который устанавливается при создании объекта посредством задания единственного символа — кода типа элементов *a*. Возможные коды типов для модуля array приведены в табл. 15.3.

Таблица 15.3. Коды типов для модуля array

Код типа	Тип C	Типе Python	Минимальный размер
'c'	char	str (длина 1)	1 байт (только в версии v2)
'b'	char	int	1 байт
'B'	unsigned char	int	1 байт
'u'	unicode char	unicode (длина 1)	2 байта (4 для версии Python "широкой" сборки)
'h'	short	int	2 байта
'H'	unsigned short	int	2 байта
'i'	int	int	2 байта
'I'	unsigned int	int	2 байта
'l'	long	int	4 байта
'L'	unsigned long	int	4 байта
'q'	long	int	8 байтов (только в версии v3)
'Q'	unsigned long long	int	8 байтов (только в версии v3)
'f'	float	float	4 байта
'd'	double	float	8 байтов

Тип 'c' доступен только в версии v2. Тип 'u' доступен в обеих версиях, v2 и v3, с размером элемента 2 байта в случае "узкой" сборки Python и 4 байта в случае "широкой" сборки. Типы q и Q (только в версии v3) доступны только на платформах, поддерживающих тип long long языка C (или, в случае Windows, тип __int64).

Размер в байтах каждого элемента может превышать минимальное значение, в зависимости от конкретной аппаратной архитектуры. Его можно получить с помощью доступного только для чтения атрибута *a*.itemsize. Модуль array предоставляет лишь объект типа array.

array `array(typecode, init='')`

Создает и возвращает объект `array` с заданным кодом типа. Аргумент `init` может быть строкой (байтовой, за исключением кода типа `'u'`), длина которой кратна значению *itemsize*: байты этой строки, интерпретируемые как значения в двоичном формате, непосредственно инициализируют элементы *a*. Он также может быть итерируемым объектом (содержащим символы для типов `'c'` и `'u'` и числа для остальных типов): каждый элемент итерируемого объекта инициализирует один элемент *a*.

Объекты массива предоставляют все методы и операции изменяемых последовательностей (рассмотрены в разделе "Операции над последовательностями" в главе 3), за исключением метода `sort`. Конкатенация с помощью операторов + или +=, а также операция присваивания значения срезу требуют, чтобы оба операнда были массивами с одним и тем же кодом типа. В отличие от этого аргументом, предоставляемым методу *a*`.extend`, может быть любой итерируемый объект, элементы которого являются допустимыми элементами для *a*.

В дополнение к методам изменяемых последовательностей объект `array` предоставляет следующие методы2.

byteswap `a.byteswap()`

Меняет порядок следования байтов в каждом элементе *a* на противоположный.

fromfile `a.fromfile(f,n)`

Читает *n* элементов, интерпретируемых как машинные (в двоичном формате) значения, из объекта файла *f* и присоединяет элементы к *a*. Обратите внимание на то, что файл *f* должен быть открыт для чтения в двоичном режиме, например в режиме `'rb'`. Если количество имеющихся в *f* элементов меньше *n*, то метод `fromfile` возбуждает исключение `EOFError` после добавления в массив доступных элементов.

fromlist *a*`.fromlist(`*L*`)`

Присоединяет к *a* все элементы списка *L*.

fromstring, frombytes *a*`.fromstring(`*s*`)` *a*`.frombytes(`*s*`)`

Метод `fromstring` (только в версии v2) присоединяет к *a* байты строки *s*, интерпретируемые как машинные значения. Значение `len(`*s*`)` должно быть в точности кратным значению *a*`.itemsize`. Метод `frombytes` (только в версии v3) работает так же (читая строку *s* как байты).

tofile *a*`.tofile(`*f*`)`

Записывает все элементы *a*, взятые в виде машинных значений, в объект файла *f*. Обратите внимание на то, что файл *f* должен быть открыт для записи в двоичном режиме, например в режиме `'wb'`.

[2] Поскольку типу *str* в версии v2 соответствуют байтовые строки, а в версии v3 — строки Unicode, то во избежание недоразумений методы, которые в версии v2 назывались *fromstring* и *tostring*, в версии v3 переименованы в методы *frombytes* и *tobytes*.

tolist `a.tolist()`

Создает и возвращает объект списка с теми же элементами, что и в *a*, действуя как вызов `list(a)`.

tostring, tobytes `a.tostring()` `a.tobytes()`

Метод `tostring` (только в версии v2) возвращает строку с байтами из всех элементов *a*, взятых в виде машинных значений. Для любого объекта *a* справедливо соотношение `len(a.tostring())== len(a)*a.itemsize`. Вызов `f.write(a.tostring())` означает то же самое, что и `a.tofile(f)`. Аналогичным образом метод `tobytes` (только в версии v3) возвращает байтовое представление элементов массива.

Расширения для работы с числовыми массивами

Как вы уже имели возможность убедиться, Python отлично поддерживает обработку числовых данных. Однако сторонняя библиотека SciPy и такие пакеты, как NumPy, Matplotlib, Sympy, IPython/Jupyter и Pandas, предоставляют еще больше инструментов, предназначенных для этих целей. В этом разделе мы познакомимся с пакетом NumPy, а затем дадим краткое описание SciPy и других пакетов (раздел "SciPy") со ссылками на соответствующую документацию.

NumPy

Для работы с простыми одномерными числовыми массивами вам вполне будет достаточно модуля `array` стандартной библиотеки. Однако для научных вычислений, сложных видов обработки изображений, многомерных массивов, линейной алгебры и других задач, требующих обработки больших объемов данных, потребуется независимый пакет NumPy. Этот пакет снабжен обширной онлайн-документацией (`https://docs.scipy.org/doc/`). Кроме того, можете бесплатно скачать электронную книгу в формате PDF *Guide to NumPy* Трэвиса Олифанта (`http://web.mit.edu/dvp/Public/numpybook.pdf`).

NumPy или numpy?

В разных местах документации данный пакет фигурирует под именами NumPy и Numpy, однако в коде он используется как пакет `numpy`, который обычно импортируют с помощью инструкции `import numpy as np`. В данном разделе используются все эти названия.

Библиотека NumPy содержит класс `ndarray`, для расширения функциональности которого можно создавать производные классы (`https://docs.scipy.org/doc/numpy/user/basics.subclassing.html`). Объект типа `ndarray` — *n*-мерный и состоит из однородных элементов (которые могут включать контейнеры, содержащие неоднородные типы). Объект *a* класса `ndarray` имеет определенное количество измерений (также называемых *осями*), известное как *ранг. Скаляр* (т.е. одиночное число) имеет

ранг 0, вектор — ранг 1, матрица — ранг 2 и т.д. Кроме того, объект ndarray имеет *форму*, которую можно получить с помощью свойства shape. Например, для матрицы *m* с двумя столбцами и тремя строками *m*.shape возвращает значение (3,2).

NumPy поддерживает более широкий диапазон числовых типов (экземпляров dtype; https://docs.scipy.org/doc/numpy/user/basics.types.html), чем Python, однако числовыми типами по умолчанию являются следующие: bool_ — одиночный байт, int_ — *int64* или *int32* (в зависимости от платформы), float_ — сокращение от *float64* и complex_ — сокращение от *complex128*.

Создание массива NumPy

Существует несколько способов создания массива в NumPy. Наиболее распространенными из них являются следующие:

- с помощью функции-фабрики np.array, исходя из последовательности (зачастую вложенной) с использованием вывода типов или путем явного указания *dtype*;
- с помощью функций-фабрик zeros, ones, empty, по умолчанию использующих в качестве *dtype* тип *float64*, и indice, по умолчанию использующей тип *int64*;
- с помощью функции-фабрики arange (с обычными аргументами *start*, *stop*, *stride*) или функции-фабрики linspace (*start*, *stop*, *quantity*) для улучшения поведения в отношении чисел с плавающей точкой;
- путем чтения данных из файлов с помощью других функций np (например, из CSV-файлов с помощью функции genfromtxt).

Ниже приведены примеры создания массивов с помощью вышеперечисленных функций.

```
import numpy as np

np.array([1, 2, 3, 4]) # из списка Python
array([1, 2, 3, 4])

np.array(5, 6, 7) # распространенная ошибка: передача элементов
                  # по отдельности (они должны передаваться в виде
                  # последовательности (например, в виде списка)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: only 2 non-keyword arguments accepted

s = 'alph', 'abet' # кортеж из двух строк
np.array(s)
array(['alph', 'abet'], dtype='<U4')

t = [(1,2), (3,4), (0,1)] # список кортежей
```

```
np.array(t, dtype='float64') # явное задание типа
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 0.,  1.]]

x = np.array(1.2, dtype=np.float16) # скаляр
x.shape
()
x.max()
1.2002

np.zeros(3) # векторная форма по умолчанию
array([ 0., 0., 0.])

np.ones((2,2)) # указание формы
array([[ 1.,  1.],
       [ 1.,  1.]])

np.empty(9) # произвольные числа типа float64
array([  4.94065646e-324, 9.88131292e-324, 1.48219694e-323,
         1.97626258e-323, 2.47032823e-323, 2.96439388e-323,
         3.45845952e-323, 3.95252517e-323, 4.44659081e-323])

np.indices((3,3))
array([[[0, 0, 0],
        [1, 1, 1],
        [2, 2, 2]],

       [[0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]]])

np.arange(0, 10, 2) # верхняя граница диапазона не включается
array([0, 2, 4, 6, 8])

np.linspace(0, 1, 5) # по умолчанию: конечная точка включается
array([ 0. , 0.25, 0.5 , 0.75, 1. ])

np.linspace(0, 1, 5, endpoint=False) # конечная точка не
                                     # включается
array([ 0. , 0.2, 0.4, 0.6, 0.8])

import io
np.genfromtxt(io.BytesIO(b'1 2 3\n4 5 6')) # использование
                                           # псевдофайла
array([[ 1., 2., 3.],
       [ 4., 5., 6.]])
```

```
with io.open('x.csv', 'wb') as f:
    f.write(b'2,4,6\n1,3,5')
np.genfromtxt('x.csv', delimiter=',') # использование фактического
                                      # CSV-файла
array([[ 2., 4., 6.],
       [ 1., 3., 5.]])
```

Форма, индексы и срезы

Каждый объект `ndarray` имеет атрибут `shape` (форма), представляющий собой кортеж целых чисел. Значение `len(a.shape)` является рангом *a*. Например, ранг одномерного числового массива (также известного как *вектор*) равен `1`, а кортеж `a.shape` имеет всего один элемент. В более общем случае каждый элемент `a.shape` представляет длину *a* для соответствующего измерения. Количество элементов массива *a*, называемое его *размером*, является произведением всех элементов `shape` (также доступным в виде свойства `a.size`). Каждое измерение *a* называется *осью*. Индексы осей принимают значения от `0` и выше, как это принято в Python. Допускаются отрицательные индексы осей, которые отсчитываются справа, поэтому индексу `-1` соответствует последняя (крайняя справа) ось.

Каждый массив (за исключением скаляра, т.е. массива ранга `0`) является последовательностью Python. Каждый вектор `a[i]` массива *a* является подмассивом *a* в том смысле, что он является массивом, ранг которого на единицу меньше ранга *a*: `a[i].shape==a.shape[1:]`. Например, если *a* — двумерная матрица (имеет ранг `2`), то для любого допустимого значения `i` элемент `a[i]` является одномерным подмассивом *a*, соответствующим строке матрицы. Если ранг *a* равен `1` или `0`, то векторами *a* являются сами элементы *a* (один элемент в случае массива ранга `0`). Поскольку *a* — это последовательность, то ее можно индексировать, используя обычный синтаксис индексирования, для получения доступа к элементам *a* или их изменения. Обратите внимание на то, что элементами *a* являются подмассивы *a*. Лишь в случае массивов ранга `1` или `0` понятие элементов массива в более широком понимании совпадает с более узким понятием элементов самого низкого уровня.

Как и в случае любой другой последовательности, возможно взятие *срезов a*: после выполнения операции `b=a[i:j]` массив *b* имеет тот же ранг, что и *a*, а `b.shape` равно `a.shape`, за исключением того, что значение `b.shape[0]` равно длине среза `i:j` (`j-i`, если `a.shape[0]>j>=i>=0`, и т.д.).

Для любого массива *a* можно выполнить вызов `a.reshape` (или, что эквивалентно, вызов `np.reshape` с *a* в качестве первого аргумента). Результирующая форма должна соответствовать `a.size`: если `a.size` равно `12`, то вы можете выполнить вызов `a.reshape(3,4)` или `a.reshape(2,6)`, но вызов `a.reshape(2,5)` возбудит исключение `ValueError`. Обратите внимание на то, что функция `reshape` не выполняет операцию на месте: результирующий массив требует явной привязки или повторной привязки, т.е. выполнения присваиваний вида `a = a.reshape(i,j)` или `b = a.reshape(i,j)`.

Элементы массива *a* (не являющегося скаляром) можно перебирать в цикле `for`, как это можно делать в отношении любой другой последовательности. Например, цикл

```
for x in a:
    process(x)
```

означает то же самое, что

```
for _ in range(len(a)):
    x = a[_]
    process(x)
```

В этих примерах каждый элемент *x* массива *a* в цикле `for` является подмассивом *a*. Например, если *a* — двумерная матрица, то каждая переменная *x* в любом из этих циклов является одномерным массивом *a*, соответствующим строке матрицы.

Вы также можете индексировать *a* и получать его срезы с помощью кортежей. Например, если ранг *a* больше или равен 2, то можно записать `a[i][j]` в виде `a[i,j]` для любых допустимых значений `i` и `j`, как для повторного связывания, так и для получения значения. Индексирование с помощью кортежа работает быстрее, и с ним гораздо удобнее работать. Не записывайте круглые скобки внутри квадратных для указания того, что индексирование осуществляется с помощью кортежа: достаточно записать индексы один за другим, разделив их запятой. Запись `a[i,j]` означает то же самое, что и `a[(i,j)]`, но форма без круглых скобок гораздо более удобна для чтения.

Индексирование — это взятие среза, когда один или несколько элементов кортежа являются срезами или (не более одного раза в одном срезе) специальной формой `...` (также доступной, только в версии v3, в виде встроенного типа Python `Ellipsis`). Форма `...` расширяется до количества символов среза (`:`), соответствующего рангу массива, срез которого извлекается. Например, выражение `a[1,...,2]` эквивалентно выражению `a[1,:,:,2]`, если ранг *a* равен 4, но выражению `a[1,:,:,:,:,2]`, если ранг *a* равен 6.

Приведенные ниже фрагменты кода демонстрируют обход элементов в цикле, индексирование и взятие срезов массива.

```
a = np.arange(8)
a
array([0, 1, 2, 3, 4, 5, 6, 7])

a = a.reshape(2,4)
a
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])

print(a[1,2])
6
```

```
a[:,:2]
array([[0, 1],
       [4, 5]])

for row in a:
    print(row)
[0 1 2 3]
[4 5 6 7]

for row in a:
    for col in row[:2]: # первые два элемента в каждой строке
        print(col)
0
1
4
5
```

Матричные операции в NumPy

Как уже упоминалось в разделе "Модуль `operator`", пакет NumPy реализует новый[3] оператор `@` для матричного умножения массивов. Результат выражения `a1 @ a2` равен результату выражения `np.matmul(a1, a2)`. Если обе матрицы — двумерные, они обрабатываются как обычные матрицы. Если один аргумент — вектор, вы повышаете его ранг до двумерного массива, временно присоединяя к его форме `1` (в начале или в конце). Не используйте оператор `@` со скаляром; в этом случае используйте оператор `*` (см. следующий пример). Матрицы допускают сложение (с помощью оператора `+`) со скаляром (см. пример), а также с векторами и другими матрицами (формы должны быть совместимыми). Для матриц также доступно точечное произведение: `np.dot(a1, a2)`. Ознакомьтесь с приведенными ниже простыми примерами.

```
a = np.arange(6).reshape(2,3) # двумерная матрица
b = np.arange(3)              # вектор

a
array([[0, 1, 2],
       [3, 4, 5]])

a + 1 # сложение со скаляром
array([[1, 2, 3],
       [4, 5, 6]])

a + b # сложение с вектором
array([[0, 2, 4],
       [3, 5, 7]])
```

[3] Введен в версии Python 3.5.

```
a * 2 # умножение на скаляр
array([[ 0, 2, 4],
       [ 6, 8, 10]])

a * b # умножение на вектор
array([[ 0, 1, 4],
       [ 0, 4, 10]])

a @ b # матричное умножение на вектор
array([ 5, 14])

c = (a*2).reshape(3,2) # использование умножения на скаляр для
c                      # создания другой матрицы
array([[ 0, 2],
       [ 4, 6],
       [ 8, 10]])

a @ c # матричное умножение двух двумерных матриц
array([[20, 26],
       [56, 80]])
```

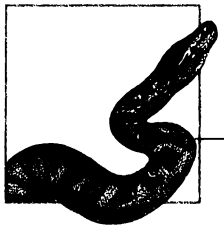
Возможности библиотеки NumPy настолько обширны, что для их подробного рассмотрения потребовалась бы целая книга, и в этом разделе мы лишь слегка коснулись некоторых деталей. За более полной информацией обратитесь к документации библиотеки NumPy (`https://docs.scipy.org/doc/numpy-1.10.0/index.html`).

SciPy

Библиотека NumPy содержит классы и методы, предназначенные для обработки многомерных массивов, тогда как библиотека SciPy обеспечивает поддержку более сложных математических вычислений. Если библиотека NumPy содержит лишь несколько методов для решения задач линейной алгебры, то библиотека SciPy предлагает намного больше инструментов для работы в этой области, включая функции, реализующие продвинутые методы декомпозиции матриц, а также более сложные функции, допускающие использование второго матричного аргумента, которые позволяют решать уравнения обобщенной задачи о собственных значениях. В общем случае, если вам приходится выполнять сложные научные или инженерные расчеты, имеет смысл установить обе библиотеки, SciPy и NumPy.

На сайте SciPy.org (`https://www.scipy.org/index.html`) приведены ссылки на документацию целого ряда других пакетов (`https://www.scipy.org/docs.html`), интегрированных с библиотеками SciPy и NumPy: Matplotlib (поддержка построения двумерных графиков), Sympy (поддержка символьной математики), IPython (мощная интерактивная консольная оболочка и ядро веб-приложения; в настоящее время дальнейшая разработка этой оболочки ведется в рамках проекта Jupyter) и Pandas (поддержка анализа данных и построения моделей; ссылки

на соответствующие руководства вы найдете по адресу `http://pandas.pydata.org/pandas-docs/stable/tutorials.html`, а ссылки на книги и другие материалы — по адресу `https://ssearch.oreilly.com/?q=pandas&x=0&y=0`). Наконец, если вас интересуют технологии *глубокого обучения* (Deep LEarning), воспользуйтесь открытой библиотекой TensorFlow, для которой предусмотрен Python API (`https://www.tensorflow.org/api_docs/python/`).



16

Тестирование, отладка и оптимизация кода

Задачи программирования не ограничиваются одним лишь написанием кода. Работа может считаться завершенной лишь тогда, когда вы убедитесь в том, что код работает корректно и обеспечивает приемлемое быстродействие. *Тестирование* (раздел "Тестирование") — это испытание кода путем его автоматического выполнения в известных условиях и проверки того, что реальные результаты соответствуют ожидаемым. *Отладка* (раздел "Отладка") — это обнаружение причин некорректного поведения кода и их устранение (зачастую код легче исправить, когда известны причины неполадок).

Термин *оптимизация* (раздел "Оптимизация") часто используют в качестве зонтичного термина, означающего комплекс мероприятий, направленных на повышение эффективности программного кода. Оптимизация подразделяется на *эталонное тестирование* (бенчмаркинг, или измерение производительности для определенного класса задач с целью определения того, находится ли она в допустимых пределах), *профилирование* (инструментирование программы дополнительным кодом для выявления узких мест, отрицательно влияющих на производительность) и собственно оптимизацию (повышение производительности программы в целом до допустимого уровня за счет устранения узких мест). Совершенно очевидно, что устранение узких мест, ухудшающих производительность программы, возможно лишь после того, как они будут найдены (с помощью профилирования), что, в свою очередь, требует выявления (с помощью бенчмаркинга) самого факта существования проблем, приводящих к снижению производительности.

В этой главе темы рассматриваются в том же естественном порядке, в котором с ними приходится сталкиваться в процессе разработки: в первую очередь выполняется тестирование, затем — отладка, а в завершение — оптимизация кода. Нередко к тестированию и отладке относятся (ошибочно, по нашему мнению) как к скучной рутинной работе, тогда как оптимизация многими воспринимается как интересное

занятие. Поэтому, если бы вам пришлось ограничиться чтением только одного из разделов этой главы, то мы рекомендовали бы раздел "Разработка достаточно быстрых приложений Python", содержащий краткую сводку основных сведений о принятом в Python подходе к оптимизации, суть которого хорошо отражает ставший классическим шутливый совет британского ученого М. Джексона (`http://wiki.c2.com/?RulesOfOptimization`): "Правила оптимизации. Правило 1: не делайте этого. Правило 2 (только для специалистов): пока что не делайте этого".

Все эти задачи весьма обширны и играют очень важную роль, и отдельное рассмотрение любой из них могло бы занять не одну книгу. В этой главе мы не пытаемся даже кратко охарактеризовать соответствующие методики и фокусируем внимание на специфических подходах, технологиях и инструментах, используемых для этих целей в Python.

Тестирование

В данной главе проводится различие между двумя видами тестирования: модульным и системным. Тестирование программного обеспечения — чрезвычайно широкая и очень важная область деятельности, в которой различия можно было провести по гораздо большему количеству признаков, однако мы сосредоточимся на рассмотрении вопросов, являющихся наиболее важными с точки зрения разработчиков. Многие разработчики неохотно тратят время на тестирование программ, наивно считая, что это отрывает их от "настоящей" работы, однако такая точка зрения весьма недальновидна: чем раньше обнаружены дефекты в программе, тем легче их устранить, — один час времени, затраченного на разработку тестов, с лихвой окупится впоследствии, поскольку тесты обеспечат скорейшее выявление дефектов и сэкономят вам массу времени, которое иначе пришлось бы потратить на отладку программного обеспечения на более поздних этапах цикла разработки.

Модульное и системное тестирование

Термин *модульное (блочное) тестирование* означает написание и выполнение тестов, предназначенных для проверки работоспособности изолированного модуля или меньшей программной единицы (блока), например класса или функции. *Системное тестирование* (другие названия — *функциональное*, *интеграционное*, *сквозное* тестирование) подразумевает выполнение всей программы с известными входными данными. В некоторых классических книгах, посвященных тестированию, проводится различие между тестированием по *методу белого ящика*, основанным на знании внутреннего устройства программы, и тестированием по *методу черного ящика*, выполняемым в отсутствие таких знаний. Эта классическая точка зрения коррелирует с современным подходом, разделяющим модульное и системное тестирование, но не дублирует его.

Модульное и системное тестирование преследуют разные цели. Модульное тестирование идет в ногу с процессом разработки; вы можете и должны тестировать каждый модуль (программную единицу) по мере его разработки. Один относительно современный подход (впервые предложенный Дж. Вайнбергом в его классической книге *The Psychology of Computer Programming*, опубликованной в 1971 году) известен как *разработка через тестирование* (test-driven development — TDD): для каждой атомарной функциональности, которую должна обеспечить ваша программа, вы прежде всего пишете модульные тесты и лишь после этого приступаете к написанию кода, реализующего данную функциональность, и запускаете тесты. Может создаться впечатление, что в подходе на основе TDD все перевернуто с ног на голову, однако у такого подхода есть ряд преимуществ. Например, он гарантирует, что ни одна функциональная возможность не избежит модульного тестирования. Ведение разработки по принципу "сначала тесты" полезно тем, что оно вынуждает сосредоточить внимание в первую очередь на том, какие задачи решает та или иная функция, класс или метод, и лишь после этого продумывать, как именно должна быть реализована данная функция, класс или метод. Инновацией в духе TDD является подход BDD (Behaviour-Driven Development — разработка на основе описания поведения; `https://habr.com/post/216923/`).

Чтобы протестировать программную единицу (модуль), которая может зависеть от других программных единиц, часто приходится использовать *программные заглушки* (stubs, mocks)[1] — код, эмулирующий реализацию интерфейсов различных модулей, которые обеспечивают известный, корректный отклик в случаях, требующих тестирования других модулей. Модуль `mock` (`https://docs.python.org/3/library/unittest.mock.html`), компонент стандартной библиотеки в версии v3, включенный в пакет `unittest` (ретроподдержка для версии v2 доступна на сайте PyPI; `https://pypi.python.org/pypi/mock`), облегчает реализацию подобных заглушек.

Системное тестирование вступает в игру на более поздних этапах, поскольку для этого требуется, чтобы система уже существовала и включала по крайней мере некоторое подмножество системной функциональности, которое предположительно (исходя из результатов модульного тестирования) работает корректно. Системное тестирование предполагает своего рода контрольную проверку: каждый модуль программы работает нормально (он прошел модульные тесты), но работает ли программа *в целом*? Если работа каждого модуля соответствует ожиданиям, но система не работает, то очевидно, что проблема заключена в интеграции модулей — организации

[1] Используемая в этой области терминология несколько размыта и может сбивать с толку: термины наподобие *dummies* (пустышки), *fakes* (куклы), *spies* (тестовые шпионы), *mocks* (ложные объекты), *stubs* (заглушки) и *test doubles* (дублеры) используются разными авторами в разном смысле. Для ознакомления с одним из авторитетных подходов к этому вопросу (хотя и не совпадающим в точности с тем, который используется нами) рекомендуем обратиться к статье Мартина Фаулера (`https://martinfowler.com/articles/mocksArentStubs.html`). См. также описание соответствующих терминов в Википедии (`https://en.wikipedia.org/wiki/Test_double`).

их совместной работы. По этой причине системное тестирование также называют *интеграционным тестированием.*

Системное тестирование напоминает выполнение программы в производственных условиях, за исключением того, что вы заранее фиксируете входные данные, поэтому любую обнаруженную проблему можно легко воспроизвести. Стоимость ошибок при системном тестировании ниже, чем при производственном применении, поскольку выходные данные тестируемой системы не используются для принятия решений, организации обслуживания клиентов, управления внешними системами и т.п. Вместо этого они систематически сравниваются с выходными данными, которые система *должна* производить при известных входных данных. Целью такого тестирования является обнаружение дешевым и воспроизводимым способом расхождений между тем, что программа *должна делать*, и тем, что она *фактически делает*.

Ошибки, обнаруженные при системном тестировании (точно так же, как и ошибки, обнаруженные в процессе производственной эксплуатации системы), могут вскрыть некоторые дефекты в модульных тестах, а также дефекты в коде. Модульное тестирование могло оказаться недостаточно полным: тестирование модуля могло не охватить всю необходимую функциональность. В этом случае модульные тесты должны быть усилены. Сделайте это *до того*, как будете вносить изменения, устраняющие проблему, после чего выполните усиленные модульные тесты для подтверждения того, что теперь они способны выявить проблему. Затем устраните проблему и вновь выполните модульные тесты, чтобы удостовериться в том, что проблему действительно удалось разрешить.

Наилучшая практика устранения ошибок

Наилучшей практикой является применение одного из приемов TDD, который мы настоятельно рекомендуем использовать: никогда не приступайте к исправлению ошибки, не добавив предварительно модульные тесты, позволяющие обнаружить эту ошибку. Такая практика является отличной дешевой страховкой от *регрессионных ошибок* (`https://ru.wikipedia.org/wiki/Регрессионное_тестирование`).

Часто ошибки, обнаруженные в процессе системного тестирования, помогают выявить проблемы в коммуникации между членами команды разработчиков2: модуль корректно реализует некую функциональность, тогда другой модуль ожидает другую функциональность. Проблему такого рода (строго говоря, интеграционную проблему) трудно обнаружить в процессе модульного тестирования. Хорошая практика разработки предполагает частое выполнение модульных тестов, потому критически важно, чтобы они выполнялись быстро. По этой причине весьма существенно, чтобы на протяжении фазы модульного тестирования каждый модуль мог предполагать, что другие модули работают корректно и так, как ожидалось.

[2] Частично это объясняется тем, что структура системы стремится отражать структуру организации, как утверждает закон Конвея (`http://www.melconway.com/Home/Conways_Law.html`).

Модульные тесты, выполняемые на достаточно поздних стадиях разработки, могут выявить интеграционные проблемы, если система имеет иерархическую структуру — распространенный и разумный принцип организации. При такой архитектуре модули самого низкого уровня не зависят ни от каких других модулей (за исключением библиотечных, которые можно считать корректными), поэтому модульного тестирования низкоуровневых модулей, если они проходят тесты, достаточно для того, чтобы убедиться в их корректности. Высокоуровневые модули зависят от низкоуровневых, а это означает, что они зависят также от правильного понимания того, какой функциональности ожидает каждый модуль и какая функциональность предоставляется. Полное модульное тестирование высокоуровневых модулей (с использованием истинных низкоуровневых модулей, а не заглушек) проверяет интерфейсы между модулями, а также собственный код высокоуровневых модулей.

Следовательно, модульное тестирование высокоуровневых модулей выполняется двояким образом. Вы выполняете тесты с заглушками для низкоуровневых модулей на ранних стадиях разработки, когда низкоуровневые модули еще не готовы, или впоследствии, когда вам необходимо проверить лишь корректность модулей более высокого уровня. На более поздних стадиях разработки вы также регулярно выполняете модульное тестирование высокоуровневых модулей, используя истинные низкоуровневые модули. Благодаря этому вы проверяете корректность всей подсистемы, от верхних уровней до нижних. Даже в этом благоприятном случае вы *все равно* должны выполнить системные тесты, чтобы быть уверенным в том, что функциональность системы обеспечивается в полном объеме и проверена и ни один из интерфейсов между модулями не пропущен.

Системное тестирование аналогично обычному выполнению программы. Специальная поддержка понадобится вам лишь для того, чтобы предоставить известные входные значения и извлечь выходные значения для сравнения с ожидаемыми. Это можно легко сделать в случае программ, выполняющих операции ввода-вывода над файлами, но труднее сделать в случае программ, операции ввода-вывода которых зависят от графического интерфейса пользователя (GUI), сети или других средств коммуникации с внешними объектами. Чтобы имитировать подобные внешние объекты, одновременно обеспечивая предсказуемость их поведения и полную наблюдаемость, обычно требуется зависящая от платформы инфраструктура. Другим полезным элементом вспомогательной инфраструктуры для системного тестирования является *фреймворк тестирования*, позволяющий автоматизировать выполнение системных тестов, включая регистрацию в журнале всех удачных и неудачных тестов. Такие фреймворки также облегчают тестировщикам подготовку наборов известных входных данных и соответствующих ожидаемых выходных результатов.

Существуют как бесплатные, так и коммерческие программы, предназначенные для этих целей, но они не зависят от того, какие языки программирования используются в тестируемой системе. Системное тестирование сродни тому, для чего существует классическое понятие *тестирование по методу черного ящика*, т.е. тестирование, которое не зависит от реализации системы, подвергаемой тестам (а значит, в том числе и от

языка программирования, используемого для реализации). Вместо этого тестирующие фреймворки обычно зависят от платформы, используемой для тестирования, ввиду платформозависимости выполняемых ими задач: выполнение программ с заданными входными данными, получение выходных данных и, в частности, имитация и получение данных через GUI, сеть и межпроцессное взаимодействие. Поскольку фреймворки для системного тестирования зависят от платформы, а не от языков программирования, далее в книге они не обсуждаются. С подробным списком инструментов тестирования для Python можно ознакомиться на вики-странице Python по следующему адресу:

`https://wiki.python.org/moin/PythonTestingToolsTaxonomy`

Модуль `doctest`

Модуль `doctest` облегчит вам создание качественных примеров в строках документирования кода, поскольку он проверяет, действительно ли выполнение примеров приводит к результатам, указанным в строках документирования. Модуль `doctest` распознает примеры, находя в строке документирования интерактивную подсказку (`'>>> '`), за которой в той же строке следует инструкция на языке Python, а в следующей строке (строках) — ожидаемый результат выполнения этой инструкции. По мере разработки модуля обновляйте строки документирования и дополняйте их новыми примерами. Выработайте привычку добавлять примеры в строки документирования, как только часть модуля (например, функция) достигает состояния частичной или полной готовности. Импортируйте модуль в интерактивном сеансе и используйте разработанные части кода для того, чтобы привести примеры, представляющие типичные случаи, а также случаи удачного и неудачного выполнения. Исключительно для этих целей используйте инструкции `from module import *`, чтобы не снабжать префиксом модуля каждое предоставляемое им имя в примерах. Скопируйте и вставьте интерактивный сеанс в строку документирования, воспользовавшись текстовым редактором, исправьте любые возможные неточности, и на этом подготовка примеров почти закончена.

Теперь ваша документация снабжена примерами, и читателям будет легче ее понять, особенно если вы выбрали удачный набор примеров и сопроводили его пояснительным текстом. Убедитесь в том, что строками документирования с примерами снабжены не только модуль в целом, но и все экспортируемые им функции, классы и методы. Если хотите, можете опустить функции, классы и методы, имена которых начинаются с символа подчеркивания (`_`), поскольку (на что указывают сами эти имена) они относятся к деталям частной реализации. Модуль `doctest` игнорирует их по умолчанию, что должно распространяться и на читателей вашего модуля.

Согласуйтесь с реальностью

Примеры, не согласующиеся с тем, как работает ваш код, более чем бесполезны. Документация и комментарии полезны лишь в том случае, если они соответствуют реальности. Их несоответствие этому требованию может привести к серьезному ущербу.

Документация и комментарии часто устаревают по мере внесения изменений в код и тем самым дезинформируют читателя, не облегчая, а затрудняя чтение кода. Лучше вообще не иметь строк документирования и комментариев, при всей неудачности такого решения, чем иметь такие, которые лишь вводят в заблуждение. Модуль `doctest` может оказать вам содействие в подготовке примеров в строках документирования. Неудачный результат выполнения модуля `doctest` должен побудить вас к пересмотру строки документирования, содержащей неудачные примеры, напоминая о необходимости постоянного обновления всей строки документирования.

В конце исходного кода модуля вставьте следующий фрагмент.

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Этот код вызовет функцию `testmod`, если модуль выполняется в качестве основной программы. Функция `testmod` проверяет строки документирования (относящиеся к модулю в целом, а также к его общедоступным функциям, классам и методам). Она находит в каждой строке документирования все примеры (путем поиска вхождений подсказки интерпретатора `'>>> '`, которой могут предшествовать пробелы) и выполняет каждый пример. Она проверяет, чтобы результаты каждого примера совпадали с теми, которые предоставлены в строке документирования сразу же за каждым примером. В случае возникновения исключений функция `testmod` игнорирует стек вызовов и проверяет лишь совпадение ожидаемых и наблюдаемых сообщений об ошибках.

В случае нормального прохождения тестов функция `testmod` завершает выполнение без вывода каких-либо сообщений. В противном случае она выводит подробные сообщения о примерах, которые не удалось выполнить, отображая ожидаемый и фактический вывод. Листинг 16.1 иллюстрирует применение модуля `doctest` к модулю `mod.py`.

Листинг 16.1. Использование модуля `doctest`

```
"""
Этот модуль предоставляет единственную функцию reverse_words,
которая обращает порядок слов в строке.

>>> reverse_words('four score and seven years')
'years seven and score four'
>>> reverse_words('justoneword')
'justoneword'
>>> reverse_words('')
''

Функцию reverse_words необходимо вызывать с одним
строковым аргументом:
```

```
>>> reverse_words()
Traceback (most recent call last):
...
TypeError: reverse_words() takes exactly 1 argument (0 given)
>>> reverse_words('one', 'another')
Traceback (most recent call last):
...
TypeError: reverse_words( ) takes exactly 1 argument (2 given)
>>> reverse_words(1)
Traceback (most recent call last):
...
AttributeError: 'int' object has no attribute 'split'
>>> reverse_words(u'however, unicode is all right too') # v2 check
u'too right all is unicode however,'

В качестве побочного эффекта функция reverse_words убирает
все лишние пробелы:

>>> reverse_words('with    redundant    spacing')
'spacing redundant with'

"""
def reverseWords(astring):
    words = astring.split()
    words.reverse()
    return ' '.join(words)

if __name__=='__main__':
    import doctest
    doctest.testmod()
```

В строке документирования этого модуля стеки вызовов заменены многоточиями (...): это неплохая практика, поскольку модуль `doctest` игнорирует трассировочную информацию, которая не имеет никакой информативной ценности для примеров, не прошедших тест. За исключением этого, строка документирования получена копированием и вставкой интерактивного сеанса, дополненного поясняющим текстом и пустыми строками для улучшения его читаемости. Сохраните этот исходный код в файле *mod.py*, а затем выполните его с помощью команды **`python mod.py`**. В данном случае вывод отсутствует, что означает успешное выполнение всех примеров. Можете воспользоваться командой **`python mod.py -v`** для получения отчета по всем тестам и завершающей текстовой сводки. Наконец, измените результаты примеров в строке документирования модуля, чтобы сделать их некорректными и чтобы увидеть, какие сообщения выводит модуль `doctest` для ошибочных примеров.

Несмотря на то что модуль `doctest` не проектировался в качестве универсального инструмента модульного тестирования, возникает соблазн использовать его для этой цели. Модульное тестирование в Python рекомендуется выполнять с помощью

модуля unittest, который обсуждается в разделе "Модуль unittest". Однако настройка модульного тестирования с помощью модуля doctest осуществляется быстрее и с меньшими усилиями, поскольку в основном это сводится к копированию и вставке текста из интерактивного сеанса. Если вам приходится сопровождать модуль, для которого отсутствуют модульные тесты, то вставка в такой модуль тестов задним числом с использованием модуля doctest является разумным компромиссом (хотя в конечном счете вы должны запланировать обновление модуля до полноценных тестов, использующих модуль unittest). Лучше иметь модульные тесты на основе строки документирования, чем вообще не иметь средств тестирования, что вполне может случиться, если вы решите, что надлежащая заблаговременная подготовка тестов с помощью модуля unittest займет слишком много времени[3].

Если вы решите использовать модуль doctest для модульного тестирования, старайтесь не заполнять строки документирования излишними тестами. Это только навредит делу, поскольку строки документирования удлинятся и их будет трудно читать. Пытайтесь удерживать объем и типы примеров в строгих рамках, достаточных для предоставления необходимой пояснительной информации, чтобы тесты не выпячивались на передний план. Вместо этого помещайте дополнительные тесты в глобальную переменную модуля, словарь __test__. В этом словаре ключами являются строки, используемые в качестве произвольных названий тестов, а соответствующими значениями — строки, которые модуль doctest выбирает и использует точно так же, как и строки документирования. Значениями в словаре __test__ также могут быть функции и объекты классов, и в этом случае модуль doctest проверяет свои строки документирования в поиске тестов, подлежащих выполнению. Эта последняя возможность предоставляет очень удобный способ выполнения модуля doctest для объектов с закрытыми именами, которые doctest по умолчанию пропускает.

Кроме того, модуль doctest предоставляет две функции, возвращающие экземпляры класса unittest.TestSuite, основанного на строках документирования, чтобы вы имели возможность интегрировать подобные тесты в фреймворки тестирования, основанные на модуле unittest. Для этой развитой функциональности существует онлайн-документация (https://docs.python.org/2/library/doctest.html#unittest-api).

Модуль unittest

Модуль unittest — это предназначенная специально для Python версия фреймворка автоматизированного тестирования, первоначально разработанного Кентом Беком для языка Smalltalk. Аналогичные широко распространенные версии этого фреймворка существуют для многих других языков программирования (например,

[3] Однако старайтесь тщательно продумывать, для каких именно целей вы используете модуль *doctest* в каждом конкретном случае. Здесь уместно процитировать Питера Норвига, весьма определенно высказавшегося по этому поводу: "Знайте, чего добиваетесь. Если вы целитесь одновременно в две мишени, то, скорее всего, промажете в обе".

пакет JUnit для Java), и для этого семейства фреймворков часто используют собирательное название xUnit. Работая с модулем `unittest`, не объединяйте в одном файле тестирующий и тестируемый код: записывайте модуль теста для каждого тестируемого модуля в отдельный файл. Обычно следуют общепринятому соглашению, в соответствии с которым модуль теста получает имя тестируемого модуля, снабженное префиксом, например `'test_'`, и помещается в подкаталог `test`, находящийся в одном каталоге с исходным кодом. Например, модуль теста для модуля *mod.py* может храниться в файле *test/test_mod.py*. Простая и последовательная система именования файлов упрощает написание и сопровождение вспомогательных сценариев, предназначенных для поиска и выполнения модульных тестов для всего пакета.

Разделение исходного кода модуля и кода модульного тестирования также позволяет упростить рефакторинг модуля, в том числе и той его возможной части, которая реализует какую-либо функциональность на языке C, не затрагивая тестирующий код. Знание того, что содержимое файла *test_mod.py* остается неизменным, независимо от каких бы то ни было изменений, внесенных в файл *mod.py*, укрепляет вашу уверенность в том, что успешное прохождение теста, содержащегося в файле *test_mod.py*, является свидетельством корректной работы измененного модуля.

Тестирующий модуль определяет один или несколько подклассов класса `TestCase`, определенного в модуле `unittest`. Каждый подкласс предоставляет один или несколько *сценариев (вариантов) тестирования* (test cases) путем определения *тестовых методов*, т.е. методов, которые вызываются без аргументов и имена которых начинаются с префикса `test`.

Подкласс может переопределить метод `setUp`, который вызывается фреймворком для подготовки нового экземпляра непосредственно перед выполнением каждого тестового сценария, и метод `tearDown`, который фреймворк вызывает для выполнения завершающих операций по освобождению ресурсов сразу же после выполнения теста. Эта пара вызовов `setUp-tearDdown` (и в целом код, управляющий конфигурированием зависимостей тестов), называется *тестовой фикстурой* (test fixture).

В необходимых случаях включайте в метод `setUp` вызов метода `addCleanup`

Если метод `setUp` распространяет исключение, то метод `tearDown` не выполняется. Поэтому, если метод `setUp` подготавливает ресурсы, которые необходимо освобождать по завершении теста, и на некоторых подготовительных этапах могут возбуждаться неперехватываемые исключения, то метод `setUp` не может рассчитывать на очистку ресурсов, выполняемую методом `tearDown`. Вместо этого после каждого успешно выполненного подготовительного шага следует вызывать метод `self.addCleanup(f, *a, **k)`, передавая ему вызываемый объект `f`, выполняющий соответствующие завершающие операции, вместе с необязательными позиционными и именованными аргументами

f. В этом случае `f(*a, **k)` будет вызываться после каждого тестового сценария (после вызова метода `tearDown`, если метод `setUp` не распространяет исключений, но безусловно, даже если метод `setUp` распространяет исключение), что гарантирует очистку ресурсов при любых условиях.

Каждый тестовый сценарий вызывает для объекта `self` методы класса `TestCase`, имена которых начинаются с префикса `assert`, для установления условий, которым должен удовлетворить тест. Модуль `unittest` выполняет методы тестового сценария подкласса `TestCase` в произвольном порядке, каждый раз с новым экземпляром подкласса, вызывая метод `setUp` непосредственно перед выполнением каждого тестового сценария и метод `tearDown` сразу же по завершении тестового сценария.

Модуль `unittest` также предоставляет другие возможности, такие как группирование тестовых сценариев в *тестовые наборы* (test suites), конфигурирование окружения теста на уровне класса или модуля, обнаружение тестов, а также другие средства расширенной функциональности. Эти дополнительные средства понадобятся вам лишь в том случае, если вы определяете специализированный фреймворк модульного тестирования или, по крайней мере, структурируете сложные процедуры тестирования для столь же сложных пакетов. В большинстве случаев сведений, изложенных в этом разделе, вам будет достаточно для того, чтобы организовать эффективное систематическое выполнение модульного тестирования. В листинге 16.2 продемонстрировано использование модуля `unittest` для организации модульного тестирования модуля *mod.py* из листинга 16.1. В этом примере с помощью модуля `unittest` выполняются те же тесты, которые в листинге 16.1 выполняются с помощью модуля `doctest`.

Листинг 16.2. Использование модуля `unittest`

```
""" Этот модуль тестирует функцию reverseWords,
предоставленную модулем mod.py. """
import unittest
import mod

class ModTest(unittest.TestCase):

    def testNormalCaseWorks(self):
        self.assertEqual(
            mod.reverse_words('four score and seven years'),
            'years seven and score four')

    def testSingleWordIsNoop(self):
        self.assertEqual(
            mod.reverse_words('justoneword'),
            'justoneword')
```

```
    def testEmptyWorks(self):
        self.assertEqual(mod.reverse_words(''), '')

def testRedundantSpacingGetsRemoved(self):
    self.assertEqual(
        mod.reverse_words('with redundant spacing'),
        'spacing redundant with')

def testUnicodeWorks(self):
    self.assertEqual(
        mod.reverse_words(u'unicode is all right too'),
        u'too right all is unicode')

def testExactlyOneArgumentIsEnforced(self):
    self.assertRaises(TypeError, mod.reverse_words)
    with self.assertRaises(TypeError):
        mod.reverse_words('one', 'another')

def testArgumentMustBeString(self):
    with self.assertRaises((AttributeError,TypeError)):
        mod.reverse_words(1)

if __name__=='__main__':
    unittest.main()
```

Выполнив этот сценарий с помощью команды **python test/test_mod.py** (или эквивалентной ей команды **python -mtest.test_mod**), вы получите несколько более длинный вывод, чем в случае выполнения модуля `doctest` с помощью команды **python mod.py**, как в листинге 16.1. Для каждого выполняемого тестового сценария тест *test_mod.py* выводит символ точки (.), а затем разделительную линию, состоящую из символов тире, строку отчета о тестах вида "Ran 7 tests in 0.110s" и завершающую строку "OK", если каждый тест был успешно пройден.

Каждый метод тестового сценария выполняет один или несколько вызовов методов, имена которых начинаются с префикса `assert`. В приведенном примере метод `testExactlyOneArgumentIsEnforced` — единственный, который содержит два таких вызова. В более сложных случаях многократные вызовы методов `assert` из одного метода тестового сценария не являются редкостью.

Даже в столь простом случае, как этот, можно отметить один незначительный аспект, демонстрирующий, что модуль `unittest` обладает намного большими возможностями и гибкостью в отношении модульного тестирования, чем модуль `doctest`. В методе `testArgumentMustBeString` мы передаем методу `assertRaises` в качестве аргумента кортеж классов исключений, означающий, что мы принимаем любое из этих исключений. Поэтому тестовый сценарий *test_mod.py* принимает в качестве допустимых несколько реализаций *mod.py*. Он принимает реализацию из листинга 16.1, которая пытается вызвать метод `split`

для своего аргумента и поэтому возбуждает исключение AttributeError, если вызывается с аргументом, не являющимся строкой. Однако она также принимает другую гипотетическую реализацию, которая возбуждает исключение TypeError, если вызывается с аргументом неподходящего типа. Можно написать код для реализации подобной функциональности, использующей модуль doctest, но такой код будет громоздким и неочевидным, в то время как модуль unittest обеспечивает простое и естественное решение.

Такого рода гибкость чрезвычайно важна для реальных модульных тестов, которые в определенном смысле выступают в качестве исполняемых спецификаций по отношению к своим модулям. Мысля пессимистически, вы могли бы счесть потребность в гибкости тестов недостатком, свидетельствующим о не полностью продуманном интерфейсе тестируемого кода. Однако лучше исходить из того, что в интересах реализатора интерфейс должен определяться достаточно гибким в разумных пределах: в условиях *X* (в этом примере — передача аргумента недопустимого типа функции reverseWords) допускается возможность любого из двух исходов (возбуждение исключения AttributeError или TypeError).

Таким образом, корректны реализации с любым из двух вариантов поведения, и реализатор может осуществлять выбор между ними, исходя из соображений производительности или ясности кода. Рассматривая модульные тесты как исполняемые спецификации их модулей (современная точка зрения, лежащая в основе методики разработки через тестирование), а не как тестирование методом белого ящика, строго ограниченным рамками конкретной реализации (в соответствии с некоторыми традиционными таксономиями тестирования), вы еще больше укрепитесь в мысли о том, что тесты — жизненно необходимая часть процесса разработки программного обеспечения.

Класс TestCase

Используя модуль unittest, вы пишете тестовые сценарии, расширяя класс TestCase и добавляя методы без аргументов, с именами, начинающимися с префикса test. В свою очередь, эти методы, управляющие выполнением тестов (тестовые методы), вызывают методы, наследуемые вашим классом от класса TestCase, имена которых начинаются с префикса assert. Назначением этих методов является указание условий, используемых в качестве критериев успешного прохождения теста.

Класс TestCase также определяет два метода, которые ваш класс может переопределить с целью группирования действий, выполняемых непосредственно перед вызовом каждого тестового метода и сразу же после его завершения. Предлагаемая классом TestCase функциональность этим не ограничивается, но остальная ее часть вам может понадобиться лишь в том случае, если вы разрабатываете фреймворки тестирования или решаете другие задачи, выходящие за рамки обычного тестирования. Наиболее часто используемые методы экземпляров (*t*) класса TestCase описаны ниже.

assertAlmostEqual `t.assertAlmostEqual(first, second, places=7, msg=None)`

Генерирует ошибку и выводит строку *msg*, если `first!=second` с точностью до *places* знаков после десятичной точки; в противном случае не выполняет никаких действий. При сравнении чисел с плавающей точкой этот метод почти всегда является более предпочтительным, чем метод `assertEqual`, поскольку в силу погрешностей, обусловленных способом представления чисел с плавающей точкой, обычно такие числа могут быть лишь приблизительно равными.

assertEqual `t.assertEqual(first, second, msg=None)`

Генерирует ошибку и выводит строку *msg*, если `first!=second`; в противном случае не выполняет никаких действий.

assertFalse `t.assertFalse(condition, msg=None)`

Генерирует ошибку и выводит строку *msg*, если *condition* имеет истинное значение; в противном случае не выполняет никаких действий.

assertNotAlmostEqual `t.assertNotAlmostEqual(first, second, places=7, msg=None)`

Генерирует ошибку и выводит строку *msg*, если `first==second` с точностью до *places* знаков после десятичной точки; в противном случае не выполняет никаких действий.

assertNotEqual `t.assertNotEqual(first, second, msg=None)`

Генерирует ошибку и выводит строку *msg*, если `first==second`; в противном случае не выполняет никаких действий.

assertRaises `t.assertRaises(exceptionSpec, callable, *args, **kwargs)`

Вызывает `callable(*args,**kwargs)`. Генерирует ошибку, если этот вызов не возбуждает никаких исключений. Если вызов возбуждает исключение, не соответствующее аргументу *exceptionSpec*, то метод `assertRaises` распространяет исключение. Если вызов возбуждает исключение, соответствующее аргументу *exceptionSpec*, метод `assertRaises` не выполняет никаких действий. Аргументом *exceptionSpec* может быть класс исключения или кортеж из классов, как в случае первого аргумента предложения `except` в инструкции `try/except`.

Обычно предпочтительнее использовать метод `assertRaises` в качестве *менеджера контекста*, т.е. в инструкциях следующего вида:

```
with self.assertRaises(exceptionSpec):
    ...блок кода...
```

В приведенном фрагменте кода вместо простого вызова объекта *callable* с определенными аргументами выполняется "блок кода", выделенный отступом в теле инструкции `with`. Ожидаемым результатом (не приводящим к генерации ошибки) является возбуждение в блоке кода исключения, соответствующего заданной спецификации исключений (класс исключения или кортеж классов). Этот альтернативный подход предпочтителен в силу большей общности, естественности и удобочитаемости.

assertTrue

`t.assertTrue(`*condition*`, `*msg*`=None)`

Генерирует ошибку и выводит строку *msg*, если *condition* имеет ложное значение; в противном случае не выполняет никаких действий. Не используйте этот метод, если можете задействовать более специфический метод наподобие `assertEqual`: специфические методы предоставляют более конкретную информацию об ошибках.

fail

`t.fail(`*msg*`=None)`

Генерирует безусловную ошибку и выводит строку *msg*. Ниже приведен соответствующий пример.

```
if not complex_check_if_its_ok(some, thing):
    self.fail('Complex checks failed on {}, {}'
    .format(some, thing))
```

setUp

`t.setUp()`

Фреймворк вызывает метод `t.setUp()` непосредственно перед вызовом тестового метода. Метод `setUp`, определенный в классе `TestCase`, не выполняет никаких действий. Метод `setUp` существует лишь для того, чтобы вы могли переопределить его в своем классе, если для каждого теста должны быть выполнены некоторые подготовительные действия.

tearDown

`t.tearDown()`

Фреймворк вызывает метод `t.tearDown()` сразу же после выполнения тестового метода. Метод `tearDown`, определенный в классе `TestCase`, не выполняет никаких действий. Метод `tearDown` существует лишь для того, чтобы вы могли переопределить его в своем классе, если для каждого теста должны быть выполнены некоторые завершающие действия по очистке ресурсов.

Кроме того, экземпляр `TestCase` поддерживает список *функций завершения* (clean-up functions), работающий по принципу "последним пришел, первым ушел" (стек LIFO). Если код в одном из ваших тестов (или в методе `setUp`) требует выполнения неких завершающих операций, вызовите метод `addCleanup`, передав ему функцию завершения *f* вместе с необходимыми ей позиционными и именованными аргументами. Для выполнения стека функций завершения можно вызвать метод

doCleanups, однако фреймворк сам вызывает метод doCleanups после выполнения метода tearDown. Сигнатуры двух методов завершения описаны ниже.

addCleanup *t*.addCleanup(*func*, *a, **k)

Добавляет (*func*, *a*, *k*) в конец списка функций завершения.

doCleanups *t*.doCleanups()

Выполняет все функции завершения, находящиеся в стеке, если таковые имеются. В основном (если не учитывать проверку на отсутствие ошибок и вывод отчета) этот метод эквивалентен следующему коду для гипотетического стека *self.list_of_cleanups*:

```
while self.list_of_cleanups:
    func, a, k = self.list_of_cleanups.pop()
    func(*a, **k)
```

Модульные тесты с большими объемами данных

Модульные тесты должны выполняться быстро: они запускаются достаточно часто по мере разработки программного продукта. Поэтому тестирование каждого существенного изменения, вносимого в код модуля, по возможности должно выполняться с использованием небольших объемов данных. Это ускорит выполнение тестов и позволит внедрять данные в исходный код теста. Если вы тестируете функцию, использующую объект файла для чтения и записи данных, используйте экземпляр класса io.TextIO в случае текстового Unicode-файла (экземпляр класса io.BytesIO в случае двоичного файла; см. раздел "Файлы в памяти: функции io.StringIO и io.BytesIO" в главе 10) для получения "файла" данных, хранящихся в памяти: это работает быстрее, чем запись данных на диск, и не требует выполнения рутинных операций по очистке ресурсов (удаление файлов на диске по завершении тестирования).

Изредка встречаются ситуации, когда невозможно испытать функциональность модуля без предоставления (и/или сравнения) требуемых для его работы данных в объемах, значительно превышающих те разумные пределы, которые еще позволяют внедрять данные в исходный код теста. В подобных случаях тест должен опираться на вспомогательные внешние файлы, в которых хранятся как данные, предоставляемые тестируемому модулю, так и данные, с которыми должны сравниваться выходные результаты. Но даже и в этом случае обычно лучше использовать в тестах экземпляры вышеупомянутых классов, чем прямые дисковые операции ввода-вывода. Более того, мы настоятельно рекомендуем использовать заглушки для тестирования модулей, взаимодействующих с такими внешними объектами, как базы данных, GUI или другие программы. Контролировать все аспекты теста гораздо легче, если вместо внешних объектов используются заглушки. Кроме того, нелишне еще раз подчеркнуть, что скорость выполнения тестов также имеет значение, а она будет значительно выше, если вместо выполнения реальных операций они имитируются в заглушках.

Используйте в тестах воспроизводимую случайность, предоставляя затравочные значения

Если в вашем коде используются псевдослучайные числа (см., например, раздел "Модуль `random`" в главе 15), вы можете упростить тестирование, обеспечив *воспроизводимость* "случайного" поведения. В частности, позаботьтесь о том, чтобы ваши тесты позволяли без труда осуществлять вызовы `random.seed` с известным аргументом, благодаря чему генерация случайных чисел станет полностью предсказуемой. То же самое относится к ситуациям, когда вы используете псевдослучайные числа для конфигурирования тестов путем генерации случайных входных данных. Такая генерация по умолчанию должна выполняться с известным затравочным значением, которое будет использоваться в большинстве тестов, одновременно предоставляя дополнительную возможность изменения затравочного значения для специфических методик тестирования, таких как *фаззинг* (`https://ru.wikipedia.org/Фаззинг/`).

Отладка

Учитывая скорость цикла разработки в Python, самым эффективным способом отладки зачастую является редактирование кода для вывода нужной информации в ключевых точках. В Python имеется множество способов, позволяющих вашему коду исследовать собственное состояние для извлечения отладочной информации. В частности, возможность такого исследования, известного как *рефлексия* или *интроспекция*, обеспечивают модули `inspect` и `traceback`.

Для отображения полученной отладочной информации часто удобно использовать модуль `print` (во многих случаях для этого также неплохо использовать модуль `pprint`, рассмотренный в разделе "Модуль `pprint`" в главе 8). Еще лучше записывать отладочную информацию в журнальный файл. Журналирование (протоколирование) информации особенно полезно в случае программ, которые выполняются без вмешательства пользователя (например, серверных программ). Отображение отладочной информации ничем не отличается от отображения любой другой информации, о чем шла речь в главе 10. Журналирование подобной информации аналогично записи в файл, однако стандартная библиотека Python предоставляет модуль `logging` (см. раздел "Пакет `logging`" в главе 5), облегчающий задачу журналирования. Повторное связывание атрибута `excepthook` модуля `sys` (табл. 7.3) позволяет вашей программе записывать в журнал информацию об ошибках непосредственно перед прекращением работы программы с распространением исключения.

Python также предоставляет расширения (хуки, перехватчики), обеспечивающие возможность интерактивной отладки. Модуль `pdb` предоставляет простой

интерактивный отладчик, работающий в текстовом режиме. Другие, более мощные интерактивные отладчики для Python входят в состав интегрированных сред разработки (IDE), таких как IDLE и различные коммерческие продукты (см. раздел "Среды разработки Python" в главе 2). Эти отладчики в данной книге не рассматриваются.

Прежде чем приступить к отладке

Прежде чем приступать к длительному процессу отладки, тщательно проверьте свой исходный код на языке Python с помощью инструментов, о которых шла речь в главе 2. Инструменты такого рода позволяют выявить лишь некоторое подмножество дефектов, но они сделают это быстрее, чем сделали бы вы в режиме интерактивной отладки, так что их использование с лихвой окупит себя.

Кроме того (опять-таки, еще до начала сеанса отладки), убедитесь в том, что код достаточно плотно охвачен модульными тестами (см. раздел "Модуль `unittest`"). Как уже упоминалось в этой главе, обнаружив дефект, не спешите устранять его и добавьте в свой набор модульных тестов (или, если это необходимо, в набор системных тестов) один или два теста, которые, будь они использованы с самого начала, обнаружили бы данный дефект. Затем вновь выполните тесты в подтверждение того, что теперь они обнаруживают и изолируют дефект, и лишь после этого приступайте к его устранению. Регулярно следуя этой процедуре, вы получите намного лучший набор тестов, научитесь писать более надежные тесты и всегда будете уверены в корректности своего кода.

Помните, что даже с учетом всех тех возможностей, которые предлагают Python, его стандартная библиотека и ваши излюбленные IDE, отладка по- прежнему остается *трудной* задачей. Держите это в уме еще до того, как начнете проектировать программу и писать код: пишите и выполняйте множество тестов, одновременно заботясь о *простоте* проекта и кода, чтобы свести объем отладки, которая потребуется в будущем, к минимуму! Здесь уместно привести классический совет, данный Брайаном Керниганом по этому поводу: "Отладить код вдвойне сложнее, чем написать. И если в написание кода вы вложили всю свою смекалку, то для его отладки вам ее по определению не хватит". Отчасти именно этим можно объяснить тот факт, что слово "хитроумный" не воспринимается как положительная характеристика кода или программиста, пишущего такой код.

Модуль `inspect`

Модуль `inspect` предоставляет функции, позволяющие получать информацию об объектах любого типа, включая стек вызовов (который содержит записи о вызовах всех функций, выполняющихся в данный момент) и файлы с исходным кодом. Описание наиболее часто используемых функций модуля `inspect` приведено ниже.

getargspec, formatargspec

`getargspec(f)`

Признана устаревшей в версии v3: все еще работает в версиях Python 3.5 и 3.6, но будет удалена в одной из будущих версий. Для интроспекции вызываемых объектов в версии v3 используйте функцию `inspect.signature(f)` и результирующий экземпляр класса `inspect.Signature` (раздел "Интроспекция вызываемых объектов в версии v3").

Здесь *f* — объект функции. Функция `getargspec` возвращает именованный кортеж из четырех элементов: (*args*, *varargs*, *keywords*, *defaults*). Элемент *args* — это последовательность имен параметров *f*. Элемент *varargs* — это имя специального параметра вида **a* или `None`, если *f* не имеет такого параметра. Элемент *keywords* — это имя специального параметра вида ***k* или `None`, если *f* не имеет такого параметра. Элемент *defaults* — это кортеж значений по умолчанию для аргументов *f*. Вы сможете самостоятельно сделать заключения относительно других деталей сигнатуры *f* на основании результатов `getargspec`: *f* имеет `len(`*args*`)-len(`*defaults*`)` обязательных позиционных аргументов, а имена необязательных аргументов *f* представлены строками, являющимися элементами среза списка `args[-len(`*defaults*`):]`.

Функция `formatargspec` принимает от одного до четырех аргументов, совпадающих с элементами именованного кортежа, возвращаемого функцией `getargspec`, и возвращает отформатированную строку, представляющую эти значения. Таким образом, вызов `formatargspec(*getargspec(f))` возвращает строку с параметрами *f* (называемую *сигнатурой f*) в круглых скобках, как в инструкции `def`, создавшей *f*.

```
import inspect
def f(a,b=23,**c): pass
print(inspect.formatargspec(*inspect.getargspec(f)))
# вывод: (a, b=23, **c)
```

getargvalues, formatargvalues

`getargvalues(f)`

Признана устаревшей в версии v3: все еще работает в версиях Python 3.5 и 3.6, но будет удалена в одной из будущих версий. Для интроспекции вызываемых объектов в версии v3 используйте функцию `inspect.signature(f)` и результирующий экземпляр класса `inspect.Signature` (раздел "Интроспекция вызываемых объектов в версии v3").

Здесь *f* — это объект фрейма (кадра стека), являющийся, например, результатом вызова функции `_getframe` из модуля `sys` (см. раздел "Тип `frame`" в главе 13) или функции `currentframe` из модуля `inspect`. Функция `getargvalues` возвращает именованный кортеж из четырех элементов: (*args*, *varargs*, *keywords*, *locals*). Элемент *args* — это последовательность имен параметров функции фрейма *f*. Элемент *varargs* — это имя специального параметра вида **a* или `None`, если функция фрейма *f* не имеет такого параметра.

Элемент *keywords* — это имя специального параметра вида **k или None, если функция фрейма *f* не имеет такого параметра. Элемент *locals* — это словарь локальных переменных *f*. Поскольку, в частности, аргументы являются локальными переменными, то значение каждого аргумента может быть получено из словаря *locals* путем его индексирования именами параметров, соответствующих аргументам.

Функция formatargvalues принимает от одного до четырех аргументов, совпадающих с элементами именованного кортежа, возвращаемого функцией getargvalues, и возвращает отформатированную строку, представляющую эти значения. Вызов formatargvalues(*getargvalues(*f*)) возвращает строку с аргументами *f* в круглых скобках в именованном виде, в каком они используются в инструкции вызова, создавшего *f*.

```
def f(x=23): return inspect.currentframe()
print(inspect.formatargvalues(
  *inspect.getargvalues(f())))
# вывод: (x=23)
```

currentframe currentframe()

Возвращает объект фрейма (кадра стека) для текущей функции (функции, вызвавшей функцию currentframe). Например, вызов formatargvalues(*getargvalues(currentframe())) возвращает строку с аргументами вызывающей функции.

getdoc getdoc(*obj*)

Возвращает строку документирования для *obj* — многострочную строку с символами табуляции, расширенными до пробелов, и удаленными из каждой строки избыточными пробелами.

getfile, getsourcefile getfile(*obj*)

Возвращает имя файла, содержащего определение *obj*; возбуждает исключение TypeError, если не может определить этот файл. Например, getfile возбуждает исключение TypeError, если *obj* является встроенным объектом. Функция getfile возвращает имя двоичного или текстового файла с исходным кодом. Функция getsourcefile возвращает имя исходного файла и возбуждает исключение TypeError, если все, что удается найти, — это двоичный файл, а не соответствующий текстовый файл с исходным кодом.

getmembers getmembers(*obj*, *filter*=None)

Возвращает все атрибуты (элементы), как данные, так и методы (в том числе специальные методы) объекта *obj*, в виде отсортированного списка пар (*name*, *value*). Если аргумент *filter* не равен None, возвращает лишь атрибуты, для которых вызываемый объект *filter*, будучи вызванным со значением атрибута, возвращает истинное значение, как в следующем фрагменте кода:

```
sorted((n, v) for n, v in getmembers(obj) if
  filter(v))
```

getmodule `getmodule(obj)`

Возвращает объект модуля, содержащего определение объекта *obj*, или значение `None`, если его не удается найти.

getmro `getmro(c)`

Возвращает кортеж, состоящий из базовых классов и предков класса *c* в порядке разрешения методов. Аргумент *c* является первым элементом кортежа. Каждый класс встречается в кортеже только один раз.

```
class newA(object): pass
class newB(newA): passclass newC(newA): pass
class newD(newB, newC): pass
for c in inspect.getmro(newD): print(c.__name__, end=' ')
newD newB newC newA object
```

getsource, getsourcelines `getsource(obj)`

Возвращает многострочную строку, являющуюся исходным кодом объекта *obj*; возбуждает исключение `IOError`, если ее не удается определить или извлечь. Функция `getsourcelines` возвращает пару: первым элементом является исходный код объекта *obj* (список строк), а вторым — номер первой строки в файле.

isbuiltin, isclass, iscode, isframe, isfunction, ismethod, ismodule, isroutine `isbuiltin(obj)`

Каждая из этих функций принимает единственный аргумент *obj* и возвращает значение `True`, если тип *obj* соответствует типу, указанному в имени функции. Соответственно, допустимыми объектами являются следующие: встроенные (написанные на языке C) функции, объекты классов, объекты кода, объекты фреймов, написанные на языке Python функции (включая лямбда-выражения), методы, модули и — для функции `isroutine` — все методы или функции, написанные как на языке C, так и на языке Python. Эти функции часто используются в качестве аргумента *filter* функции `getmembers`.

stack `stack(context=1)`

Возвращает список кортежей, содержащих по шесть элементов каждый. Первый кортеж относится к вызывающей функции, второй — к функции, вызвавшей вызывающую функцию, и т.д. Элементами каждого кортежа являются: объект фрейма, имя файла, номер строки, имя функции, список строк исходного кода контекста, окружающего текущую строку, индекс текущей строки в пределах этого списка.

Интроспекция вызываемых объектов в версии v3

В версии v3 для интроспекции сигнатуры вызываемых объектов лучше всего не использовать устаревшие функции наподобие `inspect.getargspect(f)`, которые в будущем будут удалены. Используйте для этой цели функцию `inspect.signature(f)`, возвращающую экземпляр *s* класса `inspect.Signature`.

Атрибут *s*.parameters — это экземпляр OrderedDict, сопоставляющий имена параметров с соответствующими экземплярами inspect.Parameter. Вызов *s*.bind(**a*, ***k*) связывает все параметры с заданными позиционными и именованными аргументами, тогда как вызов *s*.bind_partial(**a*, ***k*) — только подмножество этих параметров: каждый из этих вызовов возвращает экземпляр *b* класса inspect.BoundArguments.

Для получения более подробной информации относительно интроспекции сигнатур вызываемых объектов посредством этих классов и их методов и ознакомления с соответствующими примерами обратитесь к документу **PEP 362** (https://www.python.org/dev/peps/pep-0362/).

Пример использования модуля inspect

Предположим, что где-то в вашей программе выполняется вызов x.f() и вы неожиданно получаете исключение AttributeError, информирующее о том, что объект *x* не имеет атрибута *f*. Это означает, что объект *x* оказался не тем, которого вы ожидали, и вы хотите получить больше информации относительно *x* в качестве предварительного шага на пути к выяснению того, почему *x* ведет себя таким образом и как это можно исправить. Замените приведенную выше инструкцию следующим кодом.

```
try: x.f()
except AttributeError:
    import sys, inspect
    print('x is type {}, ({!r})'.format(type(x), x),
                                    file=sys.stderr)
print("x's methods are:", file=sys.stderr, end='')
for n, v in inspect.getmembers(x, callable):
    print(n, file=sys.stderr, end='')
print(file=sys.stderr)
raise
```

В этом примере используется функция sys.stderr (см. описание в табл. 7.3), поскольку она отображает информацию, связанную с ошибкой, а не с результатами работы программы. Функция getmembers модуля inspect получает имена всех методов экземпляра *x* для их последующего отображения. Если вам придется часто применять диагностику такого рода, упакуйте ее в отдельную функцию.

```
import sys, inspect
def show_obj_methods(obj, name, show=sys.stderr.write):
    show('{} is type {}({!r})\n'.format(name,obj,type(obj)))
    show("{}'s methods are: ".format(name))
    for n, v in inspect.getmembers(obj, callable):
        show('{} '.format(n))
    show('\n')
```

Теперь приведенный выше пример принимает следующий вид.

```
try: x.f()
except AttributeError:
    show_obj_methods(x, 'x')
    raise
```

Для кода, который используется в диагностических и отладочных целях, хорошо продуманная программная структура и организация важны в той же мере, что и для кода, реализующего функциональность программы. Неплохой подход, который можно использовать при определении диагностических и отладочных функций, описан в разделе "Встроенная переменная __debug__" в главе 5.

Модуль traceback

Модуль traceback позволяет извлекать, форматировать и выводить трассировочную информацию, которой обычно сопровождаются неперехватываемые исключения. По умолчанию модуль traceback воспроизводит форматирование, применяемое Python. Однако модуль traceback обеспечивает возможность более детального управления форматом вывода трассировочной информации. Он предоставляет множество функций, предназначенных для этих целей, но в типичных случаях вы будете использовать только одну из них, описание которой приведено ниже.

print_exc print_exc(*limit*=None, *file*=sys.stderr)

Функцию print_exc можно вызывать из обработчика исключений или из вызываемой им (прямо или косвенно) функции. Функция print_exc записывает в файловый объект трассировочную информацию, которую Python выводит в стандартный поток stderr в случае неперехватываемых исключений. Целочисленный аргумент limit задает количество записей, выводимых функцией print_exc из объекта traceback с трассировочной информацией и соответствующих вложенным вызовам. Например, если вы хотите, чтобы обработчик исключений инициировал вывод диагностического сообщения, как если бы исключение распространялось, но предотвратил его дальнейшее распространение (чтобы ваша программа продолжила выполнение и не были вызваны никакие другие обработчики исключений), выполните в обработчике вызов traceback.print_exc().

Модуль pdb

Модуль pdb использует специальные функции интерпретатора Python для реализации простого интерактивного отладчика командной строки. С его помощью вы сможете задавать точки останова, исполнять инструкции исходного код в пошаговом режиме, исследовать кадры стека и т.п.

Чтобы выполнить код под управлением модуля pdb, импортируйте его, а затем вызовите функцию pdb.run, передав ей в качестве единственного аргумента строку кода, подлежащего выполнению. Чтобы использовать модуль pdb для поставарийной отладки (отладка кода, выполнение которого было только что прервано

распространением исключения в интерактивном интерпретаторе), вызовите функцию pdb.pm() без аргументов. Чтобы запустить модуль pdb непосредственно из кода приложения, импортируйте его, а затем вызовите функцию pdb.set_trace().

После запуска модуля pdb он прежде всего читает текстовые файлы с расширением *.pdbrc*, находящиеся в вашем домашнем каталоге и текущем каталоге. Эти файлы могут содержать любые команды pdb, но чаще всего вы помещаете в них алиасные имена (псевдонимы) команд, чтобы определить полезные синонимы и сокращенные названия других наиболее часто используемых команд.

Получив управление, модуль pdb выводит приглашение к вводу в виде строки '(Pdb) ', вслед за которым вы вводите команды pdb. Команда help (которую можно ввести в сокращенной форме h) позволяет вывести список доступных команд. Вызов команды help с аргументом (отделенным пробелом) позволяет получить сведения о любой конкретной команде. Для большинства команд допускается сокращение их названий до одной или двух начальных букв, но команды всегда должны вводиться с использованием нижнего регистра: модуль pdb, как и сам Python, различает регистр букв. Ввод пустой строки повторяет предыдущую команду. Наиболее часто используемые команды интерактивного отладчика pdb приведены в табл. 16.1.

Таблица 16.1. Команды отладчика pdb

Команда	Описание
!	! *statement* Выполняет инструкцию *statement* Python в текущем контексте отладки
alias, unalias	alias [*name* [*command*]] Команда alias без аргументов выводит текущий список определений псевдонимов. Команда alias *name* выводит текущее определение псевдонима *name*. Аргумент command в полной форме команды представляет любую команду pdb вместе с ее аргументами и может содержать символы %1, %2 и т.д. для ссылки на конкретные аргументы, которые передаются вновь определяемому псевдониму *name*, или символ %* для ссылки на все подобные аргументы. Команда unalias *name* удаляет псевдоним
args, a	args Выводит список всех аргументов, переданных функции, которая в данный момент подвергается отладке
break, b	break [*location* [, *condition*]] Команда break без аргументов выводит список всех установленных на данный момент точек останова и количество срабатываний каждой из них. Если задан аргумент *location*, команда break устанавливает точку останова в соответствующем месте кода. Значением *location* может быть номер строки или имя функции, которым может предшествовать имя файла: это позволяет задать точку останова в файле, не являющемся

Команда	Описание
	текущим, или в начале функции, для однозначного указания которой одного имени недостаточно (например, функция может существовать более чем в одном файле). Аргумент *condition*, если он задан, является выражением, которое будет вычисляться (в контексте отладки) каждый раз перед выполнением указанной строки кода или функции. Выполнение приостанавливается, только если это выражение возвращает истинное значение. При задании новой точки останова команда `break` возвращает номер точки останова, позволяющий впоследствии ссылаться на новую точку останова в любой другой команде `pdb`, связанной с точками останова
`clear`, `cl`	`clear [ breakpoint-numbers ]` Сбрасывает (удаляет) одну или несколько точек останова. Команда `clear` без аргументов удаляет все точки останова после получения подтверждения. Чтобы деактивизировать точку останова, не удаляя ее, следует выполнить команду `disable`, которая описана ниже
`condition`	`condition breakpoint-number [ expression ]` Команда `condition n expression` устанавливает или изменяет условие в точке останова *n*. Команда `condition n` без аргумента *expression* делает точку останова безусловной
`continue`, `c`, `cont`	`continue` Продолжает выполнение кода, пока не встретится следующая точка останова
`disable`	`disable [ breakpoint-numbers ]` Деактивизирует одну или несколько точек останова. Команда `disable` без аргументов деактивизирует все точки останова (после получения подтверждения). Отличается от команды `clear` тем, что отладчик запоминает точку останова, и впоследствии ее можно заново активизировать с помощью команды `enable`
`down`, `d`	`down` Перемещает текущий кадр стека на один уровень вниз в стеке трассировки (т.е. в направлении самого последнего вызова функции). Обычно текущая позиция располагается в самом низу стека (т.е. ей соответствует последняя вызванная функция, которая теперь подвергается отладке), поэтому команда `down` не может выполнить перемещение на нижний уровень. Однако она будет полезной, если перед этим вы выполнили команду `up`, которая перемещает текущую позицию вверх
`enable`	`enable [ breakpoint-numbers ]` Активизирует одну или несколько точек останова. Команда `enable` без аргументов активизирует все точки останова после получения подтверждения

Команда	Описание

ignore	ignore *breakpoint-number* [*count*]
	
	Деактивизирует указанную точку останова на *count* проходов (по умолчанию *count* равно 0). Срабатывание точки останова со значением счетчика пропусков *count* больше 0 лишь уменьшает значение *count* на единицу. При срабатывании точки останова, счетчик пропусков которой равен 0, выполнение приостанавливается и выводится интерактивное приглашение pdb. Пусть, например, модуль *fob.py* содержит следующий код:
	
	`def f():`
	`  for i in range(1000):`
	`  g(i)`
	`def g(i):`
	`  pass`
	
	Рассмотрим следующий интерактивный сеанс отладки с помощью pdb (в зависимости от используемой вами версии Python форматирование текста на вашем экране может несколько отличаться от приведенного ниже).
	
	`>>> import pdb`
	`>>> import fob`
	`>>> pdb.run('fob.f()')`
	`> <string>(1)?()`
	`(Pdb) break fob.g`
	`Breakpoint 1 at C:\mydir\fob.py:5`
	`(Pdb) ignore 1 500`
	`Will ignore next 500 crossings of breakpoint 1.`
	`(Pdb) continue>`
	`C:\mydir\fob.py(5)g()`
	`-> pass`
	`(Pdb) print(i)`
	`500`
	
	В соответствии с выведенным pdb сообщением команда ignore приказывает отладчику игнорировать следующие 500 попаданий в точку останова 1, которую мы установили на функции *fob.g* предыдущей инструкцией break. Поэтому, когда выполнение приостановится, функция *g* уже будет вызвана к этому моменту 500 раз, что подтверждает вывод значения ее аргумента *i*, которое действительно равно 500. Теперь значение счетчика пропусков точки останова 1 равно 0. Если мы выполним еще одну команду continue и используем команду print i, то отобразится значение i, равное 501. Иными словами, как только значение счетчика пропусков уменьшается до 0, выполнение приостанавливается всякий раз, когда встречается данная точка. Если мы хотим дополнительно пропустить некоторое количество попаданий, то должны предоставить pdb еще одну команду ignore, вновь устанавливающую для точки останова 1 значение счетчика, превышающее значение 0

Команда	Описание
list, l	`list [ first [ , last ] ]` Команда `list` без аргументов выводит `11` (одиннадцать) строк, центрированных на текущей, или следующие `11` строк, если предыдущая команда также была командой `list`. Необязательные аргументы команды `list` позволяют указать диапазон выводимых строк в пределах текущего файла. Команда `list` выводит не логические, а физические строки исходного кода, включая комментарии и пустые строки
next, n	`next` Выполняет текущую строку "без захода" в любую функцию, вызываемую из текущей строки. Однако при попадании в точку останова, установленную в функциях, прямо или косвенно вызываемых из текущей строки, выполнение приостанавливается
print, p	`p expression` или `print(expression)` Вычисляет выражение в текущем контексте и отображает результат
quit, q	`quit` Немедленно прекращает выполнение как отладчика `pdb`, так и программы, которая подвергается отладке
return, r	`return` Выполняет оставшуюся часть текущей функции, приостанавливая выполнение лишь в точках останова, если таковые имеются.
step, s	`step` Выполняет текущую строку "с заходом" в любую из вызываемых в ней функций
tbreak	`tbreak [ location [ ,condition ] ]` Аналогична команде `break`, но устанавливаемая точка останова является временной (т.е. `pdb` автоматически удаляет ее после первого срабатывания)
up, u	`up` Перемещает текущий кадр стека на один уровень вверх в стеке трассировки (т.е. в направлении от последней вызванной функции к вызвавшей ее функции)
where, w	`where` Отображает трассировку стека и указывает текущий кадр (т.е. тот, в контексте которого команда `!` выполняет инструкции, команда `args` отображает аргументы, команда `print` вычисляет выражение и т.п.)

Модуль warnings

Предупреждения — это сообщения об ошибках или аномалиях, не являющихся достаточно серьезными для того, чтобы прервать поток управления программы (как это произошло бы в случае возбуждения исключения). Модуль `warnings` предоставляет возможность детально контролировать, какие предупреждения должны выводиться и какова должна быть реакция на них. Вы можете организовать вывод предупреждений при выполнении определенных условий, вызвав функцию `warn` из модуля `warnings`. Другие функции, входящие в этот модуль, позволяют управлять форматированием предупреждений, определять, куда они должны направляться, подавлять предупреждения при выполнении определенных условий или преобразовывать в исключения.

Классы

Модуль `warnings` представляет предупреждения, используя не собственные, а встроенные классы исключений. Класс `Warning` наследует класс `Exception` и его базовый класс для всех предупреждений. Вы можете определить собственные классы предупреждений, но они должны быть подклассами класса `Warning`, наследующимися от него как непосредственно, так и через другие существующие классы, включая нижеперечисленные.

`DeprecationWarning`
: Представляет предупреждения об использовании нерекомендуемых возможностей, оставленных лишь с целью обеспечения обратной совместимости.

`RuntimeWarning`
: Представляет предупреждения об использовании возможностей, семантика которых содержит риски возникновения проблем во время выполнения.

`SyntaxWarning`
: Представляет предупреждения об использовании нерекомендуемого синтаксиса.

`UserWarning`
: Представляет другие, определяемые пользователем предупреждения, не охватываемые вышеприведенными случаями.

Объекты

Python не предоставляет никаких конкретных объектов предупреждений. Предупреждение состоит из следующих компонентов: *`message`* (строка), *`category`* (подкласс `Warning`) и двух информационных элементов, идентифицирующих местоположение, в котором было возбуждено предупреждение: *`module`* (имя модуля, возбудившего предупреждение) и *`lineno`* (номер строки исходного кода, в которой было

возбуждено предупреждение). С концептуальной точки зрения все эти компоненты можно считать атрибутами объекта предупреждения *w*: далее для ясности мы используем нотацию атрибутов, однако фактически никакого объекта *w* не существует.

Фильтры

В каждый момент времени модуль `warnings` поддерживает список активных фильтров предупреждений. Когда модуль `warnings` импортируется в первый раз во время выполнения, он проверяет значение атрибута `sys.warnoptions`, чтобы определить начальный набор фильтров. Можно настроить фильтры предупреждений, используя параметр `-W` при запуске Python, чтобы изменить значение атрибута `sys.warnoptions` для данного сеанса. Не полагайтесь на начальный набор фильтров, хранимый в атрибуте `sys.warnoptions`, потому что это особенность реализации фреймворка предупреждений, которая в последующих версиях Python может быть изменена.

Каждый раз, когда возбуждается предупреждение *w*, модуль `warnings` сверяет его с каждым из фильтров, пока не обнаружит тот, который соответствует *w*. Поведение *w* определяется первым совпавшим фильтром. Каждый фильтр представляет собой кортеж из пяти элементов. Первым элементом этого кортежа является *action* — строка, определяющая, что именно должно происходить при возбуждении данного предупреждения. Остальные четыре элемента — *message*, *category*, *module* и *lineno* — определяют критерии соответствия предупреждения *w* данному фильтру, причем для совпадения с фильтром требуется выполнение всех четырех условий. Описание этих элементов приведено ниже (с использованием нотации атрибутов для указания концептуальных атрибутов предупреждения *w*).

`message`
: Строка шаблона регулярного выражения. Условие совпадения — `re.match(`*message*`, w.message, re.I)` (регистр букв игнорируется).

`category`
: `Warning` или подкласс `Warning`. Условие совпадения — `issubclass(w.category, `*category*`)`.

`module`
: Строка шаблона регулярного выражения. Условие совпадения — `re.match(`*module*`, w.module)` (регистр букв учитывается).

`lineno`
: Целое число. Условие совпадения — *lineno* `in (0, w.lineno)`: т.е. либо *lineno* имеет значение `0`, и тогда значение `w.lineno` несущественно, либо значения `w.lineno` и *lineno* должны совпадать.

Когда соответствие установлено, первое поле фильтра, `action`, определяет дальнейшее поведение.

`'always'`
: Строка `w.message` выводится всегда, независимо от того, возбуждалось ли ранее предупреждение `w` или не возбуждалось.

`'default'`
: Строка `w.message` выводится, только если данное предупреждение было возбуждено впервые в данном местоположении (определяемом конкретной парой `w.module, w.location`).

`'error'`
: Возбуждается исключение `w.category(w.message)`.

`'ignore'`
: Предупреждение `w` игнорируется.

`'module'`
: Строка `w.message` выводится, только если предупреждение `w` было возбуждено впервые в модуле `w.module`.

`'once'`
: Строка `w.message` выводится, только если предупреждение `w` появляется впервые, независимо от местоположения.

`__warningsgregistry__`

Когда какой-либо модуль генерирует предупреждение, модуль `warnings` добавляет в глобальные переменные этого модуля словарь `__warningsgregistry__`, если до этого он не существовал. Каждый ключ в этом словаре является парой (*message*, *category*) или кортежем из трех элементов (*message*, *category*, *lineno*). Соответствующее значение равно `True`, если последующие появления этого предупреждения должны подавляться. Таким образом, вы, например, можете сбросить состояние подавления всех предупреждений, исходящих из модуля `m`, выполнив вызов `m.__warningsregistry__.clear()`: если вы это сделаете, то все сообщения будут вновь выводиться (однократно), даже если ранее они активизировали фильтр с атрибутом *action*, установленным в значение `'module'`.

Функции

Модуль `warnings` предоставляет следующие функции.

`filterwarnings` `filterwarnings(`*action*`, `*message*`='.*', `*category*`=Warning, `*module*`='.*', `*lineno*`=0, `*append*`=False)`

Добавляет фильтр в список активных фильтров. Если *append* равно `True`, функция `filterwarnings` добавляет фильтр после всех существующих фильтров (т.е. присоединяет фильтр к списку существующих фильтров). В противном случае она вставляет фильтр перед любым другим существующим фильтром. Все компоненты, кроме *action*, имеют значение по умолчанию, означающее "соответствовать

всему". Как уже отмечалось, *message* и *module* — это строки шаблонов регулярных выражений, *category* — некоторый подкласс Warning, *lineno* — целое число, *action* — строка, которая определяет, что должно происходить в случае, если предупреждение соответствует данному фильтру.

formatwarning formatwarning(*message*, *category*, *filename*, *lineno*)

Возвращает строку, которая представляет заданное предупреждение с использованием стандартного форматирования.

resetwarnings resetwarnings()

Удаляет фильтры из списка фильтров. Функция resetwarnings также отбрасывает любые фильтры, первоначально добавленные с помощью параметра командной строки -W.

showwarning showwarning(*message*, *category*, *filename*, *lineno*, *file*=sys.stderr)

Выводит заданное предупреждение в заданный объект файла. Фильтры, поле action которых задает вывод предупреждений, вызывают функцию showwarning, позволяя аргументу *file* принимать заданное по умолчанию значение sys.stderr. Чтобы изменить действия, выполняемые в том случае, если фильтр выводит предупреждения, напишите собственную функцию с такой же сигнатурой и свяжите ее с переменной warnings.showwarning, тем самым переопределяя реализацию, заданную по умолчанию.

warn warn(*message*, *category*=UserWarning, *stacklevel*=1)

Генерирует предупреждение, чтобы фильтры проверили его и, возможно, вывели. Местоположением такого возбуждения является текущая функция (которая вызвала функцию warn), если аргумент *stacklevel* равен *1*, или функция, вызвавшая текущую функцию, если аргумент *stacklevel* равен 2. Таким образом, передача 2 в качестве значения аргумента *stacklevel* позволяет писать функции, которые генерируют предупреждения от имени вызывающего их кода, как это продемонстрировано в следующем примере:

```
def toUnicode(bytestr):
    try:
        return bytestr.decode()
    except UnicodeError:
        warnings.warn(
            'Invalid characters in {}'.
            format(bytestr), stacklevel=2)
        return bytestr.decode(errors='ignore')
```

Благодаря параметру *stacklevel=2* создается видимость того, что источником предупреждения является код, вызывающий функцию toUnicode, а не сама эта функция. Это играет очень важную роль, если значением поля *action* фильтра, соответствующего данному предупреждению, является *default* или *module*, поскольку в этом случае предупреждение будет выводиться только в том случае, если оно впервые исходит из данного местоположения или модуля.

Оптимизация

"Сначала добейтесь, чтобы код работал. Затем добейтесь, чтобы код работал правильно. После этого добейтесь, чтобы код работал быстро". Приведенное высказывание, которое часто повторяют в различных вариациях, известно как "золотое правило программирования". Насколько нам удалось выяснить, оно принадлежит Кенту Беку, который приписывает авторство своему отцу. Данный принцип часто цитируют, но редко соблюдают. Его эквивалентной отрицательной формой, несколько преувеличенной для усиления, является высказывание, процитированное Дональдом Кнутом (который приписывает авторство Чарльзу Хоару): "Преждевременная оптимизация — корень всех бед в программировании".

Оптимизация преждевременна, если ваш код еще не работоспособен или вы не можете точно сказать, что именно будет делать ваш код (поскольку в этом случае вы не можете быть уверены в том, что он работает). Прежде всего заставьте код работать: убедитесь в том, что он корректно справляется с теми задачами, для выполнения которых предназначен.

Оптимизация также преждевременна, если ваш код работает, но вас не удовлетворяют его общая архитектура и дизайн. Сначала устраните структурные дефекты и только после этого беритесь за оптимизацию: сначала добейтесь, чтобы код работал, а затем добейтесь, чтобы он работал правильно. Эти шаги не должны делаться по вашему усмотрению; работающий код с хорошо продуманной архитектурой — это требование *всегда* является обязательным.

В отличие от этого вам не всегда придется добиваться более быстрой работы кода. Эталонные тесты (бенчмаркинг) могут продемонстрировать, что производительность вашего кода вполне приемлема уже после выполнения первых двух шагов. Если производительность недостаточна, то профилирование нередко демонстрирует, что все проблемы производительности обусловлены небольшой частью кода, доля которой может составлять от 10 до 20% и выполнением которой программа может быть занята от 80 до 90% всего времени. Подобные критические по отношению к производительности участки кода известны как *узкие места* или *горячие точки* в программе. Бессмысленно затрачивать усилия на оптимизацию больших порций кода, ответственных, скажем, всего за 10% времени выполнения вашей программы. Даже если вы добьетесь того, что эта часть программы будет работать в 10 раз быстрее (редкое достижение), то общее время выполнения вашей программы уменьшится всего лишь на 9% — ускорение, которое не заметит ни один пользователь. Если оптимизация все-таки необходима, сосредоточьте усилия на коде, который заслуживает этого, — на узких местах. Вы можете оптимизировать узкие места, оставив весь код написанным исключительно на языке Python и тем самым не создавая препятствий для его будущего портирования в другие реализации Python. В некоторых случаях вы можете прибегнуть к тому, чтобы переписать код некоторых узких мест в виде расширений Python (о чем пойдет речь в главе 24), что может обеспечить

еще более высокую производительность (возможно, с потерей некоторых возможностей портирования программы в будущем).

Разработка достаточно быстрых приложений Python

Начинайте с проектирования, написания кода и тестирования своего приложения на языке Python, используя доступные модули расширения, если они экономят вашу работу. Это займет намного меньше времени, чем в случае использования классического компилируемого языка программирования. Затем выполните оценочное тестирование приложения, чтобы выяснить, работает ли результирующий код достаточно быстро. Нередко так оно и есть, и тогда вы можете считать, что справились со своей задачей — примите поздравления! Отправляйте программу заказчику!

Поскольку значительная часть самого Python (равно как и многие из модулей его стандартной библиотеки и модули расширений) написана с использованием высокооптимизированного языка C, может оказаться так, что ваше приложение уже работает быстрее, чем работал бы типичный C-код. Но если приложение работает слишком медленно, то прежде всего вы должны пересмотреть свои алгоритмы и структуры данных. Убедитесь в отсутствии узких мест в архитектуре приложения, сетевом трафике, доступе к базам данных и взаимодействии с операционной системой. В большинстве случаев именно эти факторы, а не используемый язык программирования, являются наиболее вероятной причиной медленной работы приложения. Нередко быстродействие приложения можно значительно повысить, отрегулировав некоторые аспекты его крупномасштабной архитектуры, и Python — отличная среда для подобных экспериментов.

Если программа по-прежнему работает слишком медленно, выполните ее профилирование, чтобы выяснить, на что именно больше всего тратится время. В приложениях часто удается обнаружить узкие в вычислительном отношении места: небольшие области исходного кода — обычно не превышающие примерно 20% от полного его объема — оказываются ответственными за потребление 80% и более всего времени выполнения программы. Оптимизируйте узкие места, применяя различные виды оптимизации, которым посвящена оставшаяся часть главы.

Если после применения обычных методов оптимизации на уровне Python в коде все еще остаются некоторые узкие в вычислительном отношении места, перепишите их в виде модулей расширения Python, о которых пойдет речь в главе 24. В конечном счете ваше приложение будет выполняться примерно с той же скоростью, как если бы оно было написано на языке C, C++, Fortran, или даже быстрее, если крупномасштабные эксперименты позволят вам выбрать лучшую архитектуру. Общая продуктивность описанного процесса программирования не намного меньше той, которой вы достигли бы, программируя весь код на языке Python. Внесение в будущем изменений и сопровождение кода не вызовет затруднений, поскольку для выражения общей структуры программы вы используете Python, тогда как более низкоуровневые и более сложные для сопровождения программного продукта языки используются лишь в специфических узких местах кода.

Следуя этому процессу при создании приложений в выбранной вами области, вы накопите библиотеку повторно используемых модулей расширения Python. Поэтому ваша продуктивность в разработке других быстро выполняющихся приложений на языке Python в данной области будет все время возрастать.

Даже если в силу каких-либо внешних обстоятельств вы будете вынуждены полностью переписать код всего приложения на более низкоуровневом языке программирования, вам все равно имеет смысл начинать с написания кода на языке Python. Быстрое прототипирование издавна считается наиболее эффективным способом разработки архитектуры программного обеспечения. Работающий прототип позволяет убедиться в правильности идентификации проблем и действенности намеченных путей их разрешения. Кроме того, прототипирование предоставляет возможность экспериментировать с крупномасштабной архитектурой, что может привести к существенному улучшению производительности приложения. Начав с прототипа, написанного на языке Python, вы сможете обеспечить его постепенную миграцию на другие языки программирования посредством модулей расширения, если в этом возникнет необходимость. На каждом этапе функциональность и тестируемость приложения будут полностью сохранены. Тем самым будут устранены риски нанесения ущерба архитектурной целостности приложения на стадии написания кода. Результирующий программный продукт будет отличаться большими надежностью и быстродействием, чем если бы изначально он был написан на более низкоуровневом языке, а ваша продуктивность, если и не будет такой же, как при создании приложений исключительно на языке Python, все равно будет более высокой, чем при использовании более низкоуровневого языка для написания всего кода.

Бенчмаркинг

Эталонное тестирование, или *бенчмаркинг*, аналогично системному тестированию: оба они во многом напоминают выполнение программы в производственных условиях. В обоих случаях в вашем распоряжении должно быть хотя бы некоторое работающее подмножество запланированной функциональности программы, и вы должны использовать заранее известные воспроизводимые входные данные. Для бенчмаркинга фиксация и сравнение выходных результатов с эталонными не требуется: поскольку вы добиваетесь, чтобы программа работала, причем работала правильно, до того, как будете добиваться повышения ее быстродействия, к моменту выполнения нагрузочного тестирования вы уже уверены в корректности ее функционирования. Вам необходим лишь набор данных, достаточно представительный для типичных системных операций, в идеальном случае — таких, выполнение которых создает наибольшую нагрузку для программы. Если ваша программа выполняет операции нескольких видов, проследите за тем, чтобы некоторые нагрузочные тесты выполнялись для каждой разновидности операций.

Вероятно, той точности измерения истекшего времени, которую обеспечивают ваши наручные часы, будет вполне достаточно для оценочного тестирования

большинства программ. Другое дело — программы с жесткими ограничениями в отношении реального времени выполнения, но их потребности очень отличаются от потребностей обычных программ во многих отношениях. За исключением программ, к которым предъявляются весьма специфические требования, различия в производительности в пределах от 5 до 10% не имеют значения для программ широкого применения.

В случае бенчмаркинга "игрушечных" программ или фрагментов кода с целью подбора наиболее эффективных алгоритмов или структур данных вам нужна более высокая точность: для решения задач подобного рода вам вполне подойдет модуль `timeit` стандартной библиотеки Python (описанный в разделе "Модуль `timeit`"). Оценочное тестирование, обсуждаемое в этом разделе, имеет другой характер: это приближение к выполнению программы в реальных условиях, единственной целью которого является выяснение того, достаточно ли приемлема производительность программы при выполнении задач каждого вида, прежде чем приступать к профилированию программы и предпринимать другие меры для ее оптимизации. Для такого "системного" бенчмаркинга ситуация, которая приближается к обычным рабочим условиям программы, является наилучшей, тогда как высокая точность хронометрирования вовсе не важна.

Крупномасштабная оптимизация

На производительность программы наибольшее влияние оказывают ее крупномасштабные аспекты: выбранные вами общая архитектура, алгоритмы и структуры данных.

Проблемы производительности, на которые вы чаще всего должны обращать внимание, связаны с традиционной для компьютерной науки оценкой временной сложности алгоритмов с использованием *О-нотации* или *нотации "О большое"*. Говоря неформальным языком, если `N` — размер входа (входных данных) алгоритма, то О-нотация выражает временную сложность (трудоемкость) алгоритма для больших `N` как величину, пропорциональную некоторой функции от `N`. (Для более строгого обсуждения следовало бы использовать *Θ-нотацию*, но на практике программисты предпочитают использовать О-нотацию, вероятно по той причине, что писать букву `O` немного легче, чем греческую букву "тета"!)

Алгоритм с порядком сложности `O(1)` (другое название — *алгоритм постоянного времени*) — это алгоритм, время выполнения которого не растет с ростом `N`. Алгоритм с порядком сложности `O(N)` (другое название — *алгоритм линейного времени*) — это алгоритм, для которого в случае достаточно больших `N` обработка вдвое большего объема данных требует в два раза больше времени, обработка втрое большего объема данных — в три раза больше времени и т.д., т.е. время обработки растет пропорционально `N`. Алгоритм с порядком сложности $O(N^2)$ (другое название — *алгоритм квадратичного времени*) — это алгоритм, для которого в случае достаточно больших `N` обработка вдвое большего объема данных требует в четыре раза больше

времени, обработка втрое большего объема данных — в девять раз больше времени и т.д., т.е. время выполнения растет пропорционально квадрату N. Идентичные понятия и нотация используются для описания потребления программой памяти ("пространства"), а не времени.

Более подробную информацию относительно нотации "О большое", алгоритмов и характеристик их сложности можно найти в любой хорошей книге, посвященной алгоритмам и структурам данных; мы рекомендуем обратиться к книге *Python Algorithms: Mastering Basic Algorithms in the Python Language* (Magnus Lie Hetland).

Чтобы понять, какое практическое значение для вашей программы имеют соображения, основанные на использовании нотации "О большое", рассмотрим два способа получения всех элементов из входного итерируемого объекта и аккумулирования их в виде списка, в котором элементы располагаются в обратном порядке.

```
def slow(it):
    result = []
    for item in it: result.insert(0, item)
    return result

def fast(it):
    result = []
    for item in it: result.append(item)
    result.reverse()
    return result
```

Мы могли бы выразить каждую из этих функций в более компактном виде, но их представление в терминах простых операций делает оценку ключевых различий между ними более наглядной. Функция `slow` создает результирующий список, вставляя каждый входной элемент перед всеми ранее полученными элементами. Функция `fast` присоединяет каждый входной элемент после всех ранее полученных элементов списка, а затем обращает порядок следования элементов в результирующем списке. Полагаясь на интуицию, можно было бы подумать, что завершающая операция обращения списка представляет дополнительную работу, и поэтому функция `slow` должна работать быстрее, чем функция `fast`. В действительности все происходит иначе.

Каждый вызов метода `result.append` требует примерно одинакового времени, независимо от того, сколько элементов уже находится в списке `result`, поскольку (почти) всегда существует свободное место для помещения очередного элемента в конец списка (строго говоря, операция `append` характеризуется *амортизированной сложностью O(1)*, однако амортизированные оценки сложности алгоритмов в данной книге не рассматриваются). Цикл `for` в функции `fast` выполняется `N` раз для получения `N` элементов. Поскольку время выполнения каждой итерации цикла постоянно, общее время выполнения цикла характеризуется показателем `O(N)`. Вызов метода `result.reverse` также выполняется за время `O(N)`, поскольку время его выполнения прямо пропорционально общему количеству элементов. Таким образом, общее время выполнения функции `fast` составляет `O(N)`. (Если вам непонятно, почему сумма

двух величин, каждая из которых характеризуется показателем O(N), также имеет сложность O(N), вспомните о том, что сумма двух линейных функций N также является линейной функцией N, а показатель O(N) имеет тот же смысл, означая, что потребляемое время является линейной функцией N.)

С другой стороны, при каждом вызове метода result.insert создается пространство в слоте 0 для вставки нового элемента, что требует перемещения всех уже имеющихся в списке элементов на один слот вперед. Необходимое для этого время пропорционально количеству имеющихся элементов. Поэтому общее время, которое необходимо для получения N элементов, пропорционально сумме 1+2+3+...N-1, для которой порядок времени выполнения равен $O(N^2)$. Поэтому общее время выполнения функции slow составляет $O(N^2)$.

Если вы не ограничиваете размер N входа строго каким-либо небольшим значением, то почти всегда решение $O(N^2)$ стоит заменить решением O(N). Если N может расти без каких-либо строгих ограничений, то решение $O(N^2)$ будет работать катастрофически медленнее по сравнению с решением O(N) для больших значений N, независимо от того, какими могут быть значения констант пропорциональности в каждом конкретном случае (а также от того, что говорят вам данные профилирования). Если только в вашей программе не присутствуют другие "бутылочные горлышки" с показателем времени выполнения $O(N^2)$ или худшим, которые вы не в состоянии устранить, то та часть программы, которая характеризуется показателем $O(N^2)$, превращается в "бутылочное горлышко", оказывающее доминирующее влияние на время выполнения программы при больших значениях N. В ваших интересах постоянно отслеживать возможные проблемы, связанные с "О большим": по сравнению с ними все остальные проблемы производительности, как правило, почти несущественны.

Кстати, функцию fast можно заставить работать еще быстрее, применив в ней идиому в духе Python. Для этого достаточно заменить первые две строки кода единственной инструкцией:

```
result = list(it)
```

Это изменение не повлияет на характер "О большого" функции fast (после внесения изменения она по-прежнему будет оставаться функцией с показателем сложности O(N)), но оно увеличит быстродействие кода на большой постоянный множитель.

Простое лучше, чем сложное, и обычно работает быстрее!

Чаще всего в Python самый простой, прозрачный и прямой способ решения задачи одновременно является и самым быстрым.

Выбор алгоритмов с хорошим показателем "О большое" в Python — в общих чертах задача того же плана, что и в любом другом языке программирования. Вам лишь нужно получить с помощью "О большого" несколько подсказок относительно оценки производительности элементарных строительных кирпичиков Python, которые мы предоставляем вам в последующих разделах.

Операции над списками

В Python списки внутренне реализованы в виде *векторов* (также называемых *динамическими массивами*), а не в виде связанных списков. Этот выбор реализации определяет почти все характеристики быстродействия списков Python в терминах "О большого".

Объединение двух списков `L1` и `L2` с длинами `N1` и `N2` в одну цепочку (т.е. операция `L1+L2`) является операцией с порядком сложности `O(N1+N2)`. Умножение списка `L` длиной `N` на целое число `M` (т.е. операция `L*M`) является операцией с порядком сложности `O(N*M)`. Обращение к любому элементу списка или его повторное связывание является операцией с порядком сложности `O(1)`. Определение длины списка с помощью функции `len()` также является операцией категории `O(1)`. Обращение к любому срезу длины `M` имеет порядок сложности `O(M)`. Повторное связывание среза длиной `M` со срезом такой же длины также является операцией категории `O(M)`. Повторное связывание среза длиной `M1` со срезом другой длины `M2` характеризуется порядком сложности `O(M1+M2+N1)`, где `N1` — количество элементов после среза в целевом списке (иными словами, подобные повторные связывания, изменяющие длину срезов, сопровождаются относительно небольшими издержками, если они выполняются в конце списка, и большими, если они выполняются в начале или вблизи середины длинного списка). Если вам необходимо выполнять операции типа FIFO (first-in, first-out — первым пришел, первым ушел), то, пожалуй, список не является наиболее быстрой структурой для этих целей: попробуйте вместо него использовать тип `collections.deque`, рассмотренный в разделе "Класс `deque`" главы 7.

Большинство методов списков, описанных в табл. 3.3, эквивалентны повторному связыванию срезов и обладают эквивалентной производительностью, выражаемой показателем "О большое". Методы `count`, `index`, `remove` и `reverse`, а также оператор `in` характеризуются порядком сложности `O(N)`. Метод `sort` обычно имеет порядок сложности `O(N*log(N))`, но высоко оптимизирован до уровня `O(N)` в некоторых важных специальных случаях, например когда список уже отсортирован или отсортирован в обратном порядке за исключением нескольких элементов. Вызов `range(a,b,c)` в версии v2 имеет порядок сложности `O((b-a)/c)`. Вызовы `xrange(a,b,c)` в версии v2 и `range` в версии v3 имеют порядок сложности `O(1)`, но выполнение цикла по всем элементам результата имеет порядок сложности `O((b-a)/c)`.

Операции над строками

Большинство методов, выполняемых для строк длиной `N` (байтовых или Unicode), имеют порядок сложности `O(N)`. Порядок сложности метода `len(`*`astring`*`)` равен `O(1)`. Самым быстрым способом получения копии строки с транслитерацией и/или удалением указанных символов является строковый метод `translate`. Единственное наиболее важное в практическом отношении соображение относительно "О большого" применительно к строкам обсуждается в разделе "Сборка строк из отдельных элементов".

Операции над словарями

В Python словари внутренне реализованы с использованием хеш-таблиц. Этот выбор реализации определяет все характеристики быстродействия словарей Python в терминах "О большого".

Обращения к элементам словаря, их повторное связывание, добавление или удаление, равно как и методы `has_key`, `get`, `setdefault` и `popitem`, а также оператор `in` имеют порядок сложности `O(1)`. Вызов *d1*`.update(`*d2*`)` имеет порядок сложности `O(len(`*d2*`))`, вызов `len(`*adict*`)` — порядок сложности `O(1)`. Методы `keys`, `items` и `values` имеют порядок сложности `O(N)` в версии v2. Те же методы в версии `v3` и методы `iterkeys`, `iteritems` и `itervalues` в версии v2 имеют порядок сложности `O(1)`, но выполнение цикла по всем элементам итераторов, возвращаемых этими методами, имеет порядок сложности `O(N)`, а непосредственное выполнение цикла по словарю имеет тот же порядок сложности, выражаемый показателем "О большое", что и метод `iterkeys` в версии v2.

Никогда не используйте инструкцию `if x in d.keys():`

Никогда не выполняйте проверку `if x in d.keys():`! Эта операция имеет порядок сложности `O(N)`, тогда как эквивалентная проверка `if x in d:` — порядок сложности `O(1)` (инструкция `if d.has_key(x):` также имеет порядок сложности `O(1)`, но работает медленнее, чем `if x in d:`, не имеет никаких преимуществ, компенсирующих этот недостаток, и признана устаревшей, поэтому ее также не следует использовать).

Если ключами в словаре являются экземпляры классов, которые определяют метод `__hash__` и методы проверки на равенство, то, разумеется, эти методы оказывают влияние на скорость выполнения операций над множествами. Представленные в этом разделе указания относительно скорости выполнения справедливы, если хеширование и проверка на равенство являются операциями с порядком сложности `O(1)`.

Операции над множествами

В Python множества, как и словари, реализованы с использованием хеш-таблиц. Все характеристики быстродействия множеств Python в терминах "О большого" совпадают с аналогичными характеристиками словарей.

Добавление или удаление элемента множества имеет порядок сложности `O(1)`, равно как и оператор `in`. Вызов `len(aset)` имеет порядок сложности `O(1)`. Выполнение цикла по множеству имеет порядок сложности `O(N)`. Если элементами множества являются экземпляры классов, которые определяют метод `__hash__` и методы проверки на равенство, то, разумеется, эти методы оказывают влияние на скорость выполнения операций над множествами. Представленные в данном разделе указания относительно скорости выполнения справедливы, если хеширование и проверка на равенство являются операциями с порядком сложности `O(1)`.

Резюме оценок "О большое" для операций над встроенными типами Python

Пусть `L` — любой список, `T` — любая строка (простая/байтовая или Unicode), `D` — любой словарь, `S` — любое множество, содержащие в качестве элементов числа (выбраны исключительно с той целью, чтобы гарантировать для операций хеширования и сравнения порядок сложности `O(1)`), а *x* — любое число (из тех же соображений). Тогда операции над встроенными типами характеризуются порядками сложности, приведенными ниже.

`O(1)`

`len(`*L*`)`, `len(`*T*`)`, `len(`*D*`)`, `len(`*S*`)`, `L[i]`, `T[i]`, `D[i]`, `del D[i]`, `if` *x* `in D`, `if` *x* `in S`, `S.add(`*x*`)`, `S.remove(`*x*`)`, а также операции присоединения/удаления, выполняемые на правом конце *L*.

`O(N)`

Циклы по *L*, *T*, *D*, *S*, общие операции присоединения/удаления в списке *L* (за исключением правого конца), все методы *T*, `if` *x* `in` *L*, `if` *x* `in` *T*, большинство методов *L*, все операции поверхностного (мелкого) копирования.

`O(N log N)`

L`.sort` в большинстве случаев (но `O(N)`, если *L* уже почти отсортирован или отсортирован в обратном порядке).

Профилирование

В большинстве программ имеются так называемые "горячие точки" (жарг. *хот-споты*, от англ. *hot spots*) — сравнительно небольшие участки кода, на выполнение которых процессор затрачивает наибольшую часть времени. Не пытайтесь угадать, где именно в программе они могут располагаться: интуиция программиста в этой области — ненадежный подсказчик. Вместо этого используйте модуль `profile` для сбора данных профилирования на протяжении одного или нескольких запусков программы с известными входными данными. После этого используйте модуль `pstats` для сравнения, интерпретации и отображения профильных данных, которые получили.

Чтобы повысить точность измерений, можете откалибровать профилировщик Python для своего компьютера (т.е. определить, какие дополнительные накладные расходы вносит сам процесс профилирования). Впоследствии профилировщик сможет вычесть эти дополнительные затраты времени из измеренных временны*х* показателей, что позволит получить более реалистичную профилирующую информацию. Модуль `cProfile` стандартной библиотеки Python предлагает ту же функциональность, что и модуль `profile`, но является более предпочтительным, поскольку работает быстрее, а значит, с ним связаны меньшие накладные расходы. Стандартная библиотека Python (только в версии v2) содержит еще один модуль, предназначенный для профилирования кода: `hotshot` (`https://docs.python.org/2/library/hotshot.html`). К сожалению, он не поддерживает потоки, и его нельзя использовать в версии v3.

Модуль profile

Модуль profile предоставляет одну часто используемую функцию, которая описана ниже.

run run(*code*, *filename*=None)

Аргумент *code* — это строка, выполняемая профилировщиком с помощью инструкции exec и обычно представляющая собой вызов основной функции профилируемой программы. Аргумент *filename* — это путь к файлу, который создается или обновляется для записи данных профилирования. Обычно вы будете запускать функцию run несколько раз, указывая разные имена файлов и разные значения аргументов основной функции своей программы, чтобы задействовать различные части программы в той пропорции, в какой в соответствии с вашими ожиданиями они будут использоваться в ходе реальной работы. Затем вы используете модуль pstats для анализа и отображения данных, собранных при различных запусках программы.

Вызвав функцию run без указания имени файла, можно получить и направить в стандартный канал вывода сводный отчет, аналогичный тому, который можно было бы получить с помощью модуля pstats. Однако при таком подходе вы не сможете ни управлять форматом вывода, ни консолидировать данные нескольких запусков в один отчет. На практике вы будете редко использовать эту возможность: данные профилирования лучше сохранять в файлах.

Кроме того, модуль profile предоставляет класс Profile (о котором пойдет речь в следующем разделе). Непосредственная инстанциализация класса Profile позволяет получить доступ к расширенной функциональности, такой как возможность выполнения функции run в указанном локально и глобальном окружении. В данной книге подобная расширенная функциональность класса profile.Profile не обсуждается.

Калибровка

Чтобы откалибровать модуль profile для своего компьютера, нужно использовать класс Profile, который предоставляется и используется модулем profile в функции run. Экземпляр *p* класса Profile содержит один метод, используемый в целях калибровки.

calibrate p.calibrate(*N*)

Выполняется в цикле *N*, затем возвращает число, представляющее скрытые накладные расходы профилирования из расчета на один вызов на вашем компьютере. *N* должно быть достаточно большим в случае быстрой машины. Выполните вызов *p*.calibrate(10000) несколько раз, убедитесь в том, что возвращаемые числа близки между собой, и выберите наименьшее из них. Если эти числа значительно различаются между собой, проделайте то же самое с большими значениями *N*.

Процедура калибровки может занять какое-то время. Однако вам нужно выполнить ее всего лишь один раз, повторив только в том случае, если характеристики компьютера изменятся, например вследствие обновления операционной системы, установки дополнительной памяти или изменения версии Python. Определив накладные расходы профилирования для своей

системы, вы сможете передавать эту информацию модулю `profile` всякий раз, когда импортируете его, непосредственно перед использованием функции `profile.run`. Проще всего это можно сделать следующим образом:

```
import profile
profile.Profile.bias = ...the overhead you measured...
profile.run('main()', 'somefile')
```

Модуль `pstats`

Модуль `pstats` содержит единственный класс — `Stats`. Он предназначен для анализа и консолидации данных профилирования, которые содержатся в одном или нескольких файлах, записанных функцией `profile.run`.

`Stats` `class Stats(`*`filename`*`, *`*`filenames`*`)`

Создает экземпляр класса `Stats` с одним или несколькими именами файлов профильных данных, записанных функцией `profile.run`.

Экземпляр *s* класса `Stats` предоставляет методы для добавления данных профилирования, а также для сортировки и вывода результатов. Каждый из этих методов возвращает экземпляр *s*, что позволяет использовать выражения, содержащие цепочки вызовов. Основные методы экземпляра *s* описаны в табл. 16.2.

Таблица 16.2. Методы класса `Stats`

Метод	Описание
`add`	*s*`.add(`*filename*`)` Добавляет файл с профильными данными в набор, сохраняемый в объекте *s* для последующего анализа
`print_callees`, **`print_callers`**	*s*`.print_callees(*`*restrictions*`)` Выводит список функций из данных профилирования, которые хранятся в объекте *s*, отсортированный в соответствии с последним вызовом метода *s*`.sort_stats` и подчиняющийся заданным ограничениям `restrictions`. Вы можете вызвать каждый из методов `print` без передачи ему ограничений или с передачей нескольких ограничений, применяемых последовательно в указанном порядке для уменьшения количества строк вывода. Отдельное ограничение, являющееся целым числом *n*, ограничивает вывод первыми *n* строками. Ограничение, являющееся числом с плавающей точкой *f* из диапазона значений от `0.0` до `1.0`, ограничивает вывод соответствующей долей строк. Ограничение, являющееся строкой, компилируется как шаблон регулярного выражения (см. раздел "Регулярные выражения и модуль `re`" в главе 9). В этом случае выводятся лишь строки, соответствующие результату вызова метода `search` для объекта регулярного выражения. Ограничения обладают свойством кумулятивности. Например, вызов *s*`.print_callees(10,0.5)` выводит первые `5` строк (половина от `10`). Ограничения применяются лишь после строк резюме и заголовка: резюме и заголовок выводятся, не подчиняясь никаким ограничениям.

Метод	Описание
	Каждая выводимая функция *f* сопровождается списком функций, вызывавших *f*, или функций, которые вызывались функцией *f*, в соответствии с именем метода
print_stats	*s*.print_stats(**restrictions*)
	Выводит статистическую информацию о данных профилирования, которые хранятся в объекте *s*, отсортированных в соответствии с последним вызовом метода *s*.sort_stats и подчиняющихся заданным ограничениям restrictions, в соответствии с приведенным выше описанием методов print_callees и print_callers. Вслед за несколькими строками резюме (дата и время сбора данных профилирования, количество вызовов функций и используемый критерий сортировки) для каждой (в отсутствие ограничений) функции выводится строка, содержащая шесть полей, названия которых отображаются в строке заголовка. Для каждой функции *f* метод print_stats выводит шесть полей: • общее количество вызовов *f*; • общее время, проведенное в *f*, за исключением других функций, которые вызывала функция *f*; • общее время, затраченное на один вызов *f* (т.е. поле 2, деленное на поле 1); • совокупное время, проведенное в *f* и во всех функциях, прямо или косвенно вызванных из *f*; • совокупное время на один вызов *f* (т.е. поле 4, деленное на поле 1); • имя функции f
sort_stats	*s*.sort_stats(*key*, **keys*)
	Предоставляет, в порядке их приоритета, один или несколько ключей, по которым впоследствии будет выполняться сортировка. Каждый ключ является строкой. Сортировка выполняется в нисходящем порядке для ключей, указывающих время или числа, и в алфавитном порядке для ключа 'nfl'. Наиболее часто используемыми ключами при вызове метода sort_stats являются следующие. 'calls' Количество вызовов функции (подобно полю 1, описанному выше при рассмотрении метода print_stats). 'cumulative' Совокупное время, проведенное в функции и во всех функциях, которые вызывались из нее (подобно полю 4, описанному выше при рассмотрении метода print_stats). 'nfl' Имя функции, ее модуль и номер строки, в которой функция встречается в своем файле (подобно полю 6, описанному выше при рассмотрении метода print_stats).

Метод	Описание
	`'time'` Общее время, проведенное в самой функции, за исключением функций, которые она вызывала (подобно полю 2, описанному выше при рассмотрении метода `print_stats`)
`strip_dirs`	*s*`.strip_dirs()` Изменяет *s*, исключая имена каталогов из имен всех модулей, чтобы сделать будущий вывод более компактным. После вызова *s*`.strip_dirs()` объект *s* остается несортированным, и поэтому обычно за вызовом метода *s*`.strip_dirs` сразу же следует вызов метода *s*`.sort_stats`

Мелкомасштабная оптимизация

Тонкая настройка работы программы может играть важную роль лишь в редких случаях. В некоторых особенно "горячих" хот-спотах она может оказывать хотя и небольшой, но заметный эффект, но этот фактор едва ли способен стать решающим. И все же тонкая настройка, выполняемая в стремлении добиться в общем-то несущественных микроулучшений, является той областью, в которой программист, вероятнее всего, будет руководствоваться своими инстинктами. Во многом именно по этой причине большинство попыток оптимизации оказываются преждевременными, и их следует избегать. Самое большее, что можно сказать в пользу тонкой настройки, это то, что если одна идиома *всегда* работает быстрее, чем другая, когда это различие измеримо, то стоит взять себе за привычку всегда использовать более быстрый путь.

В Python чаще всего бывает так, что если вы действуете естественным образом, отдавая предпочтение простоте и элегантности, то в конечном счете получите код, отличающийся не только хорошей производительностью, но также ясностью и легкостью сопровождения. Иными словами, позвольте Python делать свою работу: если Python предлагает простой и прямой способ решения какой-то задачи, то существует вероятность того, что этот способ решения данной задачи одновременно является самым быстрым. В незначительном количестве случаев подход, который интуитивно не кажется привлекательным, все же может обеспечить некоторые преимущества в производительности, о чем пойдет речь в оставшейся части раздела.

Простейшим способом оптимизации кода является запуск программ на языке Python с помощью команды `python -O` или `-OO`. Разница в производительности между опциями `-OO` и `-O` невелика, но может сэкономить немного памяти, поскольку опция `-OO` удаляет из байт-кода строки документирования, а в некоторых случаях нехватка памяти может (косвенно) ухудшать производительность. Возможности оптимизатора, входящего в состав текущих выпусков Python, не очень велики, однако вы можете получить выигрыш в производительности порядка 5, а иногда и 10% (потенциально даже больше, если использовать инструкции `assert` и `if __debug__`:

в соответствии с рекомендациями, приведенными в разделе "Инструкция assert" главы 5). Наибольшим достоинством опции -O является то, что это ничего не стоит — разумеется, при условии, что ваша оптимизация не является преждевременной (можете не пытаться использовать опцию -O, если программа все еще находится в стадии разработки).

Модуль timeit

Модуль стандартной библиотеки timeit удобно использовать для точных измерений производительности отдельных фрагментов кода. Модуль timeit можно импортировать для того, чтобы использовать его функциональность в своей программе, но для этих целей проще всего использовать командную строку:

```
python -m timeit -s 'инструкция установки' 'инструкции для тайминга'
```

Инструкция установки выполняется только один раз для инициализации переменных. Инструкции для тайминга выполняются многократно с целью измерения среднего времени их выполнения.

Предположим, вы хотите выяснить, какова относительная производительность инструкций x=x+1 и x+=1, где x — целое число. Используя командную строку для выполнения следующих команд, нетрудно убедиться в том, что скорость выполнения обеих операций примерно одинакова.

```
$ python -m timeit -s 'x=0' 'x=x+1'
1000000 loops, best of 3: 0.0416 usec per loop
$ python -m timeit -s 'x=0' 'x+=1'
1000000 loops, best of 3: 0.0406 usec per loop
```

Сборка строк из отдельных элементов

Единственной "антиидиомой" Python, способной снизить производительность программы до такого уровня, что вы не захотите ею воспользоваться, является создание длинной строки из отдельных элементов посредством выполнения в цикле операций конкатенации наподобие *big_string+=piece*. Строки в Python — неизменяемые объекты, поэтому выполнение каждой такой операции означает, что Python должен освободить M байт, ранее выделенных для строки *big_string*, а затем выделить и заполнить M+K байт для новой версии строки. Повторно выполняя эту операцию в цикле, вы получите алгоритм со временем выполнения $O(N^2)$, где N — общее количество символов. В большинстве случаев производительность на уровне $O(N^2)$ там, где вы могли бы обеспечить производительность на уровне O(N), приведет к катастрофе, как бы ни старался Python исправить эту специфическую, ужасную, но тем не менее распространенную антиидиому. На некоторых платформах ситуация может еще сильнее обостриться из-за фрагментация памяти, обусловленной освобождением многочисленных областей памяти все возрастающего размера.

Чтобы достигнуть производительности порядка O(N), соберите промежуточные элементы в список вместо поэлементного достраивания строки. В отличие от строк

списки — изменяемые объекты, поэтому операция присоединения к списку характеризуется порядком сложности O(1) (амортизированным). Замените каждое вхождение инструкции *big_string+=piece* инструкцией *temp_list.append(piece)*. Собрав все элементы в список, используйте следующий код для создания желаемой строки, обеспечивающий результирующую сложность всей операции порядка O(N):

```
big_string = ''.join(temp_list)
```

Используя генераторы списков (списковые включения), выражения-генераторы или другие непосредственно доступные возможности (такие, как вызов функции map или использование модуля itertools стандартной библиотеки) для сборки временного списка *temp_list*, нередко можно обеспечить дополнительную (ощутимую, но не порядка "большого О") оптимизацию по сравнению с многократными вызовами *temp_list*.append. К числу других способов сборки длинных строк со временем выполнения O(N), которые, по мнению некоторых программистов, обеспечивают лучшую читаемость кода, относятся конкатенация элементов в экземпляр массива array.array('u') с помощью метода extend, использование байтового массива bytearray или запись элементов в экземпляр класса io.TextIO либо io.BytesIO.

В тех особых случаях, когда нужно всего лишь вывести результирующую строку, можно получить небольшой дополнительный выигрыш в производительности, применяя к списку *temp_list* метод writelines (даже не создавая строку *big_string* в памяти). В тех случаях, когда это возможно (а именно, когда в цикле доступен открытый объект выходного файла и этот файл буферизуется), почти такую же производительность обеспечивает вызов метода write для каждого из элементов без их аккумуляции.

В качестве другого примера, в котором отказ от конкатенации строк также обеспечивает выигрыш в производительности, хотя и в меньшей степени, чем при отказе от использования оператора += в цикле в случае длинных строк, рассмотрим конкатенацию значений в следующем выражении.

```
oneway = str(x)+' eggs and '+str(y)+' slices of '+k+' ham'
another = '{} eggs and {} slices of {} ham'.format(x, y, k)
```

Подход, основанный на форматировании строк с помощью метода format вместо их конкатенации, часто способен обеспечить небольшой выигрыш в производительности и более привлекателен с точки зрения идиоматичности и ясности кода.

Поиск и сортировка

Порядок времени выполнения для оператора in, наиболее естественного инструмента поиска, составляет O(1), если правым операндом является множество или словарь, но O(N), если правый операнд — строка, список или кортеж. Если вам приходится выполнять множество операций поиска в контейнере, то в качестве контейнера лучше использовать множество или словарь, а не список или кортеж.

Типы set и dict в Python высоко оптимизированы для поиска и извлечения элементов по ключу. Поскольку создание множества или словаря на основе других типов контейнеров характеризуется временем выполнения O(N), при выполнении процедур многократного поиска лучше хранить данные в словарях или множествах, быстро выполняя необходимое преобразование в случае изменения базовой последовательности.

Метод sort списков Python также относится к числу тщательно спроектированных и высоко оптимизированных инструментов. Вы можете полагаться на производительность метода sort. Однако в версии v2 она резко снижается в случае передачи методу sort пользовательского вызываемого объекта для выполнения сравнений (с целью сортировки списка на основании пользовательских критериев). Большинство функций и методов, выполняющих сравнения, поддерживает аргумент key, который используется ими для того, чтобы определить, как именно следует сравнивать элементы между собой. Если у вас есть функция, пригодная только для использования в качестве аргумента cmp, воспользуйтесь функцией functools.cmp_to_key, описанной в табл. 7.4, для создания на ее основе новой функции, пригодной для использования в качестве ключа, и передайте вновь созданную функцию в качестве аргумента key вместо передачи исходной функции в качестве аргумента cmp.

В то же время большинство функций, предоставляемых модулем heapq (см. описание в разделе "Модуль heapq" главы 7), не поддерживает аргумент key. В подобных случаях вы можете использовать идиому "декорирование — сортировка — отмена декорирования" (decorate-sort-undecorate, DSU), рассмотренную в разделе "Идиома "декорирование — сортировка — отмена декорирования"" главы 7. (Кучи не стоит упускать из виду, поскольку иногда они могут избавить вас от необходимости сортировки всех ваших данных.)

Модуль operator предоставляет функции attrgetter и itemgetter, которые особенно хорошо приспособлены для поддержки подхода, основанного на использовании ключей, и позволяют избежать использования лямбда-выражений.

Избегайте использования инструкций exec и from ... import *

Код функций выполняется быстрее кода, находящегося в модуле на самом верхнем уровне, поскольку доступ к локальным переменным функции осуществляется очень быстро. Но если функция содержит инструкцию exec без явно определенных словарей, то ее работа замедляется. Наличие подобных инструкций exec вынуждает компилятор Python избегать умеренной, но весьма важной оптимизации, которую он обычно выполняет в отношении доступа к локальным переменным, поскольку функция exec может изменить пространство имен выполняемой функции. Ухудшает производительность и инструкция from в форме

```
from MyModule import *
```

поскольку она также может изменить пространство имен функции непредсказуемым образом, тем самым препятствуя выполнению оптимизации операций над локальными переменными.

Кроме того, функция `exec` сама по себе работает довольно медленно, но будет работать еще медленнее, если вы примените ее к строке исходного кода, а не к объекту кода. Пока что наилучший подход, как в смысле производительности, так и корректности и ясности, заключается в том, чтобы вообще избегать применения функции `exec`. Чаще всего удается найти лучшие (более быстрые, надежные и ясные) решения. Если вы *должны* применить функцию `exec`, то *всегда* используйте ее вместе с явно определенными словарями. Если требуется выполнить динамически получаемую строку более одного раза, скомпилируйте ее один раз, а затем повторно используйте результирующий объект кода. Однако, если есть такая возможность, лучше вообще отказаться от использования функции `exec`.

Функция `eval` работает не с инструкциями, а с выражениями, поэтому, хотя и работает медленно, не оказывает столь отрицательного влияния на производительность, как функция `exec`. С функцией `eval` также лучше всего использовать явно определенные словари. Если вам нужно несколько раз вычислить одну и ту же динамически получаемую строку, скомпилируйте ее один раз, а затем повторно вычисляйте результирующий объект кода. А еще лучше вообще не использовать функцию `eval`.

Для получения более подробной информации и рекомендаций, касающихся использования функций `exec`, `eval` и `compile`, обратитесь к разделу "Динамическое выполнение и инструкция `exec`" главы 13.

Оптимизация циклов

Большинство узких мест в вашей программе будет связано с циклами, особенно вложенными, поскольку тела циклов выполняются многократно. Python не пытается неявно перемещать код за пределы цикла: если внутри цикла имеется код, который достаточно выполнить лишь один раз, и данный цикл является узким местом, вынесите такой код за пределы цикла самостоятельно. Наличие подходящего для этого кода не всегда является очевидным.

```
def slower(anobject, ahugenumber):
    for i in range(ahugenumber): anobject.amethod(i)
def faster(anobject, ahugenumber):
    themethod = anobject.amethod
    for i in range(ahugenumber): themethod(i)
```

В данном случае код, который в функции `faster` выносится за пределы цикла, осуществляет поиск атрибута `anobject.amethod`. В функции `slower` этот поиск повторяется в цикле каждый раз заново, в то время как в функции `faster` он выполняется только один раз. Эти две функции не являются полностью эквивалентными: вполне возможны (хотя и в редких случаях) ситуации, когда выполнение

метода amethod может приводить к таким изменениям объекта anobject, что в результате очередного поиска того же самого именованного атрибута будет извлечен другой объект метода. Это является одной из причин того, почему Python не выполняет оптимизацию в подобных случаях. На практике такие тонкие, скрытые и хитрые ловушки встречаются крайне редко, так что вы можете смело выполнять оптимизацию, пытаясь выдавить последнюю каплю производительности из "бутылочного горлышка".

Python выполняет операции над локальными переменными быстрее, чем над глобальными. Если в цикле повторно выполняются обращения к глобальной переменной, значение которой между итерациями не изменяется, кешируйте это значение в локальной переменной, к которой и следует обращаться. То же самое касается встроенных объектов.

```
def slightly_slower(asequence, adict):
    for x in asequence: adict[x] = hex(x)
def slightly_faster(asequence, adict):
    myhex = hex
    for x in asequence: adict[x] = myhex(x)
```

В данном случае достигается весьма скромное ускорение — порядка 5% или около того.

Не кешируйте значение None: это ключевое слово, поэтому никакая оптимизация для него не требуется.

Генераторы списков (списковые включения) и выражения-генераторы могут работать быстрее циклов, что иногда справедливо также для отображений и фильтров. Там, где это возможно, старайтесь заменить циклы генераторами списков, выражениями-генераторами или, возможно, вызовами функций map и filter. Преимущества использования функций map или filter падают до нуля и даже могут стать фактором, ухудшающим производительность, если вы используете лямбда-выражения или дополнительный уровень вызова функций. Вы можете рассчитывать на небольшое ускорение лишь в том случае, если передаете функции map или filter встроенную функцию или функцию, которую вы в любом случае вызывали бы даже в явном цикле, генераторе списков или в выражении-генераторе.

К циклам, которые наиболее естественным образом заменяются генераторами списков или вызовами функций map и filter, относятся те, которые создают список путем повторного применения метода append к списку. Этот тип оптимизации проиллюстрирован ниже на примере небольшого сценария для оценки производительности (разумеется, мы могли бы использовать модуль timeit вместо того, чтобы писать собственный код, выполняющий необходимые измерения, но данный пример, в частности, предназначен также для того, чтобы показать, как это делается).

```
import time, operator

def slow(asequence):
```

```
    result = []
    for x in asequence: result.append(-x)
    return result

def middling(asequence):
    return list(map(operator.neg, asequence))

def fast(asequence):
    return [-x for x in asequence]

biggie = range(500*1000)
tentimes = [None]*10
def timit(afunc):
    lobi = biggie
    start = time.clock()
    for x in tentimes: afunc(lobi)
    stend = time.clock()
    return '{:<10}: {:.2f}'.format(afunc.__name__, stend-start)

for afunc in slow, middling, fast, fast, middling, slow:
    print(timit(afunc))
```

При запуске этого примера в версии v2 на старом ноутбуке функция `fast` выполнялась примерно 0,36 секунды, функция `middling` — 0,43 секунды, а функция `slow` — 0,77 секунды. Иными словами, на этом компьютере функция `slow` (цикл вызовов метода `append`) выполнялась на 80% медленнее функции `middling` (одиночный вызов `map`), которая, в свою очередь, выполнялась примерно на 20% медленнее функции `fast` (генератор списка).

Генератор списка позволяет наиболее непосредственным способом выразить задачу, время выполнения которой измеряется в данном примере, поэтому нет ничего удивительного в том, что он оказался самым быстрым — он работает примерно в два раза быстрее, чем вызовы метода `append` в цикле.

Оптимизация ввода-вывода

Если ваша программа интенсивно использует операции ввода-вывода, то вероятнее всего, что именно они, а не вычисления, являются "бутылочным горлышком" производительности. О таких программах говорят, что они *ограничены возможностями ввода-вывода* (I/O-bound), а не *возможностями процессора* (CPU-bound). Ваша операционная система пытается оптимизировать выполнение операций ввода-вывода, но вы можете помочь ей в этом несколькими способами. Один из них заключается в выполнении этих операций с использованием порций данных, размер которых оптимален с точки зрения производительности, а не просто удобен для работы вашей программы. Другой способ заключается в использовании потоков. Часто наилучшим способом является асинхронное выполнение операций ввода-вывода, о чем пойдет речь в главе 18.

Объем данных, являющийся с точки зрения удобства и простоты программ идеальным для единовременного чтения или записи, часто оказывается либо малым (один символ или одна строка), либо очень большим (весь файл целиком за один раз). Во многих случаях это вполне приемлемо: Python и операционная система за кулисами позволяют вашей программе оперировать при выполнении операций ввода-вывода удобными логическими порциями, организуя при этом использование размеров порций данных, обеспечивающих повышение производительности, на уровне физических операций ввода-вывода. Чтение и запись файла целиком за один раз также не оказывает отрицательного влияния на производительность, если только файл *не очень* большой. В частности, такой способ выполнения файловых операций ввода-вывода целесообразно использовать в тех случаях, когда содержащиеся в файле данные могут быть целиком размещены в оперативной памяти, оставляя при этом достаточный объем свободной памяти, чтобы операционная система и другие программы могли беспрепятственно продолжать свою работу в это же время. Большие проблемы с производительностью, обусловленные операциями ввода-вывода, возникают лишь с файлами огромных размеров.

Если проблемы производительности выходят на первый план, *никогда* не используйте файловый метод `readline`: ему присущи ограничения в отношении размеров порций данных и буфера, с которыми он может работать. (С другой стороны, метод `writelines` не создает никаких проблем, связанных с производительностью, и поэтому вы можете совершенно спокойно использовать его в своей программе, если считаете нужным.) При чтении текстовых данных организуйте непосредственный цикл по объекту файла для получения по одной строке за раз с наилучшей производительностью. Если файл имеет небольшой размер и может целиком разместиться в памяти без ущерба для других программ, хронометрируйте выполнение двух версий своей программы: одна использует непосредственный цикл по объекту файла, тогда как другая читает файл целиком в память. Любая из этих версий может оказаться немного быстрее другой.

В случае двоичных файлов, особенно большого размера, из которых при каждом запуске программы вам требуется лишь часть данных, модуль `mmap` (см. раздел "Модуль `mmap`" в главе 14) иногда способен обеспечить хорошую производительность и упростить вашу программу.

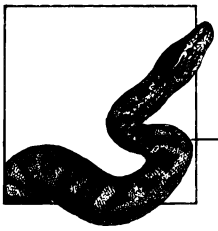
Иногда использование многопоточности в программе, интенсивно использующей операции ввода-вывода, позволяет добиться существенного выигрыша в производительности, если вы можете соответствующим образом реорганизовать архитектуру программы. Запустите несколько потоков, ответственных за выполнение операций ввода-вывода, которым вычислительные потоки посредством экземпляров класса `Queue` могут посылать запросы на выполнение этих операций, и посылайте запрос на выполнение каждой операции ввода данных сразу же, как только становится известно, что эти данные потребуются в конечном счете. Производительность возрастет только в том случае, если существуют другие задачи, которые могут выполняться вычислительными потоками вашей программы в то время, когда потоки

ввода-вывода блокируются в ожидании данных. Очевидно, что улучшение производительности таким способом можно получить, лишь если вы в состоянии управлять перекрыванием вычислений и ожидания данных, используя разные потоки для выполнения вычислений и ожидания. (Для получения более подробной информации относительно многопоточности и рекомендуемой архитектуры программ на языке Python обратитесь к разделу "Потоки в Python" главы 14.)

С другой стороны, не исключено, что вы добьетесь еще большего быстродействия и улучшите масштабируемость своих программ, если воздержитесь от использования потоков и предпочтете им асинхронные (управляемые событиями) архитектуры, рассматриваемые в главе 18.

IV

Сетевое и веб-программирование



17

Основы работы с сетями

Современные распределенные компьютерные системы интенсивно используют сети. Понимание принципов обмена данными по сети необходимо для создания подобных систем, поскольку на них основана работа любой сетевой службы. *Протокол* — это своеобразный язык, используемый по договоренности двумя сторонами (роль которых в наши дни часто играют компьютерные программы) для общения между собой. Существуют два основных способа взаимодействия по сети — с установлением и без установления соединения (раздел "Принципы организации сетей" на стр. 951), каждый со своим протоколом. Оба способа могут работать, используя различные транспортные механизмы, благодаря универсальности стека протоколов TCP/IP и интерфейса *сокетов*, первоначально предназначенного для поддержки сетей в системах BSD Unix.

С целью разделения различных функций общая задача обмена данными по сети разбивается на несколько *слоев (уровней) протоколов*. Данные могут передаваться от одной системы к другой многими способами — с помощью сети Ethernet, последовательной линии связи и т.п., и обработка всех этих различий в коде приложения привела бы к его чрезмерному усложнению. Стандартом ISO (International Standards Organization) определена семиуровневая модель, которая, как показала практика, является излишне сложной без всякой на то необходимости. В стеке протоколов TCP/IP используется четырехуровневая модель.

Код приложения реализует *прикладной уровень*, который обеспечивает обмен сообщениями между двумя процессами, потенциально выполняющимися на разных компьютерах (хотя набор протоколов TCP/IP точно так же работает для двух локальных процессов). Этот уровень передает сообщения *транспортному уровню*, обрабатывающему связь между хостами. Транспортный протокол разбивает длинные сообщения на сегменты, которые передаются *сетевому уровню*, ответственному за маршрутизацию сегментов (разбивку их на порции данных, известные как *датаграммы*) через рассчитанную последовательность транзитных участков. В свою очередь, сетевой уровень использует *канальный уровень*, или *уровень подсетей*, для передачи

датаграмм между отдельными системами на пути от источника сообщения к пункту его назначения.

Приложение, посылающее данные, передает порцию данных транспортному уровню. В свою очередь, этот уровень передает сегментированные данные сетевому уровню, который на основании знания структуры локальной сети выбирает канал для пересылки данных на первый транзитный узел и размер датаграмм для разбивки сегментов. Используя соответствующий драйвер, сетевой уровень передает датаграммы в виде пакетов через выбранный канал передачи данных. Каждая промежуточная система извлекает датаграммы из пакетов канального уровня и перенаправляет их следующей системе обернутыми в новый протокол канального уровня для своего следующего промежуточного узла. Наконец, каждая датаграмма прибывает в свой конечный пункт назначения, где сетевой уровень вновь передает его транспортному уровню, который осуществляет повторную сборку сегментов, в конечном счете доставляя результирующее сообщение процессу приложения, для которого оно предназначено.

В этой главе рассмотрены принципы организации работы в сети, модуль `socket` и основное подмножество функциональности модуля `ssl`: этого вам будет достаточно для написания небольших сетевых клиентов и серверов, а также, что более важно, для правильного использования сетевых модулей на более высоких уровнях абстракции, чему посвящена глава 19, и асинхронных архитектур, о которых пойдет речь в главе 18.

Принципы организации сетей

Работу протоколов, *ориентированных на предварительное установление соединения*, можно сравнить с разговором по телефону. Вы запрашиваете соединение с конкретной конечной точкой сети (эквивалент набора телефонного номера), и вторая сторона либо отвечает на ваш звонок, либо не отвечает. Если ваш звонок принят, вы можете либо говорить сами, либо выслушивать ответ (в случае необходимости это может происходить одновременно), поэтому вы знаете, что в процессе вашего общения никакая часть информации не теряется. В конце разговора вы оба прощаетесь друг с другом и кладете на свои места телефонные трубки, и если эти завершающие действия по какой-то причине не были выполнены (например, потому что вы внезапно перестали слышать своего собеседника), то это сразу же говорит о том, что что-то пошло не так. Основным транспортным протоколом Интернета, ориентированным на установление соединения, который используется веб-браузерами, командными оболочками, электронной почтой и многими другими приложениями, является TCP.

Работа протоколов, *не ориентированных на установление соединения*, напоминает процесс пересылки почтовых карточек. В большинстве случаев сообщения успешно доходят до адресатов, но если что-то пошло не так, вы должны самостоятельно находить выход из этого затруднения — протокол не предусматривает уведомления вас о том, что ваши сообщения не были доставлены. При обмене короткими

сообщениями и получении ответов на них протокол датаграмм требует меньших накладных расходов по сравнению с протоколами, требующими установления соединения, поэтому он пригоден для тех ситуаций, когда служба в целом может справиться с последствиями случайных сбоев, например в случае неполучения ответа от сервера службы доменных имен (DNS): большинство сообщений DNS передается без установления соединений. Основным транспортным протоколом при передаче данных через Интернет без установления соединения является UDP.

В наши дни вопросы безопасности приобретают все большее значение: понимание принципов безопасного обмена данными позволит вам быть уверенным в том, что ваши сообщения защищены, как это и должно быть. Таким образом, если мы убедили вас в том, что не стоит пытаться самостоятельно реализовать подобные технологии без досконального знания сути проблемы и возможных рисков, то мы достигли своей цели.

Все виды взаимодействия по сети предполагают обмен байтовыми строками (*октетами*, если использовать сетевую терминологию). Например, передавая текст, отправитель должен кодировать его в байты, которые получатель должен декодировать.

Интерфейс сокетов Беркли

В настоящее время для обмена данными по сети в большинстве случаев используются *сокеты*, которые предоставляют доступ к каналам передачи данных между независимыми конечными точками, используя *транспортный уровень* для перемещения информации. Понятие сокета имеет достаточно общий смысл и охватывает конечные точки, которые могут находиться как на одной машине, так и на разных компьютерах, объединенных в рамках локальной или глобальной сети. Между подходами к программированию сетевого обмена в обоих случаях нет никаких различий.

Наиболее типичные протоколы транспортного уровня — UDP (User Datagram Protocol — протокол пользовательских датаграмм) и TCP (Transmission Control Protocol — протокол управления передачей), работающие поверх сетевого IP-протокола. Эта комбинация протоколов вместе со многими протоколами прикладного уровня, которые выполняются поверх них, известна под общим названием стека протоколов *TCP/IP*. Операционная система Unix предлагает собственную разновидность сокетов для обмена данными между процессами на одном компьютере. Такие сокеты Unix могут использоваться для любого типа взаимодействия по сети.

Интерфейс сокетов изначально был реализован в системе BSD Unix и оказался весьма полезным механизмом сетевого взаимодействия, стандартизируя структуру сетевых программ. Сокеты лежат в основе сетевого программирования на языке Python, поэтому очень важно, чтобы вы понимали, как они работают. Для первоначального ознакомления с сокетами можно порекомендовать онлайн-руководство *Socket Programming How-To* (`https://docs.python.org/3/howto/sockets.html`).

Существуют две наиболее распространенные разновидности сокетов (известные как *семейства сокетов*): *интернет-сокеты*, основанные на обмене данными

по протоколу TCP/IP (доступны два варианта, соответствующие более современной, IPv6, и традиционной, IPv4, версиям протокола IP), и *сокеты Unix*, хотя имеются и другие разновидности. Интернет-сокеты обеспечивают обмен сообщениями между любыми двумя компьютерами, способными обмениваться IP-датаграммами. Сокеты Unix обеспечивают обмен сообщениями только между процессами, выполняющимися на одном и том же компьютере Unix.

Для поддержки одновременной работы многих интернет-сокетов стек протоколов TCP/IP использует конечные точки, идентифицируемые IP-адресом, *номером порта* и протоколом. Номер порта позволяет программному обеспечению, обрабатывающему протокол, различать конечные точки, которые имеют один и тот же IP-адрес и используют один и тот же протокол. Подключенный сокет работает совместно с *удаленной конечной точкой* — другим сокетом, с которым было установлено соединение и с которым он может обмениваться сообщениями.

Большинство сокетов Unix ассоциируются с именами в файловой системе Unix. На платформах Linux сокеты, имена которых начинаются с нулевого байта, связаны с пулом имен, поддерживаемым ядром. Это удобно использовать для обмена сообщениями с процессами, изолированными с помощью методики *chroot-jail* (`https://unix.stackexchange.com/questions/105/chroot-jail-what-is-it-and-how-do-i-use-it`), например, в условиях отсутствия файловых систем, совместно используемых двумя процессами.

И интернет-, и Unix-сокеты могут работать как с установлением, так и без установления соединения, поэтому тщательно продуманные программы способны работать поверх любого семейства сокетов. Рассмотрение других семейств сокетов выходит за рамки данной книги, однако отметим, что так называемые *простые сокеты* (raw sockets), подтип семейства интернет-сокетов, обеспечивают отправку и получение непосредственно пакетов канального уровня (например, пакетов Ethernet). Это может быть полезным для экспериментальных приложений, но некоторые операционные системы семейства Windows делают все возможное для того, чтобы воспрепятствовать доступу программистов к простым сокетам под тем благовидными предлогом, что это делается для того, чтобы отвадить хакеров.

Создав интернет-сокет, вы можете *привязать* (ассоциировать) его к конкретному порту (при условии, что он не используется другим сокетом). Многие серверы используют эту стратегию, предлагая доступ к службам через *порты с известными номерами*, определенные стандартом Интернета как порты с номерами в диапазоне 1–1023 (в Unix-системах для доступа к этим портам требуются привилегии пользователя *root*, или *суперпользователя*). Типичного клиента не интересует, какой номер порта он использует, поэтому обычно он запрашивает порт для временного использования, номер которого назначается и уникальность которого гарантируется драйвером протокола. Такие временные порты не нуждаются в связывании.

Представьте себе два процесса, выполняющиеся на одном компьютере, которые оба выступают в качестве клиентов одного и того же удаленного сервера. Полное связывание для каждого сокета представляется в виде `(local__IP_address,`

`local_port_number, protocol, remote_IP_address, remote_port_number)`. Когда трафик достигает сервера, целевой IP-адрес, номер конечного порта, протокол и исходный IP-адрес в точности совпадают для обоих клиентов. Однако гарантированная уникальность номеров временных портов позволяет серверу различать трафик, поступающий от каждого из двух клиентов. Именно так работает *мультиплексирование TCP/IP-протокола*, обеспечивающее нормальную работу двух клиентов с одинаковыми IP-адресами.

Адреса сокетов

Разные типы сокетов используют разные форматы адресов.

Адреса сокетов Unix представляют собой строки с именами узлов в файловой системе (на платформах Linux строки, начинающиеся с символов `b'\0'`, соответствуют именам в таблице ядра).

Адреса сокетов IPv4 представляют собой пары `(address, port)`. Первый элемент — адрес IPv4, второй — номер порта в диапазоне `1-65535`.

Адреса сокетов IPv6 представляют собой кортежи из четырех элементов вида `(address, port, flowinfo, scopeid)`. Предоставляя адрес в качестве аргумента, элементы `flowinfo` и `scopeid` можно опустить, однако иногда это может порождать определенные проблемы.

Архитектура "клиент — сервер"

Шаблон проектирования, к обсуждению которого мы приступаем, обычно называют архитектурой "клиент — сервер", в которой *сервер* прослушивает трафик, поступающий на конкретную конечную точку от *клиентов*, желающих использовать данную службу. Мы не рассматриваем *одноранговые сети* (peer-to-peer), которые (ввиду отсутствия центрального сервера) должны обеспечивать возможность того, чтобы равноправные узлы могли обнаруживать друг друга. По иронии судьбы, зачастую это достигается за счет вхождения в контакт с центральным сервером, хотя такие протоколы обнаружения, как SSDP, обеспечивают полную независимость от каких бы то ни было центральных элементов.

В большинстве случаев, хотя ни в коей мере не во всех, сетевой обмен сообщениями осуществляется с использованием клиент-серверных методик. Сервер прослушивает входящий трафик через предопределенную или объявленную конечную точку сети. В отсутствие входного трафика он не предпринимает никаких действий и просто ждет поступления запросов от клиентов. Связь осуществляется по-разному для конечных точек, устанавливающих и не устанавливающих соединение.

При работе с протоколами, не требующими предварительного установления соединения, такими как UDP, запросы поступают на сервер в случайном порядке и немедленно обрабатываются: ответ посылается запрашивающей стороне без задержки. Каждое сообщение обрабатывается независимо от других, обычно без ссылок на какой-либо возможный предыдущий процесс обмена сообщениями между двумя

сторонами. Таким образом, сетевой обмен сообщениями без установления соединений хорошо приспособлен для кратковременного взаимодействия без сохранения состояния, как в случае службы DNS или процесса начальной загрузки сети.

При работе с протоколами, ориентированными на установление соединения, клиент инициирует начальный обмен данными с сервером, который в конечном счете устанавливает соединение посредством сетевого канала связи между двумя процессами (иногда его называют *виртуальным каналом*). Через этот канал процессы могут обмениваться сообщениями до тех пор, пока оба не изъявят желание завершить сеанс связи. Обслуживание запросов в таких условиях требует использования параллелизма на основе механизмов одновременного выполнения (таких, как потоки и процессы, рассмотренные в главе 14, или асинхронное программирование, о котором пойдет речь в главе 18) для асинхронной или одновременной обработки каждого входящего соединения. Без такого параллелизма сервер не смог бы обрабатывать новые входящие соединения, не завершив работу с предыдущими, поскольку вызовы методов сокета обычно являются *блокирующими* (в том смысле, что они заставляют вызвавший их поток простаивать, пока они не завершатся или пока не истечет выделенный лимит времени — тайм-аут). Соединения — наилучший способ обработки длительных видов взаимодействия, таких как пересылка электронной почты, интерактивное взаимодействие с командной оболочкой, передача веб-содержимого, а также автоматическое обнаружение и устранение ошибок при использовании протокола TCP.

Клиентские и серверные структуры, не требующие установления соединения

Общая логика работы сервера без установления соединения соответствуют следующей процедуре.

1. Создать сокет типа `socket.SOCK_DGRAM` посредством вызова функции `socket.socket`.
2. Связать сокет с конечной точкой службы посредством вызова метода `bind` сокета.
3. Повторять следующие действия в *бесконечном* цикле:
   а) запросить у клиента датаграмму посредством вызова метода сокета `recvfrom` (этот вызов блокируется до получения датаграммы);
   б) вычислить результат;
   в) отослать результат обратно клиенту посредством вызова метода сокета `sendto`; большую часть времени сервер проводит в состоянии, описанном в п. 3а, ожидая поступления входных данных от клиентов.

Общая логика работы клиента без установления соединения соответствует следующей процедуре.

1. Создать сокет типа `socket.SOCK_DGRAM` посредством вызова функции `socket.socket`.
2. По выбору: связать сокет с конкретной конечной точкой посредством вызова метода `bind` сокета.
3. Послать запрос конечной точке сервера посредством вызова метода сокета `sendto`.
4. Ожидать ответа от сервера посредством вызова метода сокета `recvfrom`. Этот вызов блокируется до получения ответа. В таком вызове всегда следует использовать аргумент тайм-аута, чтобы обеспечить обработку в тех случаях, когда датаграмма не доходит до адреса, и требуется либо повторить попытку, либо отказаться от этого: сокеты, работающие без установления соединения, не гарантируют доставку.
5. Результат используется в логике оставшейся части программы.

Одна клиентская программа может осуществлять несколько взаимодействий с одним и тем же или несколькими серверами, в зависимости от служб, которые она собирается использовать. Многие из таких взаимодействий скрыты от программиста в коде библиотеки. В качестве типичного примера можно привести разрешение имени хоста в соответствующий сетевой адрес, для чего обычно используют библиотечную функцию `gethostbyname` (реализованную в модуле `socket`). Как правило, взаимодействие без установления соединения включает отправку одиночного пакета серверу и получение одиночного пакета в качестве ответа. Основным исключением из этого правила являются потоковые протоколы, такие как RTP, которые обычно работают поверх протокола UDP с целью минимизации задержек и периодов ожидания: в этом случае осуществляется отправка и получение большого количества датаграмм.

Клиентские и серверные структуры, ориентированные на установление соединения

Общая логика работы сервера с установлением соединения соответствует следующей процедуре.

1. Создать сокет типа `socket.SOCK_STREAM` посредством вызова функции `socket.socket`.
2. Связать сокет с подходящей серверной конечной точкой посредством вызова метода сокета `bind`.
3. Запустить в конечной точке прослушивание запросов на создание соединения посредством вызова метода сокета `listen`.
4. Повторять следующие действия в *бесконечном* цикле.
   A. Ожидать поступления запроса от клиента на создание соединения посредством вызова метода сокета `accept`. Процесс сервера блокируется

до получения запроса на создание соединения. При получении такого запроса создается новый объект сокета, другой конечной точкой которого является клиентская программа.

Б. Создать асинхронный поток управления для обработки данного соединения, передав ему вновь созданный сокет. После этого основной поток продолжает выполнение цикла с п. 4, А.

В. Взаимодействовать с клиентом в новом потоке управления, используя методы нового сокета `recv` и `send` соответственно для чтения данных, поступающих от клиента, и отправки данных клиенту. Метод `recv` блокируется до тех пор, пока не будут доступны данные от клиента (или пока клиент не укажет, что он хочет закрыть соединение, и в этом случае вызов метод `recv` возвращает пустой результат). Метод `send` блокируется лишь в том случае, если сетевым программным обеспечением буферизовано такое количество данных, что необходимо дождаться, пока транспортный уровень не освободит какую-то часть своей буферной памяти. Если сервер хочет закрыть соединение, он может сделать это, вызвав метод сокета `close`, возможно, предварительно вызвав метод `shutdown`.

Большую часть времени сервер выполняет действия, описанные в п. 4, А, ожидая поступления запросов на создание соединения от клиентов.

Общая логика работы клиента с установлением соединения соответствует следующей процедуре.

1. Создать сокет типа `socket.SOCK_STREAM` посредством вызова функции `socket.socket`.
2. По выбору: связать сокет с конкретной конечной точкой посредством вызова метода сокета `bind`.
3. Установить соединение с сервером посредством вызова метода сокета `connect`.
4. Взаимодействовать с сервером, используя методы сокета `recv` и `send` соответственно для чтения данных, поступающих от сервера, и отправки данных серверу. Метод `recv` блокируется до тех пор, пока не будут доступны данные от сервера (или пока сервер не укажет, что он хочет закрыть соединение, и в этом случае вызов метода `recv` возвращает пустой результат). Метод `send` блокируется лишь в том случае, если сетевым программным обеспечением буферизовано такое количество данных, что необходимо дождаться, пока транспортный уровень не освободит какую-то часть своей буферной памяти. Если клиент хочет закрыть соединение, он может сделать это, вызвав метод сокета `close`, возможно, предварительно вызвав метод `shutdown`.

Обычно взаимодействие с установлением соединения отличается большей сложностью по сравнению с взаимодействием без установления соединения. В частности,

установить, когда именно необходимо выполнять чтение и запись данных, в этом случае труднее, поскольку для того, чтобы определить, что передача данных с другого конца сокета завершена, необходимо проанализировать данные. Протоколы, используемые при обмене данными с установлением соединения, должны обеспечивать возможность такого определения, что иногда достигается за счет включения информации о размере данных в качестве части содержимого, а иногда и более сложными методами.

Модуль socket

Модуль `socket` Python обеспечивает обмен данными по сети с помощью интерфейса сокетов. Существуют некоторые различия между платформами, но данный модуль скрывает большинство из них, благодаря чему написание портируемых сетевых приложений превращается в относительно несложную задачу.

Модуль `socket` определяет четыре исключения: базовый класс `socket.error` (который в версии v2 является подклассом класса `exceptions.IOError`, а в версии v3 оставлен как псевдоним класса `exceptions.OSError`, не рекомендуемый к использованию) и три исключения, являющиеся его подклассами, которые описаны ниже.

herror Исключение `socket.herror` возбуждается в случае возникновения ошибок при попытке разрешения имен хостов, а именно: если не удается преобразовать заданное имя хоста в сетевой адрес с помощью функции `socket.gethostbyname` или определить имя хоста по заданному сетевому адресу функции `socket.gethostbyaddr`. Соответствующее значение представляет собой кортеж из двух элементов (*`h_errno`*, *`string`*), где *`h_errno`* — целочисленный код ошибки, возвращаемый операционной системой, а *`string`* — строка, содержащая описание ошибки.

gaierror Исключение `socket.gaierror` возбуждается в случае возникновения ошибок адресации в функциях `getaddrinfo` и `getnameinfo`.

timeout Исключение `socket.timeout` возбуждается, если время выполнения операции превышает длительность предельного времени ожидания (значение этого параметра, устанавливаемое с помощью функции `setdefaulttimeout`, можно изменить для каждого сокета по отдельности).

Кроме того, модуль `socket` определяет большой набор констант, наиболее важные из которых — семейства адресов (`AF_*`) и типы сокетов (`SOCK_*`), описанные ниже. В версии v2 эти константы являются целыми числами, а в версии v3 — элементами коллекции `IntEnum`. На это различие можно не обращать внимания, но, например, при отладке кода значение `<SocketKind.SOCK_STREAM: 1>` гораздо более информативно, чем простое число `1`. Модуль `socket` также определяет многие другие константы, используемые для настройки параметров сокета, но в документации они описаны не полностью: их использование требует знакомства с документацией низкоуровневых библиотек и системных вызовов для работы с сокетами, написанных на языке C.

`AF_INET`	Используется для создания сокетов семейства адресов IPv4.
`AF_INET6`	Используется для создания сокетов семейства адресов IPv6.
`AF_UNIX`	Используется для создания сокетов семейства адресов Unix. Эта константа определена лишь на платформах, делающих доступными сокеты Unix, и поэтому они, например, не доступны для пользователей Windows.
`AF_CAN`	Используется (только в версии v3) для создания сокетов для семейства адресов Controller Area Network (CAN). (Далее не рассматривается, но широко используется приложениями встроенных устройств.)
`SOCK_STREAM`	Используется для создания сокетов, ориентированных на установление соединения, которые предоставляют полный набор средств для обнаружения и устранения ошибок.
`SOCK_DGRAM`	Используется для создания сокетов, не требующих установления соединения, которые делают все возможное для доставки сообщений без использования соединений и средств обнаружения ошибок.
`SOCK_RAW`	Используется для создания сокетов, предоставляющих непосредственный доступ к драйверам канального уровня, используемым в типичных случаях для реализации низкоуровневых сетевых средств, рассмотрение которых выходит за рамки данной книги.
`SOCK_RDM`	Используется для создания надежных, не требующих установления соединения сокетов, используемых в протоколе TIPC, рассмотрение которого выходит за рамки данной книги.
`SOCK_SEQPACKET`	Используется для создания надежных, ориентированных на установление соединения сокетов, используемых в протоколе TIPC, рассмотрение которого выходит за рамки данной книги.

Модуль `socket` определяет ряд функций, предназначенных для создания сокетов, манипулирования адресной информацией и оказания содействия в представлении данных стандартным способом. Учитывая обширность документации модуля `socket`(`https://docs.python.org/3/library/socket.html`), мы не ставили перед собой задачу рассмотреть весь круг его возможностей в данной книге. Мы обсуждаем лишь те из них, которые имеют наибольшее значение для написания сетевых приложений.

Функции общего назначения модуля `socket`

Модуль `socket` содержит немалое количество функций, однако большинство из них полезно лишь в специфических ситуациях. Если обмен данными осуществляется между конечными точками сети, то компьютеры на каждом конце могут иметь архитектурные различия и потому по-разному представлять одни и те же данные, в связи с чем предусмотрены, например, функции, преобразующие ограниченное количество типов данных в нейтральную сетевую форму и наоборот. Ниже описаны некоторые функции более общего назначения.

Таблица 17.1. Функции общего назначения, определенные в модуле `socket`

Функция	Описание
`getaddrinfo`	`socket.getaddrinfo(`*`host`*`, `*`port`*`, `*`family`*`=0, `*`type`*`=0, `*`proto`*`=0, `*`flags`*`=0)` Принимает аргументы *`host`* и *`port`* и возвращает список кортежей (*`family`*, *`type`*, *`proto`*, *`canonical_name`*, *`socket`*), которые могут использоваться для создания соединений сокета с конкретной службой. В версии v2 все аргументы позиционные, но в версии v3 допускаются именованные аргументы. Элемент *`canonical_name`* — пустая строка, если в аргументе *`flags`* не был установлен бит `socket.AI_CANONNAME`. Если передается имя хоста, а не IP-адрес, функция возвращает список кортежей, по одному для каждого IP-адреса, связанного с именем
`getdefaulttimeout`	`socket.getdefaulttimeout()` Возвращает предельное время ожидания в секундах для операций, выполняемых сокетом, или значение `None`, если еще не установлено никакое значение. Некоторые функции позволяют явно задавать длительность тайм-аута
`getfqdn`	`socket.getfqdn([`*`host`*`])` Возвращает полностью уточненное доменное имя, ассоциированное с именем хоста или сетевым адресом (по умолчанию возвращается информация о компьютере, на котором вызывается данная функция)
`gethostbyaddr`	`socket.gethostbyaddr(`*`ip_address`*`)` Принимает строку, содержащую адрес IPv4 или IPv6, и возвращает кортеж (*`hostname`*, *`aliaslist`*, *`ipaddrlist`*). Аргумент *`hostname`* — каноническое имя хоста, соответствующее заданному IP-адресу, *`aliaslist`* — список альтернативных имен, *`ipaddrlist`* — список адресов IPv4 и IPv6
`gethostbyname`	`socket.gethostbyname(`*`hostname`*`)` Возвращает строку, содержащую адрес IPv4, ассоциированный с заданным именем хоста. Если при вызове ей передается IP-адрес, она возвращает его. Эта функция не поддерживает адреса IPv6 — для них следует использовать функцию `getaddrinfo`
`getnameinfo`	`socket.getnameinfo(`*`sock_addr`*`, `*`flags`*`=0)` Принимает адрес сокета и возвращает пару (*`host`*, *`port`*). Если аргумент *`flags`* не передается, то элемент *`host`* представляет собой IP-адрес, а элемент *`port`* является целым числом
`setdefaulttimeout`	`socket.setdefaulttimeout(`*`timeout`*`)` Устанавливает для сокетов предельное время ожидания в секундах в виде значения с плавающей точкой. Вновь создаваемые сокеты работают в режиме, определяемом значением *`timeout`*, что обсуждается в следующем разделе. Чтобы отменить использование тайм-аутов для сокетов, создаваемых впоследствии, передайте в качестве аргумента *`timeout`* значение `None`

Объекты сокетов

Объект сокета является основным средством сетевой коммуникации в Python. Новый сокет также создается, когда сокет типа `SOCK_STREAM` принимает соединение, причем каждый такой сокет используется для обмена данными с соответствующим клиентом.

Объекты сокетов и инструкции `with`

Каждый объект сокета является менеджером контекста, поэтому любой объект сокета можно использовать в начальном предложении инструкции `with` для гарантии корректного завершения работы сокета при выходе из тела инструкции `with`.

Для создания сокетов существует целый ряд способов, которые подробно описаны в следующем разделе. Сокет может работать в различных режимах, определяемых значением тайм-аута, которое может быть установлено одним из трех способов:

- предоставлением значения тайм-аута во время создания сокета;
- вызовом метода `settimeout` объекта сокета;
- в соответствии с установленным по умолчанию для модуля `socket` значением тайм-аута, возвращаемым функцией `socket.getdefaulttimeout`.

Описание режимов работы сокетов, устанавливаемых в зависимости от значения тайм-аута, приведено ниже.

`None` Устанавливает *блокирующий режим*. Каждая операция приостанавливает процесс (*блокируется*) до тех пор, пока она не завершится, если только операционная система не возбудит исключение.

`0` Устанавливает *неблокирующий* режим. Каждая операция возбуждает исключение, если она не может быть немедленно завершена или если возникает ошибка. Чтобы определить, может ли операция быть немедленно завершена, используйте модуль `selectors`, рассмотренный в разделе "Модуль `selectors`" главы 18.

`>0.0` Устанавливает режим *тайм-аута*. Каждая операция блокируется до своего завершения, или до истечения предельного времени ожидания (в этом случае возбуждается исключение `socket.timeout`), или до возникновения ошибки.

Функции для создания сокетов

Объекты сокетов представляют конечные точки сети. Для создания сокетов в модуле `socket` есть ряд функций, которые описаны ниже.

`create_connection` `create_connection([address, [timeout, [source_address]]])`

Создает сокет, подключенный к конечной точке TCP по адресу, заданному с помощью аргумента *`address`* (пара (*`host`*, *`port`*)). Элементом *`host`* может быть либо числовой сетевой адрес, либо

доменное имя хоста. В последнем случае делается попытка разрешить имя как в адрес семейства AF_INET, так и в адрес семейства AF_INET6, а затем делается попытка подключения по каждому из возвращенных адресов по очереди — это весьма удобно для создания клиентских программ, использующих любой из адресов IPv6 или IPv4.

Аргумент *timeout*, если он предоставлен, задает предельное время ожидания соединения в секундах и тем самым устанавливает режим работы сокета. Если этот аргумент отсутствует, то для определения его значения вызывается функция socket.getdefaulttimeout Аргументом *source_address*, если он предоставлен, должна быть пара (*host*, *port*), которую удаленный сокет получает в качестве конечной точки соединения. Если элемент *host* имеет значение '' или элемент *port* равен 0, то используется поведение по умолчанию, установленное операционной системой.

socket socket(*family*=AF_INET, *type*=SOCK_STREAM, *proto*=0, *fileno*=None)

Создает и возвращает объект сокета, использующий указанные семейство адресов и тип (по умолчанию — TCP-сокет, использующий адрес IPv4). Номер протокола proto используется только с сокетами CAN. В случае передачи аргумента fileno другие аргументы игнорируются: функция возвращает сокет, уже связанный с данным дескриптором файла. Сокет не наследуется дочерними процессами.

socketpair socketpair([*family*[, *type*[, *proto*]]])

Возвращает пару сокетов, использующих указанные семейство адресов, тип и (только для сокетов CAN) номер протокола. Если семейство адресов не указано, то сокеты относятся к семейству AF_UNIX на платформах, где оно доступно; в противном случае — к семейству AF_INET. Если аргумент *type* не указан, он принимает заданное по умолчанию значение SOCK_STREAM.

Ниже представлены методы объекта сокета *s* (из которых методы, обрабатывающие соединения или требующие наличия подключенных сокетов, работают только для сокетов SOCK_STREAM, тогда как другие работают и с сокетами SOCK_STREAM, и с сокетами SOCK_DGRAM). В следующей таблице точный состав набора доступных флагов зависит от конкретной платформы. Доступные значения флагов документированы на страницах руководства Unix, содержащих описания вызовов функций forrecv(2) и send(2).

accept accept()

Блокируется до тех пор, пока клиент не установит соединение с объектом сокета *s*, который должен быть привязан к адресу (с помощью вызова метода *s*.bind) и прослушивать соединения

(с помощью вызова метода `s.listen`). Возвращает новый объект сокета, который может использоваться для обмена сообщениями с другой конечной точкой соединения.

bind `bind(address)`

Выполняет привязку `s` к указанному адресу. Форма аргумента `address` зависит от семейства адресов, используемого сокетом (см. раздел "Адреса сокетов").

close `close()`

Помечает сокет как закрытый. В зависимости от наличия других существующих ссылок на данный сокет это не обязательно приводит к немедленному закрытию соединения. Если требуется немедленное закрытие соединения, необходимо предварительно вызвать метод `s.shutdown`. Простейший способ гарантировать своевременное закрытие совета заключается в том, чтобы использовать его вместе с инструкцией `with`, поскольку сокеты являются менеджерами контекста.

connect `connect(address)`

Устанавливает соединение с удаленным узлом по указанному адресу. Форма аргумента `address` зависит от семейства адресов (см. раздел "Адреса сокетов").

detach `detach()`

Переводит сокет в закрытое состояние, но допускает повторное использование объекта сокета для последующих соединений.

dup `dup()`

Возвращает дубликат сокета, не наследуемый дочерними процессами.

fileno `fileno()`

Возвращает файловый дескриптор сокета.

get_inheritable `get_inheritable()` (только в версии v3)

Возвращает значение `True`, если сокет будет наследоваться дочерними процессами. В противном случае возвращает значение `False`.

getpeername `getpeername()`

Возвращает адрес удаленной конечной точки, к которой подключен данный сокет.

getsockname `getsockname()`

Возвращает адрес, используемый данным сокетом.

gettimeout `gettimeout()`

Возвращает текущее предельное время ожидания для данного сокета.

listen `listen([backlog])`

Переводит сокет в режим ожидания трафика на связанной с ним конечной точке. Целочисленный аргумент *backlog*, если он задан, определяет максимальное количество непринятых запросов, которые операционная система разрешает поместить в очередь, прежде чем новые запросы начнут отвергаться.

makefile `makefile(mode, [bufsize])` (v2)

`makefile(mode, buffering=None, *, encoding=None, newline=None)` (v3)

Возвращает объект файла, позволяющий использовать объект сокета в файловых операциях, таких как чтение и запись. Операции могут выполняться в режиме `'r'` или `'w'` с добавлением символа `'b'` в случае передачи двоичных данных. Сокет должен быть настроен для работы в блокирующем режиме. Если установлено значение тайм-аута, то по истечении предельного времени ожидания могут наблюдаться непредсказуемые результаты. Рекомендуется, чтобы библиотеки, ориентированные на одновременную поддержку версий v2 и v3, игнорировали остальные аргументы, поскольку они недостаточно хорошо документированы и по-разному работают в различных версиях.

recv `recv(bufsiz, [flags])`

Получает из сокета максимум *bufsiz* байтов данных. Возвращает полученные данные.

recvfrom `recvfrom(bufsiz, [flags])`

Получает из объекта *s* максимум *bufsiz* байтов данных. Возвращает пару `(bytes, address)`, где элемент *bytes* — полученные данные, а элемент *address* — адрес сокета, отправившего данные.

recvfrom_into `recvfrom_into(buffer, [nbytes, [flags]])`

Получает из объекта *s* максимум *bufsiz* байтов данных и записывает их в указанный объект *buffer*. Возвращает кортеж `(nbytes, address)`, где *nbytes* — количество полученных байтов, а *address* — адрес сокета, отправившего данные.

recv_into `recv_into(buffer, [nbytes, [flags]])`

Получает из объекта *s* максимум *bufsiz* байтов данных и записывает их в указанный объект *buffer*. Возвращает количество полученных байтов.

recvmsg `recvmsg(bufsiz, [ancbufsiz, [flags]])`

Получает максимум *bufsiz* байтов обычных данных сокета и максимум *ancbufsiz* байтов вспомогательных, или *внеполосных* (out-of-band), данных. Возвращает кортеж `(data, ancdata, msg_flags, address)`, где *data* — полученные данные, *ancdata* — список `(cmsg_level, cmsg_type, cmsg_data)`, представляющий полученные вспомогательные данные, *msg_flags* хранит флаги,

полученные вместе с сообщением, а *address* — адрес сокета, отправившего данные (если сокет подключен, то это значение не определено, но отправителя можно определить на основании хранящейся в сокете информации).

send `send(`*bytes*`, [`*flags*`])`

Посылает данные *bytes* через сокет, который должен быть уже подключен к удаленной конечной точке. Возвращает количество отправленных байтов, которое должно проверяться: могут быть отправлены не все данные, и в этом случае на передачу оставшихся данных должен быть направлен отдельный запрос.

sendall `sendall(`*bytes*`, [`*flags*`])`

Посылает все данные *bytes* через сокет, который должен быть уже подключен к удаленной конечной точке. Установленное для сокета предельное время ожидания применяется даже в том случае, если для передачи всех данных потребуется несколько попыток.

sendto `sendto(`*bytes*`, `*address*`)` или

`sendto(`*bytes*`, `*flags*`, `*address*`)`

Посылает данные *bytes* (сокет *s* не должен быть подключен) сокету с указанным адресом.

sendmsg `sendmsg(`*buffers*`, [`*ancdata*`, [`*flags*`, [`*address*`]]])`

Посылает обычные и вспомогательные (внеполосные) данные подключенной удаленной точке. Аргументом *buffers* должна быть итерируемая последовательность байтовых объектов. Аргументом *ancdata* должна быть итерируемая последовательность кортежей (*data*, *ancdata*, *msg_flags*, *address*), представляющих вспомогательные данные, а аргумент *msg_flags* — это флаги, документированные на странице руководства Unix, содержащей описание системного вызова `send(2)`. Аргумент должен предоставляться лишь для неподключенных сокетов и определяет конечную точку, которой должны быть посланы данные.

sendfile `sendfile(`*file*`, `*offset*`=0, `*count*`=None)`

Посылает содержимое объекта *file* (который должен быть открыт в двоичном режиме) подключенной конечной точке. На тех платформах, где доступна функция `os.sendfile`, используется эта функция; в противном случае используется вызов `send`. Аргумент *offset*, если он предоставлен, определяет начальную байтовую позицию в файле, с которой начинаются передаваемые данные, а аргумент *count* задает максимальное количество передаваемых байтов. Возвращает общее количество переданных байтов.

set_inheritable `set_inheritable(flag)` (v3 only)

Определяет, наследуется ли сокет дочерними процессами, в соответствии с булевым значением *flag*.

setblocking `setblocking(flag)`

Определяет, работает ли сокет *s* в блокирующем режиме (см. раздел "Объекты сокетов"), в соответствии с булевым значением *flag*. Вызов `s.setblocking(True)` эквивалентен вызову `os.settimeout(None)`, а вызов `s.set_blocking(False)` — вызову `os.settimeout(0.0)`.

settimeout `settimeout(timeout)`

Устанавливает режим работы сокета *s* (см. раздел "Объекты сокетов") в соответствии со значением аргумента *timeout*.

shutdown `shutdown(how)`

Отключает частично или полностью функциональность соединения сокета в соответствии со значением аргумента *how*, как описано ниже.

`socket.SHUT_RD`	Запрещает дальнейшее получение данных с использованием сокета *s*.
`socket.SHUT_WR`	Запрещает дельнейшую передачу данных с использованием сокета *s*.
`socket.SHUT_RDWR`	Запрещает дальнейшую передачу и получение данных с использованием сокета *s*.

Кроме того, объект сокета *s* имеет следующие атрибуты.

family Атрибут, представляющий семейство адресов сокета *s*

type Атрибут, представляющий тип сокета *s*

Клиент, работающий без установления соединения

Рассмотрим простейшую эхо-службу. Текст в кодировке UTF-8 (предполагаемое кодирование в большинстве файлов исходного кода на языке Python) посылается на сервер, который пересылает ту же информацию обратно клиенту, от которого она была получена. В случае служб, работающих без установления соединения, вся работа клиентов сводится к отправке каждой порции данных определенной серверной конечной точке.

Клиент, код которого приведен ниже, работает одинаково хорошо в обеих версиях, v2 и v3. (Однако имеются незначительные различия в выводе в стандартный поток `stdout`. В версии v2 текстовые строки Unicode обозначаются как `u'...'`, тогда как байтовые строки не имеют никаких дополнительных обозначений. В версии v3 текстовые строки Unicode не имеют дополнительных обозначений, тогда как байтовые строки обозначаются как `b'...'`).

```
# coding: utf-8
from __future__ import print_function
import socket

UDP_IP = '127.0.0.1'
UDP_PORT = 8883
MESSAGE = u"""\
Это набор строк, каждая из которых
будет отправлена в виде отдельной
датаграммы UDP. Ни выявления,
ни коррекции ошибок здесь нет.
Поехали! Символы ££ должны пройти."""

sock = socket.socket(socket.AF_INET,    # Интернет
                     socket.SOCK_DGRAM) # UDP
server = UDP_IP, UDP_PORT
for line in MESSAGE.splitlines():
    data = line.encode('utf-8')
    sock.sendto(data, server)
    print('Sent', repr(data), 'to', server)
    response, address = sock.recvfrom(1024) # размер буфера: 1024
    print('Recv', repr(response.decode('utf-8')), 'from', address)
```

Обратите внимание на то, что в данном случае предполагается, что сервер работает только с байтовыми строками. Поэтому клиент преобразует данные в кодировке Unicode в байтовые строки и декодирует байтовые строки, полученные от сервера, обратно в текст Unicode.

Сервер, работающий без установления соединения

Сервер, используемый для этой службы, также довольно простой. Он привязывается к своей конечной точке, получает пакеты (датаграммы) в этой конечной точке и возвращает пакет клиенту, посылая каждую датаграмму с точно теми же данными. Сервер работает со всеми клиентами одинаковым образом и не нуждается в использовании параллелизма любого рода (хотя это может быть не так в случае служб, в которых на обработку запроса уходит заметное время).

Сервер, код которого приведен ниже, работает одинаково хорошо в обеих версиях, v2 и v3 (однако имеются незначительные различия в выводе в стандартный поток `stdout`, о которых была сказано в предыдущем разделе).

```
from __future__ import print_function
import socket

UDP_IP = '127.0.0.1'
UDP_PORT = 8883

sock = socket.socket(socket.AF_INET, # Интернет
```

```
                    socket.SOCK_DGRAM) # UDP
sock.bind((UDP_IP, UDP_PORT))
print('Serving at', UDP_IP, UDP_PORT)

while True:
    data, addr = sock.recvfrom(1024) # размер буфера: 1024
    print('Recv', repr(data), 'from', addr)
    sock.sendto(data, addr)
    print('Sent', repr(data), 'to', addr)
```

Этот код не предоставляет никакой другой возможности остановить службу, кроме как прервать ее (обычно с помощью клавиатуры посредством нажатия комбинации клавиш <Ctrl+C> или <Ctrl+Break>).

Клиент, ориентированный на установление соединения

Рассмотрим упрощенный протокол наподобие эхо-протокола, ориентированный на установление соединения: сервер позволяет клиентам подключиться к своему сокету, ожидающему получения данных, получает от них произвольные байты и посылает обратно каждому клиенту те же байты до тех пор, пока клиент не закроет соединение. Ниже приведен пример простейшего тестового клиента (код которого может выполняться как в версии v2, так и в версии v3)[1].

```
# coding: utf-8
from __future__ import print_function
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect(('localhost', 8881))
    print('Connected to server')
    data = u"""Несколько строк текста,
включающие символы не из таблицы ASCII (€£),
для проверки работы как сервера,
так и клиента."""
    for line in data.splitlines():
        sock.sendall(line.encode('utf-8'))
        print('Sent:', line)
        response = sock.recv(1024)
        print('Recv:', response.decode('utf-8'))
print('Disconnected from server')
```

Заметьте, что данными является текст, поэтому его необходимо преобразовать в подходящее представление, в качестве которого мы выбрали то, которое чаще всего

[1] Обратите внимание на то, что код клиента в этом примере не является безопасным; о том, как сделать код безопасным, рассказано в разделе "Протокол защиты транспортного уровня (TLS, SSL)".

и ожидается: UTF-8. Сервер оперирует байтами (поскольку именно байты, также называемые октетами, передаются по сети). Полученный байтовый объект декодируется обратно в текст Unicode с использованием кодировки UTF-8 для вывода на экран. Можно было бы использовать любую другую подходящую кодировку: важно лишь то, чтобы текст преобразовывался перед отправкой и декодировался после получения. Серверу, который работает с байтами, вовсе не нужно знать, какая кодировка используется, разве что это может понадобиться для занесения соответствующих записей в журнал.

Сервер, ориентированный на установление соединения

Ниже приведен пример простейшего сервера, предназначенный для тестирования клиента из раздела "Клиент, ориентированный на установление соединения". Сервер использует многопоточную функциональность, предлагаемую модулем `concurrent.futures`, который рассматривался в разделе "Модуль `concurrent.futures`" главы 14, и может работать как в версии v3, так и в версии v2 (при условии, что установлена ретроподдержка модуля `concurrent`, — см. раздел "Модуль `concurrent.futures`" в главе 14).

```
from __future__ import print_function
from concurrent import futures as cf
import socket

def handle(new_sock, address):
    print('Connected from', address)
    while True:
        received = new_sock.recv(1024)
        if not received: break
        s = received.decode('utf-8', errors='replace')
        print('Recv:', s)
        new_sock.sendall(received)
        print('Echo:', s)
    new_sock.close()
    print('Disconnected from', address)

servsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
servsock.bind(('localhost', 8881))
servsock.listen(5)
print('Serving at', servsock.getsockname())

with cf.ThreadPoolExecutor(20) as e:
    try:
        while True:
            new_sock, address = servsock.accept()
            e.submit(handle, new_sock, address)
    except KeyboardInterrupt:
```

```
            pass
    finally:
        servsock.close()
```

Возможности данного сервера ограничены. В частности, он запускает не более 20 потоков, а потому может обслуживать не более 20 клиентов одновременно. Любой дополнительный клиент, пытающийся подключиться к серверу в то время, когда обслуживаются 20 клиентов, вынужден дожидаться соединения в очереди `servsock`, а если эта очередь уже содержит 5 клиентов, то любые попытки дополнительных клиентов подключиться к серверу сразу же отвергаются. Данный сервер приведен лишь в качестве элементарного примера в демонстрационных целях, а не как полнофункциональная, масштабируемая и безопасная система.

Протокол защиты транспортного уровня (TLS, SSL)

Transport Layer Security (TLS) — это *протокол защиты транспортного уровня*, также известный как Secure Sockets Layer (SSL) — *уровень защищенных сокетов*, хотя последнее название в действительности относится к протоколу-предшественнику. Протокол TLS, работающий поверх стека протоколов TCP/IP, обеспечивает конфиденциальность и целостность данных при их передаче по сети, содействуя в организации защиты от попыток несанкционированного доступа к серверу, а также прослушивания и злонамеренного изменения трафика. В качестве введения в TLS рекомендуем прочитать обстоятельную статью на эту тему в Википедии (`https://ru.wikipedia.org/wiki/TLS`).

В Python доступ к средствам для работы с протоколом TLS обеспечивается модулем `ssl` стандартной библиотеки. Эффективное использование возможностей модуля `ssl` потребует от вас не только тщательного изучения его обширной онлайн-документации (`https://docs.python.org/3/library/ssl.html`), но и глубокого понимания самих принципов функционирования TLS (вышеупомянутая статья в Википедии, какой бы содержательной она ни была, является всего лишь введением в эту обширную и сложную тему). В частности, особое внимание следует обратить на раздел документации, посвященный вопросам безопасности: вы должны изучить полностью не только этот раздел, но и материал, содержащийся в многочисленных ссылках, которые в нем приведены.

Если после прочтения предыдущего абзаца у вас сложилось впечатление, что реализация идеальной защиты данных, передаваемых по сети, представляет собой чрезвычайно трудную задачу, то это потому, что *так оно и есть* на самом деле. Организуя защиту данных, вы должны напрягать весь свой интеллект и использовать все свое мастерство для того, чтобы противостоять изощренным атакам злоумышленников, которые могут гораздо лучше разбираться во всех тонкостях этой проблемы, чем вы, поскольку они специализируются на обходе и взломе защиты, тогда как вы (как правило) не можете концентрироваться исключительно на защите данных. Ваша основная задача — создать код, предоставляющий клиентам полезные услуги, и в том, что вы как бы вынуждены

отодвигать задачи обеспечения безопасности на второй план, кроются большие риски — ведь для того, чтобы выйти победителем в этой битве интеллектов и навыков, обеспечение необходимой защиты должно быть вашей первоочередной задачей.

С учетом вышесказанного настоятельно рекомендуем достаточно основательно изучить протокол TLS всем читателям — чем лучше все разработчики будут разбираться в вопросах безопасности, тем лучше будет для всех нас (разве что за исключением хакеров, направляющих свои усилия на взлом защиты).

Если только вы уже не достигли действительно глубокого понимания того, как работают TLS и модуль `ssl` в Python (и если это так, то вам точно известно, что именно следует делать, — возможно, даже лучше, чем нам!), рекомендуем использовать экземпляр класса `SSLContext`, который позаботится обо всех деталях применения TLS. Создайте этот экземпляр с помощью функции `ssl.create_default_context`, добавьте в случае необходимости свой сертификат (это потребуется, если вы пишете собственный сервер безопасности), а затем используйте метод `wrap_socket` экземпляра для обертывания (почти[2]) каждого созданного вами экземпляра `socket.socket` экземпляром `ssl.SSLSocket`, который ведет себя в сущности так же, как и обертываемый им объект сокета, но практически прозрачно добавляет попутную проверку параметров безопасности.

Используемые по умолчанию контексты TLS обеспечивают неплохой компромисс между требованиями безопасности и широкой применимости, и мы рекомендуем вам придерживаться их (разве что ваших знаний достаточно для выполнения тонкой настройки и усиления мер безопасности в особых случаях). Если вам нужно обеспечить поддержку устаревшего программного обеспечения, неспособного использовать наиболее современные и наиболее безопасные реализации TLS, то у вас может возникнут соблазн изучить документацию лишь в том объеме, который позволит вам ослабить реализацию требований, касающихся безопасности. Если вы так решите, то делайте это на свой собственный страх и риск, — мы категорически *не* рекомендуем использовать такой подход, поскольку тем самым вы вступите на территорию, где "водятся драконы"!

В следующих разделах мы рассмотрим лишь минимальное подмножество функциональности модуля `ssl`, которого вам будет вполне достаточно, если вы хотите всего навсего следовать нашим рекомендациям. Однако помните, что даже в этом случае мы настоятельно рекомендуем вам читать документацию TLS и `ssl` хотя бы для того, чтобы приобрести базовые знания в этой сложной области. В один прекрасный день они вам очень пригодятся!

Класс `SSLContext`

Модуль `ssl` предоставляет класс `ssl.SSLContext`, экземпляры которого хранят информацию о конфигурации TLS (включая сертификаты и частные ключи)

[2] Мы говорим "почти", поскольку, когда речь идет о коде сервера, вы не должны обертывать сокет, который связываете и используете для прослушивания и от которого принимаете соединения.

и предлагают множество методов для настройки, изменения, проверки и использования этой информации. Если вы точно знаете, что нужно делать, то можете вручную инстанциализировать, настроить и использовать собственные экземпляры SSLContext для своих специальных целей.

Однако вместо этого мы рекомендуем инстанциализировать экземпляры SSLContext, используя функцию ssl.create_default_context с единственным аргументом ssl.Purpose.CLIENT_AUTH в коде сервера (когда может потребоваться аутентификация клиентов) или функцию ssl.Purpose.SERVER_AUTH в коде клиента (которому определенно потребуется аутентификация серверов). Если ваш код предназначен для использования как в качестве клиента по отношению к некоторым серверам, так и в качестве сервера по отношению к другим клиентам (как, например, в случае некоторых прокси-объектов Интернета), то вам нужно будет создать два экземпляра SSLContext, по одному для каждой из указанных целей.

В большинстве случаев полученный вами экземпляр SSLContext будет готов к работе на клиентской стороне. Если вы пишете код сервера или клиента для одного из тех редко встречающихся серверов, которые требуют TLS-аутентификации клиентов, то вам нужно иметь файл сертификата и файл ключа. Они должны быть добавлены в экземпляр SSLContextinstance (чтобы вторая сторона могла идентифицировать вас) с помощью, например, следующего кода:

```
ctx = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ctx.load_cert_chain(certfile='mycert.pem', keyfile='mykey.key')
```

Вы передаете пути к файлам сертификата и ключа методу load_cert_chain. (Информацию о том, как получить файлы ключа и сертификата, вы найдете в онлайн-документации; https://docs.python.org/3/library/ssl.html#certificates.)

Если вы пишете код клиента, то после создания экземпляра контекста ctx вам достаточно вызвать метод ctx.wrap_socket для обертывания любого сокета, который вы собираетесь подключить к серверу, и использовать обернутый результат (экземпляр ssl.SSLSocket) вместо сокета.

```
sock = socket.socket(socket.AF_INET)
sock = ctx.wrap_socket(sock, server_hostname='www.example.com')
sock.connect(('www.example.com', 443))
# далее достаточно использовать объект 'sock' обычным образом
```

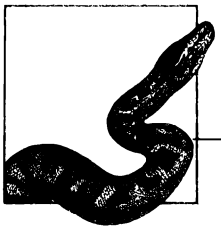
Обратите внимание на то, что в случае клиента вы должны передать методу wrap_socket аргумент server_hostname, соответствующий серверу, к которому собираетесь подключиться. Это позволяет удостовериться в корректности идентификации сервера — крайне необходимая мера в любой системе, обеспечивающей безопасность работы в Интернете.

Что касается кода серверной стороны, то сокет, связываемый с адресом или же ожидающий либо принимающий соединение, не нуждается в обертывании. Вы должны лишь связать новый сокет с результатом, возвращаемым методом accept.

```
sock = socket.socket(socket.AF_INET)
sock.bind(('www.example.com', 443))
sock.listen(5)
while True:
    newsock, fromaddr = sock.accept()
    newsock = ctx.wrap_socket(newsock, server_side=True)
    # далее используйте 'newsock' обычным образом, а затем
    # остановите и закройте его, когда завершите обмен данными
```

В данном случае вы должны передать методу `wrap_socket` аргумент `server_side=True`, уведомляя его о том, что вы работаете на стороне сервера.

Опять-таки, чтобы лучше разобраться даже в этом простом подмножестве операций модуля `ssl`, мы рекомендуем тщательно изучить онлайн-документацию.



18

Асинхронные архитектуры

Для кода, интенсивно использующего операции ввода-вывода (т.е. такого, который большую часть времени простаивает в ожидании завершения ввода-вывода) и допускающего выполнение этих операций в *неблокирующем* режиме, наилучшим с точки зрения производительности решением являются *асинхронные* архитектуры. Это означает, что ваш код может инициировать операцию ввода-вывода и, пока она выполняется, продолжить выполнение других операций, в то же время имея возможность определить момент завершения инициированной им операции ввода-вывода.

Асинхронные архитектуры иногда называют архитектурами, *управляемыми событиями*, поскольку завершение операций ввода-вывода (в канале ввода становятся доступными новые данные, или канал вывода готов к принятию новых данных) можно моделировать как внешние "события", на которые ваш код реагирует соответствующим образом.

Асинхронные архитектуры можно разбить на следующие три широкие категории.

- В архитектуре, иногда называемой *мультиплексной*, код отслеживает каналы ввода-вывода, которые в данный момент заняты выполнением операций. Если для кода нет другой работы, пока не завершится одна или несколько текущих операций ввода-вывода, он переходит в режим блокирующего ожидания (в этом случае обычно говорят, что "код блокируется"), дожидаясь завершения операций в соответствующем наборе каналов. Если завершение такой операции пробуждает ожидающий поток, ваш код обрабатывает специфические детали этого завершения (здесь слово "обработка" может также означать инициирование дополнительных операций ввода-вывода), а затем, как правило, возвращается в режим блокирующего ожидания. Python предлагает несколько низкоуровневых модулей, поддерживающих мультиплексированные асинхронные архитектуры, но для этого лучше всего использовать высокоуровневый модуль `selectors`, который обсуждается в разделе "Модуль `selectors`".
- В *архитектуре на основе обратных вызовов* код связывает с каждым ожидаемым событием *объект обратного вызова* — функцию или другой вызываемый

объект, который вызывается асинхронным фреймворком при наступлении события. Это связывание может осуществляться как явно (передача функции, которая должна вызываться, когда наступает определенное событие), так и неявно различными способами (например, за счет расширения базового класса и переопределения соответствующих методов). Предоставляемая стандартной библиотекой Python непосредственная поддержка асинхронных архитектур на основе обратного вызова не является ни достаточно сильной, ни достаточно современной, в связи с чем мы не рекомендуем ее применять и не рассматриваем в данной книге. Если вы все же захотите использовать архитектуру этого типа, рекомендуем воспользоваться сторонним фреймворком Twisted (`https://twistedmatrix.com/trac/`) или более специализированным для целей веб-разработки фреймворком Tornado (`http://www.tornadoweb.org/en/stable/`), поддерживающим сопрограммы, о чем пойдет речь в следующем разделе.

- Третья и наиболее современная категория — *асинхронные архитектуры на основе сопрограмм*, которые рассмотрены в следующем разделе.

В этой главе, предварительно обсудив общие концепции асинхронных архитектур на основе сопрограмм, мы перейдем к рассмотрению модуля `asyncio` (доступен только в версии v3), который обеспечивает реализацию подобных архитектур и архитектур на основе обратных вызовов, а также низкоуровневого модуля `selectors` (доступен как в стандартной библиотеке версии v3, так и в виде загружаемого стороннего модуля ретроподдержки для версии v2), который обеспечивает реализацию мультиплексных архитектур.

Асинхронные архитектуры на основе сопрограмм

С мультиплексными архитектурами и архитектурами на основе обратных вызовов связана одна проблема, которая заключается в том, что код, который в случае блокирующего ввода-вывода представлял бы собой единую, в целом линейную функцию, часто фрагментируется на отдельные части, взаимосвязь которых трудно усмотреть непосредственно. Этот фактор может затруднить разработку и отладку кода.

Для преодоления указанной проблемы нам нужны функции, способные приостанавливать выполнение, передавать управление фреймворку, одновременно сохраняя свое внутреннее состояние, а затем, когда фреймворк вновь вернет управление функции, возобновлять выполнение с того же места в коде, в котором оно было приостановлено. Такие функции называют *сопрограммами*.

Функции-сопрограммы и объекты сопрограмм

Термин *сопрограмма* используется для описания двух разных, но взаимосвязанных объектов: функции, определяющей сопрограмму (например, в версии 3.5 — это функции, начинающиеся с инструкции `async def`), более точным названием которой, если необходимо

исключить неоднозначность, является *функция-сопрограмма*; и объекта, возвращаемого функцией-сопрограммой и представляющего собой некую комбинацию операций ввода-вывода и вычислительных операций, выполнение которых в конечном счете завершается. Во избежание двоякого толкования в этом случае следует использовать более точное название: *объект сопрограммы*.

Функцию-сопрограмму можно рассматривать как функцию-фабрику объектов сопрограммы. Выражаясь более определенно, следует подчеркнуть, что результатом вызова функции-сопрограммы является не выполнение кода, написанного пользователем, а возврат объекта сопрограммы. Объекты сопрограммы не являются вызываемыми объектами: скорее, вы планируете их выполнение посредством вызовов соответствующих функций фреймворка, такого как `asyncio`, или ожидаете их завершения в теле другого объекта сопрограммы с помощью ключевого слова `await`.

Стандартная библиотека версии v2 не поддерживает асинхронные архитектуры на основе сопрограмм. С другой стороны, версия v3 предлагает модуль `asyncio`, обеспечивающий достаточно мощную поддержку архитектур этой категории.

Если вы хотите создавать библиотеки, особенно для нужд веб-разработки, и использовать в них асинхронные архитектуры на основе сопрограмм, способных работать в обеих версиях, v2 и v3, обратите внимание на сторонний фреймворк Tornado (`http://www.tornadoweb.org/en/stable/`), представляющий собой асинхронный веб-сервер в связке с фреймворком веб-приложений.

Приступая к написанию приложения, ориентируйтесь на конкретную версию Python

Если вы пишете приложение, а не библиотеку, всегда ориентируйтесь на конкретную версию Python (мы настоятельно рекомендуем версию v3), чем избавите себя от каких бы то ни было хлопот, связанных с кросс-платформенностью, и будете иметь полную свободу действий для написания наилучшего кода, предназначенного для решения конкретной задачи.

Чтобы упростить и сделать более удобным написание и использование сопрограмм, в версии Python 3.5 были введены два новых ключевых слова: `async` и `await`, о которых речь пойдет в разделе "Ключевые слова `async` и `await`". Теперь вместо создания сопрограмм посредством выражения `yield from`, как это нужно было делать в версиях 3.4 и более ранних в соответствии с обсуждением, приведенным в разделах "Инструкция `yield from` (только в версии v3)" главы 3, "Модуль `asyncio` (только в версии v3)" и "Сопрограммы модуля `asyncio`", следует использовать указанные ключевые слова. Это полностью сохраняет возможность взаимодействия с сопрограммами, основанными

на использовании выражения `yield from`, но улучшает читаемость кода, позволяет избежать некоторых ошибок и способствует повышению производительности. В настоящее время сопрограммы на основе ключевых слов `async` и `await` поддерживаются также фреймворком Tornado (`http://www.tornadoweb.org/en/stable/`).

Все асинхронные архитектуры имеют общий элемент: *цикл событий*. Это код (либо неявный на уровне фреймворка, либо явный на уровне приложения), выполняющий бесконечный цикл в ожидании событий (изменения состояния отслеживаемых каналов ввода-вывода или — в зависимости от природы приложения — проявления других внешних факторов, таких как взаимодействие пользователя с пользовательским интерфейсом). Объекты сопрограмм могут выполняться лишь в том случае, если выполняется цикл событий.

Наконец, все виды асинхронных архитектур могут использоваться для поддержки так называемых *фьючерсов* (другие названия — *promise-объекты, обещания* и *отложенные объекты*), которые аналогичны объектам, описанным в табл. 14.1 (хотя последние работают с потоками и процессами, тогда как асинхронные фьючерсы отражают завершение некоторых неблокирующих операций ввода-вывода). Фьючерсы, обертывающие объекты сопрограмм, могут выполняться, только если выполняется цикл событий.

Модуль `asyncio` (только в версии v3)

Модуль `asyncio` образует основу современного подхода к асинхронному программированию, используемого в версии v3, особенно в сетевых задачах. Он входит в состав стандартной библиотеки, и его основной задачей является предоставление инфраструктуры и компонентов на относительно низком уровне абстракции. Однако на вики-странице по адресу `https://github.com/python/asyncio/wiki/ThirdParty` вы найдете описание высокоуровневых модулей — таких строительных блоков, как `aiohttp` (поддержка HTTP-серверов и клиентов, включая протокол WebSocket; `https://ru.wikipedia.org/wiki/WebSocket`), веб-фреймворки (одни из них работают поверх модуля `aiohttp`, другие являются автономными), AsyncSSH (поддержка SSH-серверов и клиентов) и др.

Внесите свой вклад в экосистему `asyncio`

Если когда-нибудь вам понадобится написать код высокоуровневого модуля для задачи или протокола, которые еще не охвачены модулями, вошедшими в список на упомянутой вики-странице, и этот модуль может представлять всеобщий интерес, настоятельно рекомендуем вам тщательно протестировать его, подготовить качественную документацию (это все в любом случае вам пригодится!), сформировать пакет и выгрузить его на сайт PyPI, как описано в главе 25, а затем отредактировать вики-страницу для добавления указателя на него. В этом случае вас будет ждать не только слава автора открытого программного кода, но и помощь со стороны других программистов в улучшении вашего модуля!

В процессе написания кода на языке Python, основанного на модуле `asyncio` (в идеальном случае с привлечением плагинов с сайта `asyncio.org`), вам не обойтись без написания функций-сопрограмм. До версии Python 3.4 включительно такие функции являлись генераторами, использующими инструкцию `yield from` (обсуждалась в разделе "Инструкция `yield from` (только в версии v3)" главы 3), которая декорирована декоратором `@asyncio.coroutine` (обсуждается в разделе "Сопрограммы модуля `asyncio`"). Несмотря на то что в версиях Python 3.5 и выше подход на основе инструкции `yield from` (плюс декоратор) по-прежнему является допустимым в интересах обратной совместимости с предыдущими выпусками, вместо него лучше использовать новую инструкцию `async def`, которая обсуждается в разделе "Ключевые слова `async` и `await`". В оставшейся части главы, говоря о сопрограммах, мы будем подразумевать те функции или методы, предоставляемые модулем `asyncio`, которые действительно являются функциями сопрограмм и, таким образом, не блокируются и возвращают объекты сопрограмм (внутренне они используют инструкции `async def` в версии 3.5 и выше, а для обеспечения обратной совместимости в версии 3.4 и более ранних — декоратор `@asyncio.coroutine` и инструкцию `yield from` в теле функции).

Работа с сопрограммами с использованием модуля `asyncio`

Чтобы обеспечить обратную совместимость с версией Python 3.4, в версии v3 (под которой, как и везде в этой книге, мы подразумеваем версию Python 3.5) предоставляется возможность реализации сопрограмм любым из двух способов: в виде соответствующим образом декорированных генераторов, используя инструкцию `yield from`, или в виде собственно сопрограмм, используя новые ключевые слова `async` и `await`.

Сопрограммы модуля `asyncio`

Как уже отмечалось в разделе "Асинхронные архитектуры на основе сопрограмм", основное понятие, используемое в модуле `asyncio`, — это *сопрограмма*, т.е. функция, способная приостанавливать выполнение, сохраняя свое внутреннее состояние, явно возвращать управление фреймворку, организующему ее работу (в случае модуля `asyncio` — циклу событий, рассмотренному в разделе "Цикл событий модуля `asyncio`"), и возобновлять выполнение, когда фреймворк сообщит об этом, с того места, в котором оно было приостановлено.

Подобно функции-генератору, которая немедленно выполняется, не запуская на выполнение пользовательский код, и возвращает объект генератора, функция-сопрограмма немедленно выполняется, также не запуская на выполнение никакой пользовательский код, и возвращает объект сопрограммы. В действительности вплоть до версии Python 3.4 функция-сопрограмма была всего лишь особой разновидностью функции-генератора. Последняя — это любая функция, содержащая одно или несколько вхождений ключевого слова `yield`, и функция-сопрограмма была частным случаем такой функции, содержащей одно или несколько вхождений выражения `yield from` *`iterable`*`...`, хотя подобное выражение могло встречаться и в

функции-генераторе, не предназначенной для использования в качестве функции-сопрограммы, о чем шла речь в разделе "Инструкция `yield from` (только в версии v3)" главы 3 (результирующая неоднозначность, хотя и незначительная, послужила причиной введения в версии Python 3.5 ключевых слов `async` и `await`, устранивших любую неоднозначность). Чтобы явно обозначить подобную сопрограмму на основе выражения `yield from` как *функцию-сопрограмму* и гарантировать возможность ее использования в модуле `asyncio`, ее приходилось декорировать с помощью декоратора `@asyncio.coroutine`. В качестве примера ниже представлена сопрограмма, которая выжидает в течение `delay` секунд, а затем возвращает результат.

```
@asyncio.coroutine
def delayed_result(delay, result):
  yield from asyncio.sleep(delay)
  return result
```

Например, выполнив вызов `delayed_result(1.5, 23)`, вы получите объект генератора (а именно, объект сопрограммы), который может ожидать в течение полутора секунд, а затем вернуть число `23`. Обратите внимание на то, что рассматриваемый объект генератора не выполнится до тех пор, пока он не будет надлежащим образом подключен к циклу событий (раздел "Цикл событий модуля `asyncio`"), а цикл событий не будет запущен. Например, следующий код устанавливает для *x* значение `23` после паузы длительностью полторы секунды.

```
loop = asyncio.get_event_loop()
x = loop.run_until_complete(delayed_result(1.5, 23))
```

Если вам нужно проверить, является ли объект *f* функцией-сопрограммой (это может потребоваться главным образом в целях отладки), воспользуйтесь вызовом `asyncio.iscoroutinefunction(f)`. Чтобы проверить, является ли объект *x* объектом сопрограммы, используйте вызов `asyncio.iscoroutine(x)`.

Ключевые слова `async` и `await`

С целью повышения понятности и удобочитаемости кода в версии Python 3.5 были введены ключевые слова `async` и `await`[1]. Используйте `async` для создания сопрограмм (с помощью инструкции `async def`) и асинхронных менеджеров контекста (с помощью инструкции `async with`), а также для асинхронного итерирования (с помощью инструкции `async for`).

Если вы используете инструкцию `async def` вместо инструкции `def` и ключевое слово `await` вместо выражения `yield from`[2], то тем самым явно определите функци-

[1] В настоящее время это псевдоключевые слова: в интересах обратной совместимости вы по-прежнему можете использовать `async` и `await` в качестве обычных идентификаторов, но поступать так не следует, поскольку это ухудшит читаемость кода, а в одной из будущих версий станет считаться синтаксической ошибкой.

[2] Ключевое слово `await` нельзя использовать для ожидания завершения (исчерпания) произвольного итерируемого объекта — в качестве такового допускается лишь использование объекта сопрограммы или экземпляра `asyncio.Future`.

ю-сопрограмму, не требующую декорирования (и уже не являющуюся функцией-генератором). Например, выполнив следующий код, вы получите функцию-сопрограмму new_way, полностью эквивалентную функции delayed_result из предыдущего примера.

```
async def new_way(delay, result):
await asyncio.sleep(delay)
return result
```

(Вы могли бы декорировать функцию new_way с помощью декоратора @asyncio.coroutine, но поступать так не только не требуется, но и не рекомендуется.) Этот же способ можно использовать для написания функции, вообще не ожидающей наступления каких-либо событий с помощью ключевого слова await. (Несмотря на всю специфичность таких функций, они могут оказаться полезными в процессе прототипирования, тестирования и рефакторинга кода.) Функция-сопрограмма, определенная с помощью инструкции async def, не может содержать выражение yield, в то время как ключевое слово await можно использовать только в такой функции-сопрограмме.

Также только в истинных функциях-сопрограммах (т.е. таких, которые определены с помощью инструкции async def) можно использовать две новые полезные конструкции: async with и async for.

async with

Инструкция async with — точный аналог инструкции with, за исключением того, что менеджер контекста (объект, являющийся результатом вычисления выражения, стоящего сразу же после ключевого слова with) должен быть *асинхронным* контекстным менеджером, реализующим специальные методы __aenter__ и __aexit__, соответствующие методам __enter__ и __exit__ обычного контекстного менеджера (класс может предусмотрительно реализовать все четыре специальных метода, чтобы его экземпляры можно было использовать как с простыми инструкциями with, так и с инструкциями async with). Методы __aenter__ и __aexit__ должны возвращать объекты, позволяющие использовать ключевое слово await (в типичных случаях каждый из них является функцией-сопрограммой, определенной с помощью инструкции async def и, таким образом, возвращающей объект сопрограммы, который, разумеется, допускает использование ключевого слова await). Инструкция async with x: неявно использует вызов await *x*.__aenter__() при входе и вызов await *x*.__aexit__(*type*, *value*, *tb*) при выходе, тем самым элегантно обеспечивая гладкое поведение асинхронного менеджера контекста.

async for

Инструкция async for — точный аналог инструкции for, за исключением того, что итерируемый объект (объект, являющийся результатом вычисления выражения, стоящего сразу же после ключевого слова for) должен быть *асинхронным* итерируемым объектом, реализующим специальный метод __aiter__[3], соответствующий мето-

[3] Обратите внимание на то, что метод __aiter__ не обязан быть сопрограммой, в то время как остальные методы, имена которых начинаются с символов __a, должны быть сопрограммами.

ду `__iter__` простого итерируемого объекта (класс может предусмотрительно реализовать оба специальных метода, чтобы его экземпляры можно было использовать как с простыми инструкциями `for`, так и с инструкциями `async for`). Метод `__aiter__` должен возвращать асинхронный итератор (реализующий специальный метод `__anext__`, который соответствует методу `__next__` простого итератора, но возвращающий объект, который позволяет использовать ключевое слово `await`). Инструкция `async for x:` неявно использует вызов *x*`.__aiter__()` при входе и вызов `await` *y*`.__anext__()` для возвращаемого в конечном счете асинхронного итератора *y*, тем самым элегантно обеспечивая гладкое поведение асинхронного цикла.

Цикл событий модуля `asyncio`

Модуль `asyncio` предоставляет явный интерфейс *цикла событий*, и некоторые базовые реализации этого интерфейса, в настоящее время являющегося самым крупным компонентом данного модуля.

Интерфейс цикла событий обладает широкими функциональными возможностями и предоставляет несколько категорий методов. Сам интерфейс встроен в класс `asyncio.BaseEventLoop`. Модуль `asyncio` предлагает ряд альтернативных реализаций цикла событий, в зависимости от платформы, а сторонние пакеты позволяют расширить его возможности, например для интеграции со специализированными фреймворками пользовательских интерфейсов, такими как Qt. Основная идея заключается в том, чтобы все реализации цикла событий реализовывали один и тот же интерфейс.

Теоретически ваше приложение может иметь несколько циклов событий, управляемых несколькими явно определенными *политиками цикла событий*. На практике необходимость в таком усложнении встречается редко, и вам вполне хватит одного цикла событий, обычно выполняющегося в основном потоке, управляемом с использованием неявно определенной глобальной политики. Основное исключение могут составить ситуации, в которых вы захотите, чтобы асинхронный код выполнялся в нескольких потоках программы. В этом случае, хотя вы по-прежнему сможете обойтись одной политикой, вам потребуются несколько циклов событий — по одному на каждый поток, который вызывает методы цикла событий, отличные от метода `call_soon_threadsafe`, описанного в табл. 18.1. Каждый поток может вызывать только методы цикла, инстанциализированного в данном потоке[4].

В оставшейся части этой главы предполагается (если явно не оговорено иное), что вы используете единственный цикл событий, обычно заданный по умолчанию, который можно получить посредством вызова *loop*`= asyncio.get_event_loop()` где-то в начале кода. Иногда вам может потребоваться принудительно задать использование конкретной реализации, для чего нужно будет сначала явно инстанциализировать

[4] Для получения подробной информации относительно особенностей использования цикла событий в Windows и его обратной совместимости со старыми выпусками macOS обратитесь к онлайн-документации. На других платформах отсутствуют какие-либо особенности использования циклов событий.

цикл в соответствии с требованиями конкретной платформы или стороннего фреймворка, а затем сразу же вызвать функцию `asyncio.set_event_loop(loop)`. За исключением особенностей семантики конкретной платформы или фреймворка, которые в данной книге не рассматриваются, материал оставшейся части главы применим и к указанному второму, хотя и довольно редко встречающемуся случаю применения циклов событий.

В следующих разделах рассматриваются девять категорий методов, предоставляемых экземпляром *loop* класса `asyncio.BaseEventLoop`, и несколько тесно связанных с ними функций и классов, предоставляемых модулем `asyncio`.

Состояние и режим отладки цикла событий

Объект *loop* может находиться в одном из трех состояний: *остановлен* (это состояние только что созданного объекта *loop*, пока цикл не выполняет никаких действий), *выполняется* (выполняется вся функциональность цикла) или *закрыт* (выполнение цикла необратимо завершено, и его повторный запуск невозможен). Независимо от своего состояния объект *loop* может находиться в режиме *отладки* (проверка работоспособности кода с выводом подробной информации, которая может быть полезной при разработке кода) или не находиться (более быстрое выполнение операций с меньшим количеством сообщений; это нормальный режим при "производственном" использовании объекта *loop*, в отличие от разработки, отладки и тестирования кода), о чем пойдет речь в разделе "Разработка и отладка кода на основе модуля `asyncio`". Объект *loop* предоставляет следующие методы, связанные с состоянием и режимом работы цикла.

`close` `close()`

Переводит объект *loop* в закрытое состояние, отменяет выполнение текущих обратных вызовов, сбрасывает очереди и запрашивает у объекта, ответственного за выполнение цикла, прекращение выполнения. Метод *loop*`.close()` можно вызвать только в том случае, если вызов *loop*`.is_running()` возвращает значение `False`. После вызова метода *loop*`.close()` нельзя вызывать для объекта *loop* другие методы, кроме методов *loop*`.is_closed()` или *loop*`.is_closing()` (в этом случае оба они возвращают значение `True`) и *loop*`.close()` (который в этом случае не выполняет никаких действий).

`get_debug` `get_debug()`

Возвращает значение `True`, если объект *loop* находится в режиме отладки; в противном случае возвращает значение `False`. Начальным значением является `True`, если переменная среды `PYTHONASYNCIODEBUG` не является пустой строкой; в противном случае оно равно `False`.

`is_closed` `is_closed()`

Возвращает значение `True`, если объект *loop* закрыт; в противном случае возвращает значение `False`.

is_closing `is_closing()`

Возвращает значение `True`, если объект *`loop`* закрывается или уже закрыт; в противном случае возвращает значение `False`.

is_running `is_running()`

Возвращает значение `True`, если объект *`loop`* выполняется; в противном случае возвращает значение `False`.

run_forever `run_forever()`

Выполняет объект *`loop`* до тех пор, пока не будет вызван метод *`loop`*`.stop()`. В конечном счете возвращает управление после того, как метод `stop` поместит объект *`loop`* в остановленное состояние.

run_until_complete `run_until_complete(`*`future`*`)`

Выполняется до тех пор, пока не завершится объект *`future`* (экземпляр класса `asyncio.Future`, который рассмотрен в разделе "Фьючерсы"). Если объект *`future`* является объектом сопрограммы или другим объектом, позволяющим использовать ключевое слово `await`, он обертывается вызовом `asyncio.ensure_future`, рассмотренным в разделе "Задачи". По завершении объекта *`future`* метод `run_until_complete` возвращает его результат или возбуждает исключение.

set_debug `set_debug(`*`debug`*`)`

Устанавливает для объекта *`loop`* отладочный режим в соответствии со значением *`bool`*`(debug)`.

stop `stop()`

Если метод `stop` вызывается, когда объект *`loop`* остановлен, он однократно опрашивает селектор ввода-вывода, используя значение *`timeout`*, равное нулю, после чего выполняет все запланированные обратные вызовы (как предыдущие, так и выполняющиеся в ответ на события ввода-вывода, которые произошли во время выполнения данного одноразового опроса селектора ввода-вывода), а затем осуществляет выход, оставляя объект *`loop`* в остановленном состоянии.

Если метод `stop` вызывается во время выполнения объекта *`loop`* (например, в результате вызова метода `run_forever`), то он выполняет все текущие запланированные обратные вызовы, а затем осуществляет выход, оставляя объект *`loop`* в остановленном состоянии (в этом случае обратные вызовы, вновь запланированные другими обратными вызовами, не выполняются; они остаются в очереди и будут выполнены после повторного вызова метода `run_forever`). После этого метод `run_forever`, который перевел объект *`loop`* в состояние выполнения, возвращает управление вызвавшему его коду.

Разработка и отладка кода на основе модуля asyncio

В процессе разработки кода, использующего модуль `asyncio`, большую помощь может оказать проверка работоспособности кода с записью соответствующих данных в журнал.

В дополнение к вызову `loop.set_debug(True)` (или установке непустой строки в качестве значения переменной окружения `PYTHONASYNCIODEBUG`) установите для уровня протоколирования значение `DEBUG`, выполнив, например, в начале программы вызов `logging.basicConfig(level=logging.DEBUG)`.

Если в режиме отладки обнаружится, что некоторые объекты `transport` или объекты цикла событий не были закрыты явным образом, вы получите весьма полезные предупреждения `ResourceWarning` во время запуска Python (см. табл. 2.1), что часто является свидетельством наличия логических ошибок в коде: включите вывод предупреждений в соответствии с рекомендациями, приведенными в разделе "Модуль `warnings`" главы 16, — например, используйте параметр командной строки `-Wdefault` (без пробела между буквой `W` и словом `default`).

За более подробными советами и рекомендациями, касающимися разработки и отладки кода, использующего модуль `asyncio`, обратитесь к соответствующему разделу онлайн-документации (`https://docs.python.org/3/library/asyncio-dev.html`).

Методы `call`, `time` и `sleep`

Основным функциональным назначением цикла событий модуля `asyncio` является планирование вызовов функций (табл. 18.1) для их выполнения "при первом удобном случае" или с указанным временем задержки. Для достижения последней из этих целей объект *`loop`* поддерживает собственные внутренние часы (ведущие отсчет в секундах и долях секунды), показания которых не всегда совпадают с показаниями системных часов (о них шла речь в главе 12).

Используйте функцию `functools.partial` для передачи именованных аргументов в вызовах

Методы объекта *`loop`*, предназначенные для планирования вызовов, не поддерживают непосредственно передачу именованных аргументов. Если вам необходимо передать именованные аргументы, оберните вызываемую функцию вызовом функции `functools.partial`, описанной в табл. 7.4. Это наилучший способ достижения данной цели (более предпочтительный, чем лямбда-выражения или замыкания), поскольку отладчики (включая отладчик модуля `asyncio`) выполняют интроспекцию подобных обернутых функций, предоставляя дополнительную информацию в более понятной форме, чем в случае альтернативных подходов.

Таблица 18.1. Методы цикла событий

Метод	Описание
`call_at`	`call_at(`*`when`*`, `*`callback`*`, *`args`*`)` Планирует выполнение обратного вызова `callback(*`*`args`*`)` на момент времени, когда значение `loop.time()` совпадет со значением аргумента (или при первой возможности после этого, если точно в то же время будет

Метод	Описание
	выполняться другой объект). Возвращает экземпляр `asyncio.Handle`, для которого можно вызвать метод `cancel()`, чтобы отменить обратный вызов (не сопровождается никакими последствиями, если это уже произошло)
`call_later`	`call_later(`*`delay`*`, `*`callback`*`, *`args`*`)` Планирует выполнение обратного вызова `callback(*args)` через *`delay`* секунд после текущего момента времени (значение *`delay`* может включать дробную часть) или при первой возможности после этого, если точно в то же время будет выполняться другой объект. Возвращает экземпляр `asyncio.Handle`, для которого можно вызвать метод `cancel()`, чтобы отменить обратный вызов (не сопровождается никакими последствиями, если это уже произошло)
`call_soon`	`call_soon(`*`callback`*`, *`args`*`)` Планирует выполнение обратного вызова `callback(*`*`args`*`)` при первой же возможности (в порядке очередности "первым пришел, первым ушел" по отношению к другим запланированным обратным вызовам). Возвращает экземпляр `asyncio.Handle`, для которого можно вызвать метод `cancel()`, чтобы отменить обратный вызов (не сопровождается никакими последствиями, если это уже произошло)
`call_soon_threadsafe`	`call_soon_threadsafe(`*`callback`*`, *`args`*`)` Аналогичен методу `call_soon`, но безопасен в отношении вызова из потока, отличного от того, в котором выполняется объект *`loop`* (обычно объект *`loop`* должен выполняться в основном потоке, чтобы обеспечить возможность обработки сигналов и доступ из других процессов)
`time`	`time()` Возвращает внутреннее текущее время цикла в виде значения с плавающей точкой

Кроме собственных методов объекта *`loop`*, внутреннее текущее время цикла может использоваться также функцией `sleep` уровня модуля (т.е. функцией, предоставляемой непосредственно модулем `asyncio`), которая описана ниже.

`sleep` *`coroutine`* `sleep(`*`delay`*`, `*`result`*`=None)`

Функция-сопрограмма, создающая и возвращающая объект сопрограммы, который завершается через *`delay`* секунд и возвращает результат (значение аргумента *`delay`* может включать дробную часть).

Соединения и сервер

Объект *`loop`* может иметь открытые каналы связи нескольких видов: соединения, устанавливаемые с прослушиванием сокетов (потоковых, датаграммных или — на платформах Unix — Unix-сокетов) других систем; соединения, которые прослушивают входящие соединения (сгруппированные в экземпляры класса `Server`);

и соединения на основе прямых и обратных каналов связи с подпроцессами. Ниже приведены методы, которые объект *loop* предоставляет для создания соединений различных видов.

create_connection *coroutine* create_connection(*protocol_factory*, *host*=None, *port*=None, *, *ssl*=None, *family*=0, *proto*=0, *flags*=0, *sock*=None, *local_addr*=None, *server_hostname*=None)

protocol_factory — это единственный обязательный аргумент, представляющий собой вызываемый объект, который не принимает аргументов и возвращает экземпляр класса protocol. Метод create_connection — это функция-сопрограмма: его результирующий объект сопрограммы, выполняемой циклом, в случае необходимости создает соединение, обертывает его экземпляром класса transport, создает экземпляр класса protocol, связывает экземпляр protocol с экземпляром transport с помощью метода connection_made экземпляра protocol и наконец, когда все это сделано, возвращает пару (*transport*, *protocol*) (раздел "Классы transport и protocol").

Если у вас имеется уже подключенный потоковый сокет, который нужно только обернуть экземплярами transport и protocol, передайте его в качестве именованного аргумента sock, избегая передачи любого из аргументов *host*, *port*, *family*, *proto*, *flags*, *local_addr*, иначе метод create_connection создаст и подключит новый потоковый сокет (относящийся к семейству AF_INET или AF_INET6, в зависимости от хоста или семейства, если они явно указаны, и имеющий тип SOCK_STREAM). И еще вам потребуется передать некоторые или все аргументы, необходимые для указания конкретных свойств подключаемого сокета.

Необязательные аргументы, если таковые имеются, должны передаваться как именованные аргументы.

Если аргумент ssl имеет истинное значение, то это обязывает к использованию протокола защиты транспортного уровня SSL/TLS (в частности, если им является экземпляр ssl.SSLContext, рассмотренный в разделе "Класс SSLContext" главы 17, то этот экземпляр используется для создания экземпляра transport). В таком случае вы можете при желании передать аргумент *server_hostname* в качестве имени хоста, которому должен соответствовать сертификат сервера (передача пустой строки означает отключение проверки соответствия имени хоста, что создает серьезные риски для системы безопасности). Если вы укажете аргумент *host*, то можете опустить аргумент *server_hostname*. В этом случае аргумент *host* определяет имя хоста, которое должно соответствовать сертификату.

Все остальные необязательные, только именованные аргументы могут передаваться лишь в том случае, если не передается аргумент *sock*. Аргументы *family*, *proto* и *flags* передаются далее методу

getaddrinfo для разрешения аргумента *host*. Аргумент *local_addr* — это пара (*local_host*, *local_port*), передаваемая далее методу getaddrinfo для разрешения локального адреса, к которому должен быть привязан создаваемый сокет.

create_datagram_endpoint

coroutine create_datagram_endpoint(*protocol_factory*, *local_addr*=None, *remote_addr*=None, *, *family*=0, *proto*=0, *flags*=0, *reuse_address*=True, *reuse_port*=None, *allow_broadcast*=None, *sock*=None)

Этот метод во многом аналогичен методу create_connection, за исключением того, что создает не потоковое, а датаграммное соединение (т.е. сокет имеет тип SOCK_DGRAM). Необязательный аргумент *remote_addr* можно передавать в виде пары (*remote_host*, *remote_port*). Аргумент *reuse_address*, если он равен True (значение по умолчанию), означает повторное использование локального сокета, не дожидаясь истечения его естественного тайм-аута. Аргумент *reuse_port*, если он равен True, на некоторых Unix-платформах разрешает привязку нескольких датаграммных соединений к одному и тому же порту. Аргумент *allow_broadcast*, если он равен True, разрешает данному соединению широковещательную передачу датаграмм.

create_server

coroutine create_server(*protocol_factory*, *host*=None, *port*=None, *, *family*=socket.AF_UNSPEC, *flags*=socket.AI_PASSIVE, *sock*=None, *backlog*=100, *ssl*=None, *reuse_address*=None, *reuse_port*=None)

Аналогичен методу create_connection, за исключением того, что создает или обертывает *прослушивающий* сокет и возвращает экземпляр класса asyncio.Server (с атрибутом sockets, представляющим список объектов socket, прослушиваемых сервером, методом close() для асинхронного закрытия всех сокетов, а также методом-сопрограммой wait_closed() для ожидания закрытия всех сокетов).

Аргументом *host* может быть строка или последовательность строк для привязки к нескольким хостам; значению None соответствует привязка ко всем интерфейсам.

Аргумент *backlog* — максимальное количество соединений, находящихся в очереди (передаваемых методу listen базового сокета). Означает повторное использование локального сокета, не дожидаясь истечения его естественного тайм-аута. Аргумент *reuse_port*, если он равен True, на некоторых Unix-подобных платформах разрешает привязку нескольких прослушивающих соединений к одному и тому же порту.

create_unix_connection

coroutine create_unix_connection(*protocol_factory*, *path*, *, *ssl*=None, *sock*=None, *server_hostname*=None)

Аналогичен методу create_connection, за исключением того, что для соединения используется сокет Unix (семейство сокетов

AF_UNIX, тип сокетов SOCK_STREAM), заданный аргументом *path*. Сокеты Unix обеспечивают быстрый и безопасный обмен данными, но только между процессами на одном Unix-компьютере.

create_unix_server *coroutine* create_unix_server(*protocol_factory*, *path*=None, *, *sock*=None, *backlog*=100, *ssl*=None)

Аналогичен методу create_server, но предназначен для соединений, которые используют сокет Unix, заданный аргументом *path*.

Помимо сокетов циклы событий могут подключать каналы передачи данных подпроцессов, используя приведенные ниже методы.

connect_read_pipe *coroutine* connect_read_pipe(*protocol_factory*, *pipe*)

Возвращает пару (*transport*, *protocol*), обертывающую неблокирующий файловый объект *pipe*, доступный в режиме чтения.

connect_write_pipe *coroutine* connect_write_pipe(*protocol_factory*, *pipe*)

Возвращает пару (*transport*, *protocol*), обертывающую неблокирующий файловый объект *pipe*, доступный в режиме записи.

Задачи

В модуле asyncio объект задачи (класс asyncio.Task, рассмотренный в разделе "Задачи", является подклассом класса asyncio.Future, рассмотренного в разделе "Фьючерсы") обертывает объект сопрограммы и организует его выполнение. Ниже описан метод объекта *loop*, позволяющий создать задачу.

create_task create_task(*coro*)

Создает и возвращает экземпляр Future (обычно являющийся экземпляром Task), обертывающий объект сопрограммы *coro*.

Параметры функции-фабрики, используемой циклом событий для создания задач, можно настраивать, но это требуется редко, за исключением случаев, когда вы пишете пользовательские реализации цикла событий, поэтому далее этот вопрос не обсуждается.

Другим, в общих чертах эквивалентным способом создания задачи является вызов функции asyncio.ensure_future с единственным аргументом в виде объекта сопрограммы или другого объекта, способного ожидать наступления событий. В данном случае эта функция создает и возвращает экземпляр Task, обертывающий объект сопрограммы. (В случае вызова функции asyncio.ensure_future с аргументом, являющимся экземпляром Future, она возвращает аргумент в неизменном виде.)

Создавайте задачи с помощью метода `loop.create_task`

Мы рекомендуем использовать метод *loop*`.create_task` как более явный и удобочитаемый способ создания задач по сравнению с приблизительно эквивалентным ему методом `asyncio.ensure_future`.

Наблюдение за дескрипторами файлов

Вы можете использовать объект *loop* на несколько более низком уровне абстракции, чтобы наблюдать за дескрипторами файлов с целью отслеживания моментов времени, когда они становятся доступными для чтения или записи, и вызывать соответствующие функции обратного вызова при освобождении дескрипторов. Описание методов, предоставляемых объектом *loop* для наблюдения за файловыми дескрипторами, приведено ниже. (На платформах Windows в случае использования реализации `SelectorEventLoop` объекта *loop*, заданной по умолчанию, эти методы применимы лишь к файловым дескрипторам, представляющим сокеты. Если же используется альтернативная реализация `ProactorEventLoop`, которую вы можете инстанциализировать явным образом, то эти методы вообще нельзя применять.)

`add_reader` `add_reader(`*fd*`, `*callback*`, *`*args`*`)`

Если дескриптор *fd* становится доступным для чтения, выполняет вызов `callback(`*`*args`*`)`.

`add_writer` `add_writer(`*fd*`, `*callback*`, *`*args`*`)`

Если дескриптор *fd* становится доступным для записи, выполняет вызов `callback(`*`*args`*`)`.

`remove_reader` `remove_reader(`*fd*`)`

Прекращает наблюдение за дескриптором *fd* с целью определения его доступности для чтения.

`remove_writer` `remove_writer(`*fd*`)`

Прекращает наблюдение за дескриптором *fd* с целью определения его доступности для записи.

Операции с сокетами, имена хостов

Также на низком уровне абстракции объект *loop* предоставляет четыре метода-сопрограммы, соответствующие методам объектов сокета, рассмотренным в главе 17.

`sock_accept` *coroutine* `sock_accept(`*sock*`)`

Сокет *sock* должен быть прослушивающим, неблокирующим и привязанным к адресу. Метод `sock_accept` возвращает объект сопрограммы, результатом выполнения которой является пара `(`*conn*`, `*address*`)`, где *conn* — это новый объект сокета, предназначенный для отправки и получения данных с использованием данного соединения, а *address* — адрес, связанный с сокетом на другом конце соединения.

sock_connect *coroutine* sock_connect(*sock*, *address*)

Сокет *sock* должен быть неблокирующим и неподключенным. Чтобы метод sock_connect не использовал самостоятельно DNS, аргументом *address* должен быть возвращаемый результат вызова *getaddrinfo* (например, в случае сетевых сокетов аргумент *address* должен включать IP-адрес, а не имя хоста). Метод sock_connect возвращает объект сопрограммы, результатом выполнения которого является возврат значения None и подключение сокета *sock* к запрошенному соединению.

sock_recv *coroutine* sock_recv(*sock*, *nbytes*)

Сокет *sock* должен быть неблокирующим и уже подключенным к соединению. Аргумент *nbytes* — целое число, представляющее максимальное количество получаемых байтов. Метод *sock_recv* возвращает объект сопрограммы, результатом выполнения которого является возврат байтового объекта, содержащего полученные сокетом данные (или возбуждение исключения в случае ошибки в сети).

sock_sendall *coroutine* sock_sendall(*sock*, *data*)

Сокет *sock* должен быть неблокирующим и уже подключенным к соединению. Метод sock_sendall возвращает объект сопрограммы, результатом выполнения которого является возврат значения *None* и отправка через сокет всех данных, содержащихся в аргументе *data*. (В случае ошибки в сети возбуждается исключение. Способов определения количества байтов, отправленных сокету на втором конце соединения до того, как в сети возникла ошибка, не существует.)

Часто возникает необходимость в выполнении DNS-поиска. Объект *loop* предоставляет два метода-сопрограммы, которые работают аналогично одноименным функциям, описанным в табл. 17.1, но асинхронным, неблокирующим способом.

getaddrinfo *coroutine* getaddrinfo(*host*, *port*, *, *family*=0, *type*=0, *proto*=0, *flags*=0)

Возвращает объект сопрограммы, результатом выполнения которого является возврат кортежа (*family*, *type*, *proto*, *canonname*, *sockaddr*), аналогичного тому, который возвращается вызовом socket.getaddrinfo.

getnameinfo *coroutine* getnameinfo(*sockaddr*, *flags*=0)

Возвращает объект сопрограммы, результатом выполнения которого является возврат пары (*host*, *port*), аналогичной тому, которая возвращается вызовом socket.getnameinfo.

Сигналы Unix

На Unix-подобных платформах объект *loop* (когда он выполняется в основном потоке) предоставляет два метода, описанных ниже. Они позволяют добавлять и удалять обработчики сигналов, получаемых процессом, — это специфическая для Unix, ограниченная разновидность межпроцессного взаимодействия, неплохо описанная в Википедии (https://ru.wikipedia.org/wiki/Сигналы_(UNIX)).

add_signal_handler `add_signal_handler(signum, callback, *args)`

Устанавливает обработчик сигнала, заданного номером *signum*, для вызова `callback(*args)`.

remove_signal_handler `remove_signal_handler(signum)`

Удаляет обработчик сигнала, заданного номером *signum*, если таковой существует. Возвращает значение `True` в случае успешного удаления обработчика и значение `False` в случае отсутствия обработчика, подлежащего удалению (в любом случае после вызова данного метода обработчика, соответствующего сигналу *signum*, уже не существует).

Объект Executor

Объект *loop* может организовать выполнение функции в *исполнительном объекте* (executor) — пуле потоков или процессов, рассмотренном в разделе "Модуль `concurrent.futures`" главы 14: эта возможность полезна, если требуется выполнить некоторые операции блокирующего ввода-вывода или операции, интенсивно нагружающие процессор (в последнем случае используйте в качестве исполнительного объекта экземпляр класса `concurrent.futures.ProcessPoolExecutor`). Соответствующие два метода описаны ниже.

run_in_executor *coroutine* `run_in_executor(executor, func, *args)`

Возвращает объект сопрограммы, который выполняет функцию `func(*args)` в исполнительном объекте *executor* и возвращает результат работы функции *func* по ее завершении. Аргумент *executor* должен быть экземпляром класса `Executor` или иметь значение `None`, которому соответствует текущий исполнительный объект, используемый объектом *loop* по умолчанию.

set_default_executor `set_default_executor(executor)`

Устанавливает *executor* в качестве исполнительного объекта по умолчанию для объекта *loop*. Объект *executor* должен быть экземпляром класса `Executor` или иметь значение `None`, которому соответствует используемый по умолчанию исполнительный объект (пул потоков).

Обработка ошибок

Обработку исключений, возникающих в цикле событий, можно настраивать. Объект *loop* предоставляет для этой цели следующие четыре метода.

call_exception_handler `call_exception_handler(context)`

Вызывает текущий обработчик исключений объекта *loop*.

default_exception_handler `default_exception_handler(context)`

Обработчик исключений, предоставляемый классом объекта *loop*. Он вызывается, если возникает исключение,

	но обработчик не установлен, и может быть вызван обработчиком, чтобы обеспечить обработку исключения, предусмотренную по умолчанию.
get_exception_handler	`get_exception_handler()` Получает и возвращает текущий обработчик исключений объекта *loop* — вызываемый объект, имеющий два аргумента: *loop* и *context*.
set_exception_handler	`set_exception_handler(`*handler*`)` Устанавливает *handler* в качестве текущего обработчика исключений объекта *loop* — вызываемый объект, имеющий два аргумента: *loop* и *context*.

Объект *context* — это словарь, содержимое которого описано ниже (в будущих выпусках могут быть добавлены дополнительные ключи). Все ключи, за исключением *message*, — необязательные. Чтобы избежать возбуждения исключения `aKeyError` при попытке доступа к словарю *context* по ключу *key*, используйте вызов `context.get(`*key*`)`.

exception
: Экземпляр `Exception`.

future
: Экземпляр `asyncio.Future`.

handle
: Экземпляр `asyncio.Handle`.

message
: Экземпляр *str*, сообщение об ошибке.

protocol
: Экземпляр `asyncio.Protocol`.

socket
: Экземпляр `socket.socket`.

transport
: Экземпляр `asyncio.Transport`.

В следующих разделах обсуждаются три другие категории объектов, знакомство с которыми потребуется вам для работы с модулем `asyncio`, а также функциональность, предоставляемая модулем `asyncio` для каждой из этих категорий.

Фьючерсы

Класс `asyncio.Future` почти совместим с классом `Future`, предоставляемым модулем `concurrent.futures` и описанным в табл. 14.1 (планируется, что в одной из будущих версий Python оба интерфейса `Future` будут унифицированы, но

никаких гарантий, что эти намерения будут реализованы, не существует). Основные различия между экземпляром *af* класса `asyncio.Future` и экземпляром *cf* класса `concurrent.futures.Future` заключаются в следующем:

- *af* не является потокобезопасным;
- *af* нельзя передавать функциям `wait` и `as_completed` модуля `concurrent.futures`;
- методы `af.result` и `af.exception` не поддерживают аргумент *timeout* и могут вызываться только в том случае, если вызов `af.done()` возвращает значение `True`;
- метода *af*`.running()` не существует.

Для обеспечения потоковой безопасности обратные вызовы, добавленные с помощью метода *af*`.add_done_callback`, выполняются по завершении объекта *af* посредством вызова *loop*`.call_soon_threadsafe`.

Кроме того, объект *af* предоставляет три метода в дополнение и поверх тех, которые предоставляет объект *cf*.

`remove_done_callback` `remove_done_callback(`*func*`)`

Удаляет все экземпляры *func* из списка вызовов *af* и возвращает количество удаленных им экземпляров.

`set_exception` `set_exception(`*exception*`)`

Помечает объект *af* как завершенный и устанавливает *exception* в качестве его исключения. Если объект *af* уже завершен, то метод `set_exception` возбуждает исключение.

`set_result` `set_result(`*value*`)`

Помечает объект *af* как завершенный и устанавливает его результат равным *value*. Если объект *af* уже завершен, то метод `set_result` возбуждает исключение.

(В действительности объект *cf* также имеет методы `set_exception` и `set_result`, но в случае *cf* предполагается, что они должны вызываться исключительно блочными тестами и реализациями класса `Executor`. Идентичные методы объекта *af* не имеют таких ограничений.)

Наилучший способ создания объектов `Future` модуля `asyncio` — это использовать метод `create_future` объекта *loop*, не имеющий аргументов. В худшем случае вызов *loop*`.create_future()` делает то же самое, что и инструкция `return futures.Future(`*loop*`)`, но при этом альтернативные реализации цикла получают шанс переопределить данный метод и предоставить лучшую реализацию фьючерсов.

Задачи

Класс asyncio.Task является подклассом класса asyncio.Future: экземпляр *at* класса asyncio.Task обертывает объект сопрограммы и планирует его выполнение в цикле.

Данный класс определяет два метода класса: all_tasks(), который возвращает набор всех задач, определенных для объекта *loop*, и current_task(), который возвращает текущую задачу, выполняющуюся в данный момент (или значение None, если ни одна задача не выполняется).

Семантика метода *at*.cancel() слегка отличается от семантики метода cancel других фьючерсов: он не гарантирует отмену задачи, а возбуждает исключение CancelledError в обернутой сопрограмме — последняя может перехватить исключение (возможно, для выполнения завершающих операций по освобождению ресурсов или для того, чтобы отказать в отмене выполнения сопрограммы). Вызов *at*.cancelled() возвращает значение True только в том случае, если в обернутой сопрограмме имеется распространяющееся (или спонтанно возбужденное) исключение CancelledError.

Модуль asyncio предоставляет несколько функций, облегчающих работу с задачами и другими фьючерсами. Все они поддерживают необязательный (только именованный) аргумент loop=None, который позволяет использовать цикл событий, отличный от текущего, заданного по умолчанию (значению None соответствует использованию текущего цикла событий по умолчанию). Во всех методах в качестве аргументов, являющихся фьючерсами, также можно использовать объекты сопрограмм (которые автоматически обертываются экземплярами asyncio.Task). Ниже приводится описание этих функций.

as_completed as_completed(*futures*, *, *loop*=None, *timeout*=None)

Возвращает итератор, значениями которого являются экземпляры Future (возвращаемые примерно в порядке завершения). Если аргумент *timeout* не равен None, то он представляет значение в секундах (которое может иметь дробную часть), и в этом случае функция as_completed возбуждает исключение asyncio.TimeOuterror через *timeout* секунд, если все фьючерсы не завершатся к этому времени.

gather gather(**futures*, *, *loop*>=None, *return_exceptions*=False)

Возвращает единичный фьючерсный объект *f*, результатом выполнения которого является список результатов выполнения аргументов *futures*, расположенных в порядке завершения аргументов (все они должны быть фьючерсами в одном и том же цикле событий). Если аргумент *return_exceptions* имеет значение False, то любое исключение, возбужденное в содержащемся фьючерсе, немедленно распространяется через *f*. Если аргумент *return_exceptions* имеет значение True, то указанные исключения, как и результаты выполнения содержащихся фьючерсов, помещаются в список результатов *f*.

Вызов `f.cancel()` отменяет любой еще не выполненный содержащийся фьючерс. В случае независимой отмены любого содержащегося фьючерса это происходит так, как если бы он возбудил исключение `CancelledError` (следовательно, это не приведет к отмене фьючерса, если аргумент *return_exceptions* имеет значение `True`).

run_coroutine_threadsafe

`run_coroutine_threadsafe(`*coro*`, `*loop*`)`

Передает объект сопрограммы *coro* циклу событий *loop* и возвращает экземпляр `concurrent.futures.Future` для доступа к результату. Эта функция предназначена для того, чтобы разрешить другим потокам безопасно передавать объекты сопрограмм циклу.

shield

`shield(`*f*`, *, `*loop*`=None)`

Ожидает завершения экземпляра *f* класса `Future`, защищая *f* от отмены, если отменяется сопрограмма, выполняющая инструкцию `await asyncio.shield(`*f*`)` (или инструкцию `yield from asyncio.shield(`*f*`)`).

timeout

`timeout(`*timeout*`, *, `*loop*`=None)`

Возвращает менеджер контекста, который возбуждает исключение `asyncio.TimeoutError`, если блок не завершился по истечении *timeout* секунд (значение *timeout* может иметь дробную часть). Например:

```
try:
    with asyncio.timeout(0.5):
        await first()
        await second()
except asyncio.TimeoutError:
    print('Alas, too slow!')
else:
    print('Made it!')
```

Этот фрагмент выводит строку `Made it!`, если оба ожидающих объекта, `first()` и `second()`, последовательно завершаются в пределах половины секунды. В противном случае выводится строка `Alas, too slow!`.

wait

coroutine `wait(`*futures*`, *, `*loop*`=None, `*timeout*`=None, `*return_when*`=ALL_COMPLETED)`

Эта функция-сопрограмма возвращает объект сопрограммы, который ожидает завершения фьючерсов, содержащихся в непустом итерируемом объекте *futures*, и возвращает кортеж из двух множеств фьючерсов (*done*, *still_pending*). Аргумент *return_when* должен быть одной из трех констант, определенных в модуле `concurrent.futures`: `ALL_COMPLETED` (значение по умолчанию) — возврат, когда завершатся все фьючерсы (поэтому возвращаемое множество *still_pending* пустое); `FIRST_COMPLETED` — возврат, как только завершается любой из фьючерсов; `FIRST_EXCEPTION` — как в случае `ALL_COMPLETED`,

но возврат осуществляется также тогда, когда любой из фьючерсов возбуждает исключение (и в этом случае множество *still_pending* может быть непустым).

wait_for *coroutine* wait_for(*f*, *timeout*, *, *loop*=None)

Эта функция-сопрограмма возвращает объект сопрограммы, который ожидает завершения фьючерса *f* в пределах *timeout* секунд (аргумент *timeout* может иметь дробное значение, а также значение None, которому соответствует неограниченное время ожидания) и возвращает результат *f* (или же возбуждает исключение фьючерса *f* или исключение asyncio.TimeoutError, если истекает время *timeout*).

Классы transport и protocol

Для получения более подробной информации относительно классов transport и protocol обратитесь к соответствующему разделу онлайн-документации (https://docs.python.org/3/library/asyncio-protocol.html). Мы же ограничимся лишь обсуждением концептуальных основ и некоторых деталей, касающихся работы с этими классами, а также приведем пару примеров. Ключевая идея состоит в том, что класс transport предоставляет все средства, гарантирующие доставку потока (или датаграммы) "сырых", не интерпретированных байтов во внешнюю систему или извлечение их из внешней системы. Класс protocol выполняет прямую и обратную трансляцию байтов в значимые семантические сообщения.

Класс transport предоставляется модулем asyncio для абстрагирования любого из различных типов каналов передачи данных (TCP, UDP, SSL, каналы и т.п.). Вам не придется непосредственно инстанциализировать класс transport, вместо этого вы будете вызывать методы объекта *loop*, которые создают экземпляр transport и базовый канал и предоставляют созданный экземпляр.

Класс protocol предоставляется модулем asyncio для абстрагирования различных типов протоколов (потоковых, на основе диаграмм, каналов подпроцессов). Расширьте один из подходящих базовых классов, переопределив методы обратных вызовов, в которых вы хотите выполнить те или иные действия (базовые классы по умолчанию предоставляют пустые реализации таких методов, поэтому методы для событий, которые вас не интересуют, можно не переопределять). После этого передавайте свой класс в качестве аргумента *protocol_factory* методам объекта *loop*.

Экземпляр *p* класса protocol всегда имеет ассоциированный с ним экземпляр *t* класса Transport. Между этими экземплярами создается связь "один к одному". Как только соединение установлено, объект *loop* вызывает метод *p*.connection_made(t): объект *p* должен сохранить *t* в качестве атрибута объекта self и может вызвать для объекта *t* некоторые методы для инициализации и настройки параметров.

В случае потери или закрытия соединения объект *loop* вызывает метод p.connection_lost(*exc*), где аргумент может иметь значение None, указывающее на закрытие соединения обычным способом (как правило, по достижении конца файла, EOF), или предоставлять экземпляр Exception, содержащий информацию об ошибке, которая послужила причиной потери соединения.

Каждый из методов `connection_made` и `connection_lost` вызывается ровно один раз для каждого экземпляра *p*. Все другие обратные вызовы методов *p* происходят между этими двумя вызовами. Во время выполнения этих других вызовов экземпляр *p* получает от объекта *t* информацию о полученных данных или достижении конца файла EOF и/или запрашивает у экземпляра *t* отправку данных. Все взаимодействие экземпляров *p* и *t* осуществляется посредством выполнения обратных вызовов, определенных в одном из них, для другого.

Примеры использования протоколов: эхо-клиент и эхо-сервер

Ниже приведен основанный на использовании класса `protocol` пример реализации клиента для того же простого эхо-протокола, который был представлен в разделе "Клиент, ориентированный на установление соединения" главы 17. (Поскольку модуль `asyncio` существует только в версии v3, мы не заботились о совместимости кода этого примера с версией v2.)

```
import asyncio

data = """ Несколько строк текста,
включающих символы не из таблицы ASCII (€£),
для проверки работы как сервера,
так и клиента."""

class EchoClient(asyncio.Protocol):

  def __init__(self):
      self.data_iter = iter(data.splitlines())

  def write_one(self):
      chunk = next(self.data_iter, None)
      if chunk is None:
          self.transport.write_eof()
      else:
          line = chunk.encode()
          self.transport.write(line)
          print('Sent:', chunk)

  def connection_made(self, transport):
      self.transport = transport
      print('Connected to server')
      self.write_one()

  def connection_lost(self, exc):
      loop.stop()
      print('Disconnected from server')
```

```
  def data_received(self, data):
      print('Recv:', data.decode())
      self.write_one()

loop = asyncio.get_event_loop()
echo = loop.create_connection(EchoClient, 'localhost', 8881)
transport, protocol = loop.run_until_complete(echo)
loop.run_forever()
loop.close()
```

В обычных условиях вы не доставляли бы себе лишние хлопоты, используя модуль `asyncio` для столь простого клиента, который всего лишь отправляет данные на сервер, получает ответы и использует функцию `print` для отображения того, что при этом происходит. Однако цель данного примера состоит в том, чтобы продемонстрировать использование модуля `asyncio` (и, в частности, его протоколов) в клиенте (что было бы удобно, если бы клиент должен был обмениваться данными с несколькими серверами и/или одновременно выполнять другие неблокирующие операции ввода-вывода).

Тем не менее в этом примере ради сокращения объема кода введены определенные упрощения (например, вызов метода *loop*`.stop` в случае потери соединения), которые в высококачественном коде были бы неприемлемыми. С критическими замечаниями в адрес упрощенных примеров эхо-сервера и клиента и примером тщательно продуманной реализации этой же функциональности вы сможете ознакомиться на сайте GitHub по адресу `https://github.com/ambv/aioecho`.

Ниже приведен основанный на использовании класса `protocol` пример реализации сервера, работающего только в версии v3 и обеспечивающего ту же (намеренно упрощенную) эхо-функциональность.

```
import asyncio

class EchoServer(asyncio.Protocol):

  def connection_made(self, transport):
      self.transport = transport
      self.peer = transport.get_extra_info('peername')
      print('Connected from', self.peer)

  def connection_lost(self, exc):
      print('Disconnected from', self.peer)

  def data_received(self, data):
      print('Recv:', data.decode())
      self.transport.write(data)
      print('Echo:', data.decode())
```

```
loop = asyncio.get_event_loop()
echo = loop.create_server(EchoServer, 'localhost', 8881)
server = loop.run_until_complete(echo)
print('Serving at', server.sockets[0].getsockname())
loop.run_forever()
```

В коде этого сервера отсутствуют какие-либо внутренние ограничения на количество одновременно обслуживаемых клиентов, а класс `transport` обрабатывает любую фрагментацию данных, передаваемых по сети.

Потоки `asyncio`

Фундаментальные операции, выполняемые объектами `transport` и `protocol`, описанными в предыдущем разделе, основаны на парадигме обратных вызовов. Как отмечалось в разделе "Асинхронные архитектуры на основе сопрограмм", представление кода, который мог бы быть линейным потоком инструкций, в виде фрагментированного набора многочисленных функций и методов может затруднить процесс разработки. С помощью модуля `asyncio` вы можете частично избавиться от этого, используя сопрограммы, фьючерсы и задачи для реализации той же функциональности, однако между экземплярами протоколов и подобными инструментами отсутствует какая-либо внутренняя связь, поэтому вы фактически должны связать их самостоятельно. Вполне разумно стремиться к более высокому уровню абстракции, сфокусированному непосредственно на сопрограммах, для достижения тех же целей, которых можно было бы достигнуть, используя объекты `transport`, `protocol` и их обратные вызовы.

Модуль `asyncio` приходит на выручку, предоставляя *потоки* (streams) на основе сопрограмм, о чем можно прочитать в онлайн-документации (`https://docs.python.org/3/library/asyncio-stream.html`). Модуль `asyncio` непосредственно предоставляет четыре вспомогательные функции-сопрограммы на основе потоков: `open_connection`, `open_unix_connection`, `start_server` и `start_unix_server`. Будучи полезными сами по себе, они существуют главным образом для того, чтобы продемонстрировать, как лучше всего создавать потоки, и в самой документации содержится настоятельная рекомендация использовать эти образцы кода посредством их копирования и внесения в них необходимых изменений. Вы без труда найдете соответствующий исходный код, например, на сайте GitHub (`https://github.com/python/cpython/blob/2d264235f6e066611b412f7c2e1603866e0f7f1b/Lib/asyncio/streams.py`). Та же логика применима (и явно задокументирована в комментариях к исходному коду) к самим классам потоков: `StreamReader` и `StreamWriter`. Эти классы обертывают объекты `transport` и `protocol` и в необходимых случаях предоставляют методы-сопрограммы.

Ниже приведен пример реализации клиента на основе потоков для того же протокола, который был представлен в разделе "Клиент, ориентированный на установление соединения" главы 17, с использованием традиционного (до выхода версии Python 3.5) подхода к сопрограммам.

```
import asyncio

data = """ Несколько строк текста,
включающих символы не из таблицы ASCII (€£),
для проверки работы как сервера,
так и клиента."""

@asyncio.coroutine
def echo_client(data):
    reader, writer = yield from asyncio.open_connection(
        'localhost', 8881)
    print('Connected to server')
    for line in data:
        writer.write(line.encode())
        print('Sent:', line)
        response = yield from reader.read(1024)
        print('Recv:', response.decode())
    writer.close()
    print('Disconnected from server')

loop = asyncio.get_event_loop()
loop.run_until_complete(echo_client(data.splitlines()))
loop.close()
```

Функция `asyncio.open_connection` в конечном счете (посредством результата своего объекта сопрограммы, которого ожидает инструкция `yield from waits`) возвращает пару потоков, `reader` и `writer`, а далее все просто (добавляется еще одна инструкция `yield from` для получения конечного результата `reader.read`). Использование современного (Python 3.5) варианта сопрограммы не сопровождается никакими осложнениями.

```
import asyncio

data = """ Несколько строк текста,
включающих символы не из таблицы ASCII (€£),
для проверки работы как сервера,
так и клиента."""

async def echo_client(data):
    reader, writer = await asyncio.open_connection(
        'localhost', 8881)
    print('Connected to server')
    for line in data:
        writer.write(line.encode())
        print('Sent:', line)
        response = await reader.read(1024)
        print('Recv:', response.decode())
    writer.close()
```

```
    print('Disconnected from server')

loop = asyncio.get_event_loop()
loop.run_until_complete(echo_client(data.splitlines()))
loop.close()
```

Необходимые в этом случае преобразования кода носят исключительно механический характер: удалить декоратор, заменить `def` на `async def` и заменить `yield from` на `await`.

То же справедливо и в отношении серверов на основе потоков. Ниже приведен пример сервера, в котором используются традиционные сопрограммы.

```
import asyncio

@asyncio.coroutine
def handle(reader, writer):
    address = writer.get_extra_info('peername')
    print('Connected from', address)

    while True:
        data = yield from reader.read(1024)
        if not data: break
        s = data.decode()
        print('Recv:', s)
        writer.write(data)
        yield from writer.drain()
        print('Echo:', s)

    writer.close()
    print('Disconnected from', address)

loop = asyncio.get_event_loop()
echo = asyncio.start_server(handle, 'localhost', 8881)
server = loop.run_until_complete(echo)

print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

А вот эквивалентный пример, в котором используются современные сопрограммы и который также требует внесения исключительно изменений механического характера.

```
import asyncio

async def handle(reader, writer):
    address = writer.get_extra_info('peername')
    print('Connected from', address)

    while True:
        data = await reader.read(1024)
        if not data: break
        s = data.decode()
        print('Recv:', s)
        writer.write(data)
        await writer.drain()
        print('Echo:', s)

    writer.close()
    print('Disconnected from', address)

loop = asyncio.get_event_loop()
echo = asyncio.start_server(handle, 'localhost', 8881)
server = loop.run_until_complete(echo)

print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

Те же самые соображения, которые были приведены в предыдущем разделе, применимы и к этим примерам клиента и сервера: каждый из клиентов, в том виде как он есть, можно считать "перегибом", если учесть, насколько проста решаемая им задача (но полезно показать, как действовали бы клиенты на основе модуля `asyncio` при решении более сложных задач). Для каждого из серверов не существует внутренних ограничений на количество клиентов, которое он в состоянии обслуживать, а кроме того, они нечувствительны к любой фрагментации данных, которая может происходить при передаче данных по сети.

Синхронизация и очереди

Модуль `asyncio` предлагает классы для синхронизации сопрограмм, аналогичные тем, которые предлагает модуль `threading` для синхронизации потоков (см. раздел "Объекты синхронизации потоков" в главе 14), и очереди, аналогичные потокобезопасным очередям (см. раздел "Модуль `queue`" в главе 14). В данном разделе

документированы небольшие различия между классами синхронизации и очередей, предлагаемыми модулем asyncio, и одноименными классами, рассмотренными в главе 14.

Примитивы синхронизации модуля asyncio

Модуль asyncio предоставляет классы BoundedSemaphore, Condition, Event, Loop и Semaphore, очень похожие на одноименные классы, рассмотренные в разделе "Объекты синхронизации потоков" главы 14, за исключением, разумеется, того, что классы модуля asyncio синхронизируют сопрограммы, а не потоки. Классы синхронизации asyncio и их методы не поддерживают аргумент *timeout* (вместо этого используйте функцию asyncio.wait_for, чтобы применить тайм-аут к сопрограмме или фьючерсу модуля asyncio).

Некоторые методы классов синхронизации asyncio (а именно те, которые имеют отношение к захвату или ожиданию экземпляра класса синхронизации, и потому существует вероятность их блокирования) являются методами-сопрограммами, возвращающими объекты сопрограмм. В частности, это относится к методам BoundedSemaphore.acquire, Condition.acquire, Condition.wait, Condition.wait_for, Event.wait, Loop.acquire и Semaphore.acquire.

Очереди модуля asyncio

Модуль asyncio предоставляет класс Queue и его подклассы LifoQueue и PriorityQueue, весьма сходные с одноименными классами, рассмотренными в разделе "Модуль queue" главы 14, за исключением, разумеется, того, что классы модуля asyncio управляют очередями данных между сопрограммами, а не потоками. Классы очередей asyncio и их методы не поддерживают аргумент *timeout* (вместо этого используйте функцию asyncio.wait_for, чтобы применить тайм-аут к сопрограмме или фьючерсу модуля asyncio).

Некоторые методы классов очередей asyncio (все те, для которых существует вероятность их блокирования) являются методами-сопрограммами, возвращающими объекты сопрограмм. В частности, это относится к методам get, join и put класса Queue и обоих его подклассов.

Модуль selectors

Модуль selectors включен в стандартную библиотеку Python только в версии v3. Однако вы можете установить ретроподдержку версии v2 посредством команды **pip2 install selectors34**, а затем, чтобы обеспечить переносимость кода, выполнить импорт следующим образом.

```
try:
    import selectors
except ImportError:
    import selectors34 as selectors
```

Модуль selectors поддерживает мультиплексирование ввода-вывода, выбирая наилучший для вашей платформы механизм из низкоуровневого модуля select. От вас вовсе не требуется знания каких-либо низкоуровневых деталей модуля select, и поэтому он не рассматривается в данной книге.

Виды файлов, с которыми способен работать модуль selectors, зависят от вашей платформы. Сокеты (см. главу 17) поддерживаются на всех платформах. В Windows никакие другие виды файлов не поддерживаются. В Unix-подобных системах могут поддерживаться также другие виды файлов (для открытия файлов в неблокирующем режиме используйте низкоуровневый вызов os.open с флагом os.O_NONBLOCK, описанным в табл. 10.5).

События модуля selectors

Модуль selectors предоставляет две константы, EVENT_READ (файл доступен для чтения) и EVENT_WRITE (файл доступен для записи). Для одновременной индикации обоих событий используйте оператор побитового ИЛИ (|), как показано ниже:

```
selectors.EVENT_READ | selectors.EVENT_WRITE
```

Класс SelectorKey

Методы селекторных классов возвращают экземпляры класса SelectorKey — именованного кортежа, предоставляющего четыре атрибута, которые описаны ниже.

data
: Данные, связанные с файловым объектом при его регистрации с помощью селектора.

event
: Любая из констант, упомянутых в разделе "События модуля selectors", или результат их объединения с помощью оператора побитового ИЛИ.

fd
: Дескриптор файла, с которым связано произошедшее событие.

fileobj
: Файловый объект, зарегистрированный с помощью селекторного класса: им может быть либо целое число (в этом случае оно эквивалентно *fd*), либо объект с методом fileno() (в этом случае *fd* равен *fileobj*.fileno()).

Класс BaseSelector

Чтобы получить экземпляр конкретного подкласса абстрактного класса BaseSelector, оптимального для вашей платформы, вызовите функцию selectors.DefaultSelector(). (Не беспокойтесь по поводу специфических деталей конкретного класса, который вы получаете; все они реализуют одну и ту же функциональность.)

Каждый такой класс является менеджером контекста и потому пригоден для использования в инструкции with, обеспечивающей гарантированное закрытие селектора, когда работа с ним закончена.

```
with selectors.DefaultSelector() as sel:
    # используйте sel в этом блоке или в вызываемых в нем функциях
...
# здесь, по завершении блока, sel корректно закрывается
```

Экземпляр селектора *s* предоставляет следующие методы.

close `close()`

Освобождает все ресурсы *s*. Использовать *s* после вызова `s.close()` нельзя.

get_key `get_key(`*fileobj*`)`

Возвращает экземпляр `SelectorKey`, связанный с файловым объектом *fileobj*, или возбуждает исключение `KeyError`, если *fileobj* не зарегистрирован.

get_map `get_map()`

Возвращает отображение с файлами в качестве ключей и экземплярами `SelectorKey` в качестве значений.

modify `modify(`*fileobj*`, `*events*`, `*data*`=None)`

Аналогичен вызову `s.unregister(`*fileobj*`)`, за которым следует вызов `s.register(`*fileobj*`, `*events*`, `*data*`)`, но работает быстрее. Возвращает экземпляр `SelectorKey`, связанный с файловым объектом *fileobj*, или возбуждает исключение `KeyError`, если *fileobj* не зарегистрирован.

register `register(`*fileobj*`, `*events*`, `*data*`=None)`

Регистрирует файловый объект для выбора и мониторинга запрошенных событий, а также связывания с ним произвольного элемента данных. Аргументом *fileobj* может быть либо целое число (дескриптор файла), либо любой объект с методом `.fileno()`, возвращающим дескриптор файла. Аргументом *events* может быть одна из констант `EVENT_READ` и `EVENT_WRITE` или их объединение с помощью побитового оператора ИЛИ. Аргумент *data* — произвольный объект. Часто используемая идиома — передача в качестве аргумента *data* функции, которая будет вызываться при наступлении отслеживаемого события.

Возвращает новый экземпляр `SelectorKey` или возбуждает исключение `KeyError`, если файловый объект *fileobj* уже зарегистрирован, и исключение `ValueError`, если *fileobj* и/или события недействительны.

select `select(`*timeout*`=None)`

Возвращает список пар `(`*sk*`, `*events*`)` для файловых объектов, готовых к выполнению операций чтения и/или записи. Возвращает пустой список в случае превышения времени ожидания (или, в зависимости от версии Python, если ожидание прервано сигналом). Элемент *sk* — экземпляр `SelectorKey`.

Элемент *events* — это константа EVENT_READ, EVENT_WRITE или их объединение с помощью оператора побитового ИЛИ, если отслеживаются оба события и оба происходят.

Если *timeout*>0, то метод select ожидает в течение времени (измеряемого в секундах), не превышающего значение *timeout* (которое может иметь дробную часть), возможно, возвращая пустой список.

unregister unregister(*fileobj*)

Отменяет регистрацию файлового объекта *fileobj* для мониторинга событий. Вызывайте метод *s*.unregister для файла, прежде чем закрыть файл. Возвращает экземпляр SelectorKey, связанный с *fileobj*, или возбуждает исключение KeyError, если *fileobj* не был зарегистрирован.

В каких ситуациях использовать селекторы

Модуль selectors довольно низкоуровневый, но иногда это может пригодиться, когда на первый план выходят вопросы производительности, особенно в случае возложения на сокет несложных задач.

Пример эхо-сервера, использующего селекторы

Ниже приведен пример сервера на основе модуля selectors, который использует упрощенный эхо-протокол, рассмотренный в разделе "Клиент, ориентированный на установление соединения" главы 17. Код этого примера написан таким образом, чтобы его можно было выполнить как в версии v2 (при условии, что установлена ретроподдержка в виде модуля selectors34; о ретроподдержке рассказывалось в разделе "Модуль concurrent.futures" главы 14), так и в версии v3. В этом примере выводится больше информации, чем в примере, приведенном в разделе "Сервер, ориентированный на установление соединения" главы 17, чтобы лучше продемонстрировать работу низкоуровневых механизмов взаимодействия сразу с несколькими клиентами с использованием байтовых порций, размер которых оптимизирован для выполнения операций ввода-вывода.

```
from __future__ import print_function
import socket
try:
    import selectors
except ImportError:
    import selectors34 as selectors
ALL = selectors.EVENT_READ | selectors.EVENT_WRITE

# Сопоставление сокета с байтами, которые будут
# через него пересылаться, если это возможно
data = {}

# Набор сокетов, подлежащих закрытию с отменой регистрации,
# если это возможно
```

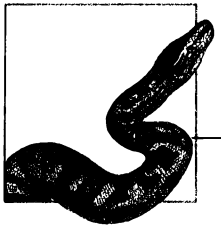
```
socks_to_close = set()

def accept(sel, data, servsock, unused_events):
    # Принять новое соединение, зарегистрировать его
    # с помощью селектора
    sock, address = servsock.accept()
    print('Connected from', address)
    sel.register(sock, ALL, handle)

def finish(sel, data, sock):
    # Сокет должен быть закрыт, а его регистрация отменена,
    # если он находится в селекторе
    if sock in sel.get_map():
        ad = sock.getpeername()
        print('Disconnected from {}'.format(ad))
        sel.unregister(sock)
        sock.close()
    data.pop(sock, None)

def handle(sel, data, sock, events):
    # Клиентский сокет, только что подготовленный для
    # записи и/или чтения
    ad = sock.getpeername()
    if events & selectors.EVENT_WRITE:
        tosend = data.get(sock)
        if tosend:
            # Отправить столько, сколько готов принять клиент
            nsent = sock.send(tosend)
            s = tosend.decode('utf-8', errors='replace')
            print(u'{} bytes sent to {} ({})'.format(nsent, ad, s))
            # Байты, которые должны быть отправлены позже,
            # если таковые имеются
            tosend = tosend[nsent:]
        if tosend:
            # Дополнительные байты для последующей отправки,
            # отслеживать их
            print('{} bytes remain for {}'.format(len(tosend), ad))
            data[sock] = tosend
        else:
            # Больше нет байтов для отправки -- пока что
            # игнорировать EVENT_WRITE
            print('No bytes remain for {}'.format(ad))
            data.pop(sock, None)
            sel.modify(sock, selectors.EVENT_READ, handle)
    if events & selectors.EVENT_READ:
        # Получить вплоть до 1024 байта за один раз
        newdata = sock.recv(1024)
        if newdata:
            # Добавить новые полученные данные как такие,
```

```
            # которые должны быть отправлены обратно
            s = newdata.decode('utf-8', errors='replace')
            print(u'Got {} bytes from {} ({})'.format(
                  len(newdata), ad, s))
            data[sock] = data.get(sock, b'') + newdata
            sel.modify(sock, ALL, handle)
        else:
            # Получение пустой строки означает отключение клиента
            socks_to_close.add(sock)

try:
    servsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    servsock.bind(('', 8881))
    servsock.listen(5)
    print('Serving at', servsock.getsockname())
    with selectors.DefaultSelector() as sel:
        sel.register(servsock, selectors.EVENT_READ, accept)
        while True:
            for sk, events in sel.select():
                sk.data(sel, data, sk.fileobj, events)
            while socks_to_close:
                sock = socks_to_close.pop()
                finish(sel, data, sock)
except KeyboardInterrupt:
    pass
    finally:
    servsock.close()
```

Модуль `selectors` не предоставляет цикл событий. Цикл событий, необходимый для асинхронного программирования, в данном примере реализован с помощью инструкции `while True:`.

Продемонстрированный в этом примере низкоуровневый подход заставил нас написать большой объем кода, несоразмерный с простотой задачи, особенно с учетом того факта, что от сервера не требуется проверять или обрабатывать входящие байты (в большинстве сетевых задач входящие потоки или порции байтов *должны* анализироваться с целью выделения из них семантически значимых сообщений). Положительной стороной такого похода, характеризующегося минимальным потреблением ресурсов, является достижение высокой производительности.

Кроме того, следует особо отметить (о чем можно судить по детализированной информации, выводимой с помощью функции `print`), что представленный в этом примере сервер не страдает ограничениями сервера, описанного в разделе "Сервер, ориентированный на установление соединения" главы 17. В том виде, как он приведен, данный код способен обрабатывать любое количество клиентов, ограниченное лишь объемом доступной памяти, и может справиться с любой фрагментацией передаваемых по сети данных с использованием сколь угодно большого количества одновременных соединений



19

Модули для работы с клиентскими сетевыми протоколами

Стандартная библиотека Python предоставляет несколько модулей, упрощающих использование протоколов Интернета, в частности на стороне клиента (а также в некоторых простых серверах). Намного большее количество подобных пакетов можно найти на сайте Python Package Index (`https://pypi.python.org/pypi`). Эти сторонние пакеты поддерживают широкий спектр протоколов, а некоторые из них предоставляют лучшие программные интерфейсы, чем их эквиваленты, предлагаемые стандартной библиотекой. Если у вас возникает потребность в использовании протокола, не поддерживаемого стандартной библиотекой, или его поддержка вас не удовлетворяет, обязательно посетите сайт PyPI — не исключено, что вам удастся подобрать там лучшее решение.

В этой главе мы рассмотрим ряд пакетов стандартной библиотеки, которые могут оказаться вполне удовлетворительными для некоторых вариантов использования сетевых протоколов, особенно простых: если вы напишете код, не требующий привлечения сторонних пакетов, то это облегчит установку вашего приложения на других компьютерах. Мы также кратко обсудим несколько сторонних пакетов, предназначенных для работы с некоторыми важными сетевыми протоколами, не включенными в стандартную библиотеку. Эта глава не охватывает сторонние пакеты, использующие подход на основе асинхронного программирования (за информацией относительно таких пакетов обратитесь к разделу "Модуль `asyncio` (только в версии v3)" главы 18).

Что касается использования протокола HTTP[1] на стороне клиента, а также других сетевых ресурсов (например, анонимных FTP-сайтов), доступ к которым лучше всего

[1] HTTP (Hypertext Transfer Protocol — протокол передачи гипертекста) — основной протокол Всемирной паутины: он используется каждым веб-сервером и браузером и стал доминирующим протоколом прикладного уровня в Интернете.

получать с помощью URL-адресов, то их поддержка, предлагаемая стандартной библиотекой, несколько усложнена и не обеспечивает совместимости между версиями v2 и v3. С учетом этого обстоятельства мы решили не обсуждать соответствующие модули стандартной библиотеки, а рассмотреть и рекомендовать вам для использования великолепный сторонний пакет `requests` (`http://docs.python-requests.org/en/master/`) с его хорошо продуманным API, который обеспечивает v2/v3-совместимость.

Протоколы электронной почты

В наши дни для *отправки* электронной почты в большинстве случаев используются серверы, реализующие протокол SMTP (Simple Mail Transport Protocol — простой протокол передачи почты), а для *получения* — серверы и клиенты, реализующие протокол POP3 (Post Office Protocol Version 3 — протокол почтового отделения, версия 3) и/или протокол IMAP4: его оригинальную версию, описанную в спецификации **RFC 1730** (`https://tools.ietf.org/html/rfc1730.html`), или версию IMAP4rev1, описанную в спецификации **RFC 2060** (`https://tools.ietf.org/html/rfc2060.html`). На стороне клиента эти протоколы поддерживаются модулями `smtplib`, `poplib` и `imaplib` стандартной библиотеки Python.

Если вам необходимо написать клиентский код, работающий либо по протоколу POP3, либо по протоколу IMAP4, то в качестве стандартной рекомендации можно было бы посоветовать вам остановить свой выбор на протоколе IMAP4, поскольку он обладает более широкими возможностями и зачастую — согласно собственной онлайн-документации Python — более строго реализован на стороне сервера. К сожалению, модуль `imaplib` стандартной библиотеки, предназначенный для работы с IMAP4, чересчур сложен и слишком обширен, чтобы его можно было рассмотреть в данной книге. Если вы все же решитесь пойти этим путем, используйте онлайн-документацию наряду с обязательным изучением объемистой спецификации RFC 1730 или 2060 и, возможно, других родственных спецификаций, таких как RFC 5161 и 6855 (описание возможностей) или RFC 2342 (описание пространств имен). Избежать изучения указанных спецификаций в дополнение к онлайн-документации стандартного библиотечного модуля вам просто не удастся: многие из аргументов, передаваемых при вызове функций и методов, содержащихся в модуле `imaplib`, а также результаты, возвращаемые этими вызовами, представляют собой строки, формат которых описан только в соответствующих документах RFC, а не в онлайн-документации Python. Именно из этих соображений модуль `imaplib` не рассматривается в данной книге. В последующих разделах мы обсудим лишь модули `poplib` и `stmplib`.

Если у вас все же возникнет желание использовать мощные возможности протокола, то мы только приветствовали бы это, но лучше не привлекать непосредственно модуль `imaplib` стандартной библиотеки, а воспользоваться более простым высокоуровневым сторонним пакетом IMAPClient (`https://pypi.python.org/pypi/IMAPClient`), доступным для установки с помощью команды **`pip install`**

как в версии v2, так и в версии v3, и подробно описанным в онлайн-документации (`http://imapclient.readthedocs.io/en/stable/`).

Модуль poplib

Модуль `poplib` предоставляет класс `POP3`, обеспечивающий доступ к почтовому ящику POP. Спецификация протокола POP содержится в документе **RFC 1939** (`https://www.ietf.org/rfc/rfc1939.txt`).

POP3 `class POP3(`*host*`, `*port*`=110)`

Возвращает экземпляр *p* класса `POP3`, подключенный к хосту и порту. Родственный ему класс `POP3_SSL` ведет себя точно так же, но подключается к хосту (по умолчанию через порт 995) посредством более безопасного TLS-канала. Он необходим для подключения к таким почтовым серверам, выдвигающим минимальные требования безопасности, как `'pop.gmail.com'`*.

* В частности, для подключения к учетной записи Gmail вам также потребуется подкорректировать ее настройки: включить POP, разрешить использование ненадежных приложений и убедиться в том, что не используется двухфакторная аутентификация, — действия, которые, вообще говоря, мы не можем рекомендовать, поскольку они ослабляют защиту вашей почты.

Экземпляр *p* предоставляет множество методов, наиболее часто используемые из которых приведены ниже (во всех случаях *msgnum* — числовой идентификатор сообщения в виде строки или целого значения).

dele *p*`.dele(`*msgnum*`)`

Помечает сообщение *msgnum* для удаления и возвращает строку ответа сервера. Сервер помещает подобные запросы на удаление сообщений в очередь и выполняет их позже, когда вы закроете соединение посредством вызова *p*`.quit`.

list *p*`.list(`*msgnum*`=None)`

Возвращает кортеж (*response*, *messages*, *octets*), где *response* — строка ответа сервера; *messages* — список байтовых строк, по одной для каждого из сообщений в почтовом ящике, каждая из которых включает два слова `b'msgnum` *bytes*`'`, представляющие номер сообщения и его размер в байтах; а *octets* — размер в байтах всей строки ответа. Если аргумент *msgnum* не равен `None`, то метод `list` возвращает вместо кортежа одну строку, представляющую ответ сервера для заданного сообщения *msgnum*.

pass_ *p*`.pass_(`*password*`)`

Отправляет пароль на сервер. Данный метод должен вызываться после вызова метода *p*`.user`. Замыкающий символ подчеркивания в имени метода необходим, поскольку `pass` является ключевым словом Python. Возвращает строку ответа сервера.

quit *p*.quit()

Завершает сеанс и приказывает серверу выполнить удаления, запрошенные вызовами top.dele. Возвращает строку ответа сервера.

retr *p*.retr(*msgnum*)

Возвращает кортеж (*response*, lines, *bytes*), где *response* — строка ответа сервера, *lines* — список всех строк сообщения *msgnum* в виде байтовых строк, *bytes* — общее количество байтов в сообщении.

set_debuglevel *p*.set_debuglevel(*debug_level*)

Устанавливает уровень отладки в соответствии с целочисленным значением аргумента *debug_level*: 0 (значение по умолчанию) — не выполнять отладку, 1 — умеренный вывод отладочной информации, 2 и больше — полный вывод всей управляющей информации, обмениваемой с сервером.

stat *p*.stat()

Возвращает пару (*num_msgs*, *bytes*), где *num_msgs* — количество сообщений в почтовом ящике, *bytes* — полное количество байтов.

top *p*.top(*msgnum*, *maxlines*)

Подобен методу *retr*, но возвращает не более *maxlines* строк тела сообщения (также возвращает все строки заголовков, как это делает метод retr). Может пригодиться для выбора только начальных строк длинных сообщений.

user *p*.user(*username*)

Отправляет серверу имя пользователя *username*; за этим вызовом следует вызов метода *p*.pass_.

Модуль smtplib

Модуль smtplib предоставляет класс SMTP, предназначенный для отправки почты на SMTP-сервер. Спецификация SMTP-протокола содержится в документе RFC 2821.

SMTP class SMTP([*host*, *port*=25])

Возвращает экземпляр *s* класса SMTP. Если предоставлен аргумент *host* (и, возможно, необязательный аргумент *port*), неявно вызывает метод *s*.connect(*host*, *port*).

Сестринский класс SMTP_SSL ведет себя точно так же, но подключается к хосту (по умолчанию через порт 465) посредством более безопасного TLS-канала. Он необходим для подключения к таким почтовым серверам, выдвигающим минимальные требования безопасности, как 'smtp.gmail.com'.

Экземпляр *s* предоставляет множество методов, наиболее часто используемые из которых приведены ниже.

connect *s*.connect(*host*=127.0.0.1, *port*=25)

Подключается к SMTP-серверу, используя указанные хост (по умолчанию — локальный хост) и порт. Порт 25 используется по умолчанию для SMTP-службы. Порт 465 используется по умолчанию для более безопасного подключения "SMTP поверх TLS".

login *s*.login(*user*, *password*)

Выполняет вход на сервер, используя указанные имя пользователя и пароль. Этот вызов необходим только в том случае, если SMTP-сервер требует аутентификации (в наши дни ее требуют практически все серверы).

quit *s*.quit()

Завершает SMTP-сеанс.

sendmail *s*.sendmail(*from_addr*, *to_addrs*, *msg_string*)

Посылает почтовое сообщение *msg_string* от имени отправителя, адрес которого содержится в строке *from_addr*, каждому из получателей, адреса которых содержатся в элементах списка *to_addrs*. Аргумент *msg_string* должен быть полным сообщением, соответствующим спецификации RFC 822 и содержащим в одной многострочной байтовой строке следующие компоненты: заголовки, пустую строку-разделитель и тело сообщения. Аргументы *from_addr* и *to_addrs* лишь направляют почтовый транспорт, не добавляя и не изменяя заголовки в строке *msg_string*. Чтобы подготовить сообщения, совместимые со спецификацией RFC 822, используйте пакет `email`, рассмотренный в разделе "Обработка сообщений MIME и электронной почты" главы 21.

Клиенты HTTP и URL

Большую часть времени ваш код использует протоколы HTTP и FTP через URL-слой, расположенный на более высоком уровне абстракции и поддерживаемый модулями и пакетами, которые рассматриваются в следующих разделах. Стандартная библиотека Python также предлагает низкоуровневые модули, поддерживающие конкретные протоколы: для FTP-клиентов — `ftplib` (`https://docs.python.org/3/library/ftplib.html`); для HTTP-клиентов в версии v3 — `http.client` (`https://docs.python.org/3/library/http.client.html`) и в версии v2 — `httplib` (`https://docs.python.org/2.7/library/httplib.html`). (HTTP-серверы рассмотрены в главе 20). Если вам нужно написать FTP-сервер, обратите внимание на сторонний модуль `pyftpdlib` (`https://pypi.python.org/pypi/pyftpdlib/`). Реализации новейшего протокола HTTP/2 (`https://http2.github.io/`) все еще находятся на ранних стадиях разработки, но по состоянию дел на момент выхода книги вашим наилучшим выбором будет сторонний модуль `hyper`. (Как обычно, сторонние модули можно загружать и устанавливать из каталога PyPI — `https://pypi.python.org/pypi` — с помощью заслуживающего самых высоких похвал инструмента `pip`.) Мы не обсуждаем эти низкоуровневые модули в данной книге, сосредоточившись в следующих разделах на рассмотрении более абстрагированных способов доступа на уровне URL.

Доступ к ресурсам с помощью URL

URL — это разновидность URI (Uniform Resource Identifier — унифицированный идентификатор ресурса). URI — это строка, *идентифицирующая* ресурс (но не обязательно указывающая его местоположение), тогда как URL указывает *местонахождение* ресурса в Интернете. URL — это строка, состоящая из нескольких необязательных частей, называемых *компонентами*: *схема* (scheme), *местонахождение* (location), *путь* (path), *параметры* (query) и *якорь* (fragment). (Второй компонент иногда называют *сетевым расположением*.) Ниже приведен пример URL-строки, содержащей все компоненты.

```
scheme://lo.ca.ti.on/pa/th?qu=ery#fragment
```

В качестве другого примера рассмотрим следующую строку URL:

```
https://www.python.org/community/awards/psf-awards/#october-2016
```

В этой строке схема — `http`, сетевое расположение — `www.python.org`, путь — `/community/awards/psf-awards/`, параметры запроса отсутствуют, а якорь — `#october-2016`. (В большинстве схем по умолчанию используется "известный порт", если явно не указано иное. Например, `80` — это "известный порт" для схемы HTTP.) Одни знаки пунктуации являются частью одного из разделяемых ими компонентов, другие — всего лишь разделители, не являющиеся частью какого-либо компонента. Опускание знаков пунктуации неявно означает отсутствие компонента. Например, в строке `mailto:me@you.com` схема — это `mailto`, путь — `me@you.com`, а сетевое расположение, параметры запроса и якорь отсутствуют. Отсутствие символов `//` означает, что данный URI не содержит расположения, отсутствие символа `?` означает отсутствие параметров запроса, а отсутствие символа `#` — отсутствие якоря.

Если компонент расположения заканчивается двоеточием, за которым следует число, то этим числом обозначен TCP-порт конечной точки. В противном случае подключение использует "известный порт", ассоциируемый с данной схемой (например, порт `80` для HTTP).

Модули `urllib.parse` (v3) и `urlparse` (v2)

Модуль `urllib.parse` в версии v3 (`urlparse` в версии v2) предоставляет функции для анализа и синтеза URL-строк. Из них наиболее часто используются функции `urljoin`, `urlsplit` и `urlunsplit`.

`urljoin` `urljoin(`*`base_url_string`*`, `*`relative_url_string`*`)`

Возвращает URL-строку *u*, полученную присоединением строки *`relative_url_string`*, которая может быть строкой относительного URL, к строке *`base_url_string`*. Процедуру присоединения, которую выполняет метод `urljoin`, можно кратко описать следующим образом.

- Если одна из двух строк аргументов пустая, то строкой *u* является второй аргумент.
- Если строка *relative_url_string* явно определяет схему, отличную от схемы строки *base_url_string*, то строкой *u* является строка *relative_url_string*. В противном случае схемой строки *u* является схема строки *base_url_string*.
- Если схема не допускает относительные URL (например, схема `mailto`) или строка *relative_url_string* явно определяет расположение (даже если оно совпадает с тем, которое определено в строке *base_url_string*), то все остальные компоненты *u* берутся из *relative_url_string*. В противном случае расположением для строки *u* будет расположение, указанное в строке *base_url_string*.
- Путь строки *u* получается посредством объединения путей строк *base_url_string* и *relative_url_string* согласно стандартному синтаксису для абсолютных и относительных URL-путей в соответствии со спецификацией **RFC 1808** (`https://tools.ietf.org/html/rfc1808`).

```
from urllib import parse as urlparse # в версии v3
# в версии v2 было бы: import url parse
urlparse.urljoin(
  'http://somehost.com/some/path/here','../other/path')
# Результат: 'http://somehost.com/some/other/path'
```

urlsplit `urlsplit(`*url_string*, *default_scheme*`='', `*allow_fragments*`=True)`

Анализирует строку *url_string* и возвращает кортеж (фактически экземпляр `SplitResult`, который можно рассматривать как кортеж или использовать с именованными атрибутами) из пяти строковых элементов: *scheme*, *netloc*, *path*, *query* и *fragment*. Аргумент *default_scheme* является первым элементом, если в строке *url_string* отсутствует явная схема. Если аргумент *allow_fragments* равен `False`, то последним элементом кортежа всегда является пустая строка `''`, независимо от наличия якоря в строке *url_string*. Элементы, соответствующие отсутствующим компонентам, являются пустыми строками `''`.

```
urlparse.urlsplit(
'http://www.python.org:80/faq.cgi?src=fie')
# Результат:
# ('http','www.python.org:80','/faq.cgi','src=fie','')
```

urlunsplit `urlunsplit(`*url_tuple*`)`

Аргумент *url_tuple* — любой итерируемый объект, содержащий ровно пять элементов, которые все являются строками. Любое значение, возвращаемое вызовом `urlsplit`, является допустимым аргументом для `urlunsplit`. Метод `urlunsplit` возвращает строку URL с заданными компонентами и необходимыми разделителями, но опускает лишние разделители (например, результат не будет содержать разделителя #, если якорем, т.е. последним элементом, в строке *url_tuple* является пустая строка `''`).

```
urlparse.urlunsplit((
'http','www.python.org:80','/faq.cgi','src=fie',''))
# Результат: 'http://www.python.org:80/faq.cgi?src=fie'
```

Вызов `urlunsplit(urlsplit(x))` возвращает нормализованную форму URL-строки `x`, которая не обязательно совпадает с `x`, поскольку `x` не обязан быть нормализованным. Например:

```
urlparse.urlunsplit(
urlparse.urlsplit('http://a.com/path/a?'))
# Результат: 'http://a.com/path/a'
```

В данном случае нормализация гарантирует, что избыточные разделители, такие как замыкающий символ `?` в аргументе функции `urlsplit`, будут отсутствовать в результате.

Сторонний пакет `requests`

Сторонний пакет `requests` (`https://pypi.python.org/pypi/requests/`), для которого имеется отличная онлайн-документация (`http://docs.python-requests.org/en/master/`), поддерживает как версию v2, так и версию v3, и именно его мы рекомендуем использовать для получения доступа к URL-адресам HTTP. Как и в случае любого другого стороннего пакета, для установки пакета `requests` лучше всего использовать простую команду **`pip install requests`**. В этом разделе мы суммируем наилучшие способы применения данного пакета в несложных случаях.

Изначально пакет `requests` поддерживает лишь транспортный протокол HTTP. Для доступа к URL-адресам с использованием иных протоколов вам потребуется установить также другие сторонние пакеты (так называемые *адаптеры протоколов*), такие как пакет `requests-ftp` (`https://pypi.python.org/pypi/requests-ftp`) для URL-адресов FTP, или же пакеты, предоставляемые в виде части чрезвычайно полезного пакета `requests-toolbelt` (`http://toolbelt.readthedocs.io/en/latest/index.html`) с утилитами для пакета `requests`.

Функциональность пакета `requests` в основном базируется на трех предоставляемых им классах: `Request`, который моделирует HTTP-запрос, подлежащий отправке на сервер; `Response`, который моделирует HTTP-ответ сервера на запрос; и `Session`, который обеспечивает определенную непрерывность на протяжении ряда последовательно выполняемых запросов/ответов, называемую *сеансом*. Для обычного варианта взаимодействия, включающего один запрос и один ответ, в этом нет необходимости, и потому класс `Session` часто можно игнорировать.

Отправка запросов

Чаще всего вам не придется явно обращаться к классу `Request`, вместо этого вы будете вызывать вспомогательную функцию `request`, которая подготавливает и отправляет экземпляр `Request`, а также возвращает экземпляр `Response`. Функция `request` имеет два обязательных позиционных аргумента, оба строковых: `method` — используемый HTTP-метод и `url` — URL-адрес. Также она поддерживает многочисленные необязательные именованные аргументы (наиболее часто используемые из них рассматриваются в следующем разделе).

С целью упрощения обработки запросов и ответов модуль `requests` предоставляет функции, имена которых совпадают с именами методов HTTP: `delete`, `get`, `head`, `options`, `patch`, `post` и `put`. Каждая из этих функций имеет единственный обязательный позиционный аргумент, `url`, и те же необязательные именованные аргументы, что и функция `request`.

Если хотите обеспечить определенную непрерывность на протяжении нескольких запросов, вызовите конструктор `Session` для создания экземпляра *s*, а затем используйте методы этого экземпляра `request`, `get`, `post` и др., которые ведут себя точно так же, как и одноименные функции, непосредственно предоставляемые модулем `requests` (однако методы экземпляра *s* объединяют настройки *s* с необязательными именованными параметрами для подготовки запроса к отправке в соответствии с заданным `url`).

Необязательные именованные параметры функции `request`

Функция `request` (точно так же, как и функции `get`, `post` и др. и одноименные методы экземпляра *s* класса `Session`) поддерживает множество необязательных именованных параметров. За описанием их полного набора обратитесь к онлайн-документации пакета `requests` (`http://docs.python-requests.org/en/master/api/#requests.request`), если вам нужна такая расширенная функциональность, как управление через прокси-объекты, аутентификация, специальная обработка перенаправлений, потоковая передача, cookie-файлы и т.п. Ниже описаны наиболее часто используемые именованные параметры функции `request`.

`data`
: Словарь, последовательность пар "ключ–значение", байтовая строка или файловый объект, используемые в качестве тела запроса.

`headers`
: Словарь HTTP-заголовков, отправляемых в запросе.

`json`
: Данные Python (обычно словарь), преобразуемые в формат JSON для включения в тело запроса.

`files`
: Словарь с именами в качестве ключей и объектами наподобие файлов или *кортежами файлов* в качестве значений, используемый совместно с методом POST для выгрузки файлов с составным содержимым. Форматы значений параметра `files` обсуждаются в следующем разделе.

`params`
: Словарь или байтовая строка, отправляемая в качестве строки параметров запроса вместе с запросом.

`timeout`
: Значение с плавающей точкой, выражающее максимальное время ожидания (в секундах) ответа на запрос, прежде чем будет возбуждено исключение.

Параметры `data`, `json` и `files` обеспечивают несовместимые между собой способы определения тела запроса. Всегда задавайте только один из них и только для HTTP-методов, использующих тело запроса, а именно: `PATCH`, `POST` и `PUT`. Единственным исключением из этого правила является случай, когда вы можете одновременно предоставить параметр `data`, передающий словарь, и параметр `files`, причем эта возможность широко применяется: в данном случае как пары "ключ–значение" в словаре, так и файлы формируют тело запроса в виде единого набора составных данных в соответствии с документом **RFC 2388** (`https://www.ietf.org/rfc/rfc2388.txt`).

Аргумент `files` (и другие способы определения тела запроса)

Если вы определяете тело запроса с помощью параметров `json` или `data`, передавая байтовую строку или файловый объект (который должен быть открыт для чтения, обычно в двоичном режиме), то результирующие байты используются непосредственно в теле запроса. Если же вы определяете его с помощью параметра `data`, передавая словарь или последовательность пар "ключ–значение", то тело запроса создается в виде *формы*, включающей пары "ключ–значение", которые представляются в формате *application/x-www-form-urlencoded* согласно соответствующему стандарту (`https://url.spec.whatwg.org/#application/x-www-form-urlencoded`).

Если вы определяете тело запроса с помощью параметра `files`, то оно также создается в виде формы, но на этот раз в формате *multipart/form-data*, соответствующем составным данным (единственный способ выгрузки файлов с использованием методов PATCH, POST или PUT в HTTP-запросе). Каждый выгружаемый вами файл форматируется в собственную часть формы. Если в дополнение к этому вы хотите, чтобы форма предоставила серверу дополнительные параметры, то кроме аргумента `files` следует передать аргумент `data` со значением в виде словаря (или последовательности пар "ключ-значение"), содержащего дополнительные параметры, которые преобразуются в дополнительную часть составной формы.

Для большей гибкости аргументу `files` разрешено быть словарем (его элементы образуют произвольно упорядоченную последовательность пар "ключ–значение") или последовательностью пар "ключ–значение" (в последнем случае порядок расположения ее элементов сохраняется в результирующем теле запроса).

В любом случае каждое значение в парах "ключ–значение" может быть либо строкой типа `str` (или, что лучше всего[2], типа `bytes` или `bytearray`), и в этом случае оно включается непосредственно в содержимое выгружаемого файла, либо файловым объектом, открытым для чтения (далее модуль `requests` вызывает для этого объекта метод `read()` и использует полученный результат в качестве содержимого выгружаемого файла; в подобных случаях мы настоятельно рекомендуем открывать файл в двоичном режиме во избежание любой неоднозначности в отношении размера содержимого). Если выполняется любое из этих условий, то модуль `requests`

[2] Поскольку это обеспечивает полный и явный контроль на тем, какие в точности октеты выгружаются на сервер.

использует часть "имя" пары (например, значение ключа в словаре) в качестве имени файла (если только ему не удастся действовать еще эффективнее, поскольку открытый файловый объект способен определить имя собственно базового файла), делает наиболее правдоподобное предположение о типе содержимого файла и использует минимальные заголовки для части файла, относящейся к форме.

Возможен и другой вариант, когда значением в каждой паре "имя–значение" является кортеж, содержащий от двух до четырех элементов: *fn*, *fp*, [*ft*, [*fh*]] (здесь квадратные скобки используются в качестве метасинтаксиса для обозначения необязательных элементов). В данном случае *fn* — имя файла, *fp* — содержимое (точно так же, как в предыдущем абзаце), необязательный элемент *ft* — это тип содержимого (в случае его отсутствия модуль `requests` пытается угадать его, о чем было сказано в предыдущем абзаце), а необязательный словарь *fh* — это дополнительные заголовки для части файла, относящейся к форме.

Как необходимо изучать примеры использования модуля `requests`

В реальных приложениях обычно не приходится обращаться к внутреннему экземпляру *r* класса `requests.Request`, функции которого, такие как `requests.post`, выполняют от вашего имени все операции по созданию, подготовке и отправке запросов. Однако для того, чтобы детально разобраться в том, что именно делает модуль `requests`, полезно самостоятельно выполнить соответствующие операции на более низком уровне абстракции. (Создание, подготовка и исследование экземпляра *r* — в его отправке нет никакой потребности!) Например, код

```
import requests

r = requests.Request('GET', 'http://www.example.com',
    data={'foo': 'bar'}, params={'fie': 'foo'})
p = r.prepare()
print(p.url)
print(p.headers)
print(p.body)
```

выводит следующую информацию (вывод результата вызова `p.headers` разбит на две строки для улучшения читаемости текста):

```
http://www.example.com/?fie=foo
{'Content-Length': '7',
 'Content-Type': 'application/x-www-form-urlencoded'}
foo=bar
```

Аналогичным образом, если используется параметр `files`:

```
import requests

r = requests.Request('POST', 'http://www.example.com',
    data={'foo': 'bar'}, files={'fie': 'foo'})
p = r.prepare()
```

```
print(p.headers)
print(p.body)
```

то выводится следующая информация (некоторые строки разбиты на несколько строк для улучшения читаемости текста):

```
{'Content-Type': 'multipart/form-data;
  boundary=5d1cf4890fcc4aa280304c379e62607b',
  'Content-Length': '228'}
b'--5d1cf4890fcc4aa280304c379e62607b\r\nContent-Disposition: formdata;
name="foo"\r\n\r\nbar\r\n--5d1cf4890fcc4aa280304c379e62607b\r
\nContent-Disposition: form-data; name="fie"; filename="fie"\r
\n\r\nfoo\r\n--5d1cf4890fcc4aa280304c379e62607b--\r\n'
```

Удачи в интерактивных исследованиях!

Класс Response

В модуле `requests` есть один класс, о котором вы никогда не должны забывать, — это класс `Response`. Каждый запрос, посланный серверу (обычно это неявно делают такие методы, как `get`), возвращает экземпляр *r* класса `requests.Response`.

Первое, что вы захотите сделать, — это проверить целочисленное значение `r.status_code`, которое информирует вас о состоянии запроса с помощью типичных HTTP-кодов: `200` означает "все хорошо", `404` — "страница не найдена" и т.д. Если вместо этого вы захотите просто получать исключения для кодов состояния, указывающих на тип ошибки, используйте вызов `r.raise_for_status()`. Он не сделает ничего в случае успешного запроса, но в противном случае возбудит исключение `requests.exceptions.HTTPError`. (Другие исключения, не связанные с кодами состояний HTTP, — например, исключение `ConnectionError`, соответствующее любой проблеме в сети, или `TimeoutError`, соответствующее превышению длительности тайм-аута, — могут возбуждаться и возбуждаются без подобного явного вызова.)

Возможно, вы также захотите проверить HTTP-заголовки ответа. Для этого следует использовать словарь *r*`.headers` с нечувствительными к регистру строковыми ключами, указывающими на имена заголовков, общий список которых, соответствующий спецификациям HTTP, можно найти, например, в Википедии (`https://ru.wikipedia.org/wiki/Список_заголовков_HTTP`). Большинство заголовков можно безопасно проигнорировать, но иногда вы захотите их проверить. Например, с помощью вызова *r*`.headers.get('content-language')` можно проверить, указан ли в ответе используемый язык, и если это так, то предоставить пользователю возможность воспользоваться услугами службы переводов.

Как правило, для перенаправления запроса никакой проверки специального кода состояния или заголовков не требуется: по умолчанию запросы автоматически следуют перенаправлению для всех методов, за исключением `HEAD` (это поведение можно изменить, передав в запрос именованный параметр `allow_redirection`). Если вы разрешите перенаправление, то, возможно, захотите проверить значение *r*`.history` — списка экземпляров `Response`, созданных во время запроса, в порядке

очередности от более ранних до более поздних ответов, включая текущий экземпляр *r* (поэтому, в частности, в отсутствие перенаправлений список `r.history` будет пуст).

Чаще всего после возможных проверок кодов состояния и заголовков вы захотите использовать тело ответа. В простых случаях вы будете получать тело ответа в виде байтовой строки `r.content` или декодировать его из формата JSON с помощью вызова `r.json()` (после того как определите, что ответ закодирован именно в этом формате, воспользовавшись, например, вызовом `r.headers.get('content-type')`).

Во многих случаях вы будете получать доступ к телу ответа как к тексту (Unicode), используя свойство `r.text`. Его значение декодируется (из октетов, которые в действительности формируют тело ответа) с использованием кодеков, в отношении которых модуль `requests` пытается использовать наиболее вероятные предположения на основании проверки заголовка `content-type` и поверхностного анализа самого тела ответа. Вы сможете выяснить, какой именно кодек был использован (или будет использоваться) посредством атрибута `r.encoding`, который содержит имя кодека, зарегистрированного в модуле `codecs` (рассмотрен в разделе "Модуль `codecs`" главы 8). Вы даже можете переопределить выбор используемого кодека, присвоив атрибуту `r.encoding` имя нового кодека.

Более сложные темы, такие как потоковая передача данных, в данной книге не рассматриваются. Если вы заинтересованы в получении дополнительной информации, обратитесь к онлайн-документации модуля `requests` (`http://docs.python-requests.org/en/master/user/quickstart/#raw-response-content`).

Пакет `urllib` (v3)

В дополнение к модулю `urllib.parse` (см. раздел "Модули `urllib.parse` (v3) и `urlparse` (v2)") пакет `urllib` в версии v3 предоставляет следующие модули: `urllib.robotparser`, который специально предназначен для парсинга файла *robots.txt*, документированного в общеизвестном неофициальном стандарте `http://www.robotstxt.org/orig.html` (в версии v2 используйте для этого модуль `robotparser` стандартной библиотеки; `https://docs.python.org/2/library/robotparser.html`); `urllib.error`, который содержит все типы исключений, возбуждаемых другими модулями `urllib`; а также, что наиболее важно, модуль `urllib.request`, предназначенный для открытия и чтения интернет-ресурсов.

Предоставляемая модулем `urllib.request` версии v3 функциональность повторяет функциональность модуля `urrlib2` версии v2 (раздел "Модуль `urllib2` (v2)") и дополняет ее частью функциональности модуля `urllib` версии v2 (раздел "Модуль `urllib` (v2)"). Для более подробного ознакомления с модулем `urllib.request` обратитесь к онлайн-документации (`https://docs.python.org/3/library/urllib.request.html#module-urllib.request`) и прочтите соответствующую статью из серии *HOWTO* Майка Фурда, посвященную этой теме (`https://docs.python.org/3/howto/urllib2.html#urllib-howto`).

Модуль urllib (v2)

Модуль urllib версии v2 содержит функции, обеспечивающие чтение данных из ресурса по указанному URL-адресу. Он поддерживает следующие протоколы (схемы): *http*, *https*, *ftp*, *gopher* и *file*. Схема *file* обозначает локальный файл. Модуль urllib использует ее по умолчанию для URL-адресов, в которых схема не указана. В главе 22 приведен типичный пример применения функции urllib.urlopen для извлечения HTML- и XML-страниц, используемых далее в других примерах.

Функции

Модуль urllib версии v2 предоставляет ряд функций (табл. 19.1), из которых самой простой и наиболее часто используемой является функция urlopen.

Таблица 19.1. Функции модуля urllib

Функция	Описание

quote	quote(*str*, *safe*='/')
	Возвращает копию строки *str*, в которой специальные символы преобразуются с использованием стандартных для Интернета экранированных последовательностей вида %xx. Буквенно-цифровые символы, символы _, . - (подчеркивание, запятая, точка, дефис), а также любой из символов, указанных в строке *safe*, не экранируются.
	`print(urllib.quote('zip&zap'))`
	`# вывод: zip%26zap`

quote_plus	quote_plus(*str*, *safe*='/') Аналогична функции quote, но дополнительно заменяет пробелы знаками +

unquote	unquote(*str*)
	Возвращает копию строки *str*, в которой каждая экранированная форма %xx заменяется соответствующим символом.
	`print(urllib.unquote('zip%26zap'))`
	`# вывод: zip&zap`

unquote_plus	unquote_plus(*str*) Аналогична функции unquote, но дополнительно заменяет знаки + символами пробела
urlcleanup	urlcleanup() Очищает кеш функции urlretrieve, описанной ниже
urlencode	urlencode(*query*, *doseq*=False) Возвращает аргумент *query* в URL-кодированной форме. Аргумент query может быть либо последовательностью пар (*name*, *value*), либо отображением, и в этом случае результирующая строка кодирует пары (*key*, *value*) отображения.

Функция	Описание

	`urllib.urlencode([('ans',42),('key','val')])`
	`# результат: 'ans=42&key=val'`
	`urllib.urlencode({'ans':42, 'key':'val'})`
	`# результат: 'key=val&ans=42'`
	Порядок следования элементов в словаре произволен. Если вам нужно, чтобы пары "ключ-значение" в URL-кодированной форме следовали в определенном порядке, используйте последовательность в качестве аргумента *query*, как это сделано в первом из вызовов в приведенном выше фрагменте кода.
	Если аргумент `doseq` равен `True`, то любое значение *value* в *query*, которое является последовательностью и не является строкой, кодируется в виде отдельных параметров, по одному на каждый элемент в *value*.
	`urllib.urlencode([('K',('x','y','z'))], True)`
	`# результат: 'K=x&K=y&K=z'`
	Если аргумент *doseq* равен `False` (значение по умолчанию), то каждое значение кодируется так, как если бы функция `quote_plus` была применена к его строковой форме, предоставляемой встроенной функцией `str`, независимо от того, является ли значение последовательностью:
	`urllib.urlencode([('K',('x','y','z'))], False)`
	`# результат: 'K=%28%27x%27%2C+%27y%27%2C+%27z%27%29'`

urlopen	`urlopen(`*urlstring*`, `*data*`=None, `*proxies*`=None)` Получает доступ к URL-адресу и возвращает доступный только для чтения файловый объект *f*. Объект *f* предоставляет файловые методы `read`, `readline`, `readlines` и `close`, а также два других метода, описанных ниже. *f*`.geturl()` Возвращает URL-адрес объекта *f*. Результат может отличаться от аргумента *urlstring* вследствие нормализации (о которой упоминалось ранее при описании функции `urlunsplit`) и перенаправлений HTTP (т.е. указаний на то, что запрашиваемые данные находятся в другом месте). Модуль `urllib` поддерживает перенаправления прозрачно, и в случае необходимости вы сможете проверить их с помощью метода `geturl`. *f*`.info()` Возвращает экземпляр *m* класса `Message` из модуля `mimetools` (раздел "Модули `rfc822` и `mimetools` (v2)" в главе 21). Заголовки *m* предоставляют информацию об экземпляре *f*. Например, `'Content-Type'` — это MIME-тип данных, содержащихся в *f*, и методы *m*`.gettype()`, *m*`.getmaintype()` и *m*`.getsubtype()` выдают ту же информацию.

Функция	Описание

	Если аргумент *data* равен None и в *urlstring* указана схема *http*, то функция urlopen посылает GET-запрос. Если аргумент *data* не равен None, то в *urlstring* должна быть указана схема *http*, и функция urlopen посылает POST-запрос. В этом случае аргумент *data* должен быть задан в URL-кодированной форме, в которую его обычно преобразуют с помощью рассмотренной ранее функции urlencode.
	
	Функция urlopen может использовать прокси-сервера, не требующие аутентификации. Чтобы воспользоваться этой возможностью, задайте в переменных среды http_proxy, ftp_proxy и gopher_proxy URL-адреса прокси-сервера. Обычно вы будете делать это платформозависимым способом, используя системные настройки окружения перед запуском Python. Только на компьютерах macOS функция urlopen неявно извлекает URL-адреса прокси-сервера из конфигурационных параметров Интернета прозрачным способом. Другая возможность состоит в том, чтобы передать в качестве аргумента proxies отображение, ключами которого являются названия схем, а соответствующими значениями — URL-адреса прокси-сервера.
	
	`f = urllib.urlopen('http://python.org',`
	`proxies={'http':'http://prox:999'})`
	
	Функция urlopen не поддерживает прокси-сервера, требующие аутентификации. Для этого следует использовать обладающий более широкими возможностями модуль urllib2 (раздел "Модуль urllib2 (v2)")

urlretrieve	urlretrieve(*urlstring*, *filename*=None, *reporthook*=None, *data*=None)
	
	Аналогична функции urlopen(*urlstring*, *data*), но предназначена для загрузки больших файлов. Возвращает пару (*f*, *m*), где *f* — строка, содержащая путь к файлу в локальной файловой системе, а *m* — экземпляр класса Message модуля mimetools, аналогичный результату вызова метода info для результирующего значения вызова функции urlopen, описанной выше.
	
	Если аргумент *filename* равен None, функция urlretrieve копирует извлеченные данные во временный локальный файл, а *f* — путь к этому файлу. Если аргумент *filename* не равен None, функция urlretrieve копирует извлеченные данные в файл с именем *filename*, а *f* равен *filename*. Если аргумент *reporthook* не равен None, то он должен быть вызываемым объектом с тремя аргументами, как в следующей функции:
	
	`def reporthook(block_count, block_size, file_size):`
	`  print(block_count)`
	
	Функция urlretrieve вызывает функцию *reporthook* нуль или несколько раз в процессе извлечения данных, передавая ей при каждом вызове следующие аргументы: *block_count* — количество блоков данных, извлеченных на данный момент, *block_size* — размер каждого блока (в байтах) и *file_size* — общий размер файла (в байтах). Функция

Функция	Описание
	`urlretrieve` передает `-1` в качестве значения аргумента *`file_size`*, если она не может определить размер файла, что зависит от используемого протокола и от того, насколько полно сервер реализует этот протокол. Функция *`reporthook`* позволяет вашей программе использовать графический или текстовый интерфейс для того, чтобы информировать пользователя о ходе выполнения текущей операции извлечения файла

Класс `FancyURLopener`

Обычно вы будете использовать модуль `urllib` в версии v2 посредством предоставляемых им функций (чаще всего — `urlopen`). Однако вы можете адаптировать функциональность `urllib` к своим потребностям, создав подкласс класса `FancyURLopener` модуля `urllib`, и связать экземпляр вашего класса с атрибутом `_urlopener` модуля `urllib`. Настраиваемые атрибуты экземпляра *`f`* подкласса `FancyURLopener` описаны ниже.

`prompt_user_passwd` *`f`*`.prompt_user_passwd(`*`host, realm`*`)`

Возвращает пару (*`user, password`*), которую модуль `urllib` должен использовать для аутентификации доступа к хосту в безопасной среде. Вариант, используемый по умолчанию в классе `FancyURLopener`, предлагает пользователю ввести эти данные в интерактивном текстовом режиме. Ваш подкласс может переопределить этот метод для взаимодействия с пользователем посредством графического интерфейса пользователя (GUI) или для извлечения данных аутентификации из постоянного хранилища.

`version` `f.version`

Строка, которую экземпляр *`f`* использует для идентификации самого себя на сервере, — например, посредством заголовка `User-Agent` протокола HTTP. Вы сможете переопределить этот атрибут, создав подкласс или же непосредственно связав его в экземпляре `FancyURLopener`.

Модуль `urllib2` (v2)

Модуль `urllib2` представляет собой супермножество функциональности модуля `urllib` с гибкими настройками. Он обеспечивает возможность непосредственной работы с более сложными аспектами таких протоколов, как HTTP. Например, вы можете послать запрос с модифицированными заголовками или POST-запрос с URL-кодированным телом, а также обрабатывать аутентификацию в различных контекстах в обеих ее формах, Basic и Digest, непосредственно или посредством HTTP-прокси.

В оставшейся части этого раздела рассматриваются лишь те возможности модуля `urllib2` версии v2, которые позволяют вашей программе настраивать указанные дополнительные аспекты извлечения данных удаленных ресурсов, заданных своими

URL-адресами. Мы не будем обсуждать вопросы, требующие углубленного знания HTTP и других сетевых протоколов, которые не относятся непосредственно к Python, но которые вам обязательно придется изучить, если вы захотите в полной мере использовать ту богатую функциональность, которую предлагает модуль `urllib2`.

Функции

Модуль `urllib2` предоставляет функцию `urlopen`, которая в целом аналогична функции `urlopen` модуля `urllib`. Чтобы настроить модуль `urllib2`, следует еще до вызова функции `urlopen` установить требуемое количество обработчиков, сгруппированное в *открывающий объект*, используя функции `build_opener` и `install_opener`.

Вместо URL-строки можно передать функции `urlopen` экземпляр класса `Request`. Этот экземпляр может включать как URL-строку, так и дополнительную информацию о способе доступа к ресурсу, о чем рассказано в разделе "Класс `Request`".

`build_opener` `build_opener(*handlers)`

Создает и возвращает экземпляр класса `OpenerDirector` (раздел "Класс `OpenerDirector`") с заданными обработчиками *`handlers`*. Каждый обработчик может быть подклассом класса `BaseHandler`, инстанциализируемым без аргументов, или экземпляром такого подкласса, инстанциализируемого любым способом. Функция `build_opener` добавляет экземпляры различных классов обработчиков, предоставляемых модулем `urllib2`, перед обработчиками, определенными вами, располагая их в следующем порядке: прокси-обработчики (если обнаружены настройки прокси-сервера); обработчики неизвестных схем; обработчики схем *http*, *file* и *https*; обработчики ошибок HTTP и обработчики перенаправлений HTTP. Однако, если вы передаете в *`handlers`* экземпляры или подклассы указанных классов, то это указывает на то, что вы переопределяете обработчики, заданные по умолчанию.

`install_opener` `install_opener(opener)`

Устанавливает *`opener`* в качестве открывающего объекта для последующих вызовов функции `urlopen`. Аргумент *`opener`* может быть экземпляром класса `OpenerDirector`, таким, как результат вызова функции `build_opener`, или же любым объектом с совместимой сигнатурой.

`urlopen` `urlopen(url, data=None)`

Почти идентична функции `urlopen` модуля `urllib`. Однако вы можете настроить ее поведение посредством классов `opener` и `handler` модуля `urllib2` (разделы "Класс `OpenerDirector`" и "Классы обработчиков"), а не класса `FancyURLopener`, как в случае модуля `urllib`. Аргумент *`url`* может быть URL-строкой, как в случае функции `urlopen` модуля `urllib`. Также возможно, чтобы аргумент *`url`* был экземпляром класса `Request`, рассмотренного в следующем разделе.

Класс Request

Функции urlopen можно передать в качестве аргумента не URL-строку, а экземпляр класса Request. В такой экземпляр может быть встроена как URL-строка, так и, по вашему выбору, другая информация о том, как должен осуществляться доступ к ресурсу по целевому URL-адресу.

Request

`class Request(urlstring, data=None, headers={})`

Аргумент *urlstring* — это строка URL-адреса, встроенная в данный экземпляр класса Request. Например, в отсутствие аргументов *data* и *headers* вызов

`urllib2.urlopen(urllib2.Request(urlstring))`

в точности аналогичен вызову

`urllib2.urlopen(urlstring)`

Если аргумент *data* не равен None, то конструктор Request неявно вызывает для нового экземпляра *r* его метод `r.add_data(data)`. Аргумент *headers* должен быть отображением имен заголовков в значения заголовков. Конструктор Request выполняет эквивалент следующего цикла:

`for k,v in headers.items(): r.add_header(k,v)`

Конструктор Request также принимает необязательные параметры, позволяющие, например, настраивать поведение cookie-файлов HTTP, но необходимость в такой дополнительной функциональности возникает лишь в редких случаях — обработки cookie-файлов, заданной в классе по умолчанию, обычно вполне достаточно. Относительно тонкой настройки параметров cookie-файлов на стороне клиента обратитесь к онлайн-документации (https://docs.python.org/2/library/cookielib.html). Модуль cookielib стандартной библиотеки в данной книге не рассматривается. Экземпляр *r* класса Request предоставляет методы, описанные ниже.

add_data

`r.add_data(data)`

Задает аргумент *data* в качестве данных экземпляра *r*. После этого вызов `urlopen(r)` становится аналогичным вызову `urlopen(r, data)`, т.е. он требует, чтобы схемой экземпляра *r* была *http*, и использует POST-запрос с телом *data*, который должен быть URL-кодированной строкой.

Несмотря на свое название, метод add_data не обязательно *добавляет* данные. Если в *r* уже имеются данные, установленные конструктором *r* или предыдущими вызовами метода `r.add_data`, то самый последний вызов `r.add_data` *заменяет* предыдущее значение данных *r* новым значением. В частности, вызов `r.add_data(None)` удаляет предыдущие данные *r*, если они имелись.

add_header `r.add_header(key, value)`

Добавляет заголовок с заданными ключом `key` и значением `value` в заголовки `r`. Если схемой `r` является *http*, то заголовки `r` посылаются в качестве части запроса. В случае добавления нескольких заголовков с одним и тем же ключом более поздние добавления заменяют предыдущие, поэтому из всех заголовков с данным ключом имеет значение лишь тот, который задан последним.

add_unredirected_header `r.add_unredirected_header(key, value)`

Аналогичен методу `add_header`, за исключением того, что заголовок добавляется только для первого запроса и не используется, если запрашивающая процедура принимает какие-либо дальнейшие перенаправления HTTP и следует им.

get_data `r.get_data()`

Возвращает данные экземпляра `r`, либо `None`, либо URL-кодированную строку.

get_full_url `r.get_full_url()`

Возвращает URL-адрес экземпляра `r`, предоставленный конструктору для `r`.

get_host `r.get_host()`

Возвращает компонент `host` URL-адреса экземпляра `r`.

get_method `r.get_method()`

Возвращает HTTP-метод экземпляра `r`, любую из строк `'GET'` или `'POST'`.

get_selector `r.get_selector()`

Возвращает селекторные компоненты URL-адреса экземпляра `r` (путь и все последующие компоненты).

get_type `r.get_type()`

Возвращает компонент схемы URL-адреса экземпляра `r` (т.е. протокол).

has_data `r.has_data()`

Аналогичен инструкции `r.get_data() is not None`.

has_header `r.has_header(key)`

Возвращает `True`, если экземпляр `r` имеет заголовок с заданным ключом; в противном случае возвращает `False`.

set_proxy `r.set_proxy(host, scheme)`

Настраивает экземпляр `r` для использования прокси-сервера с заданными хостом и схемой для доступа к URL-адресу экземпляра `r`.

Класс OpenerDirector

Экземпляр *d* класса OpenerDirector собирает экземпляры классов обработчиков и управляет их использованием для открытия URL-адресов различных схем и обработки ошибок. Обычно экземпляр *d* создают посредством вызова функции build_opener и затем устанавливают с помощью вызова функции install_opener. Для более сложных вариантов использования вы можете обращаться к различным атрибутам и методам *d*, но необходимость в этом возникает лишь в редких случаях, и мы не рассматриваем их в данной книге.

Классы обработчиков

Модуль urllib2 предоставляет класс BaseHandler, который может использоваться в качестве суперкласса для любых написанных вами пользовательских классов обработчиков. Он также предоставляет многочисленные конкретные подклассы класса BaseHandler, обрабатывающие схемы *gopher*, *ftp*, *http*, *https* и *file*, а также аутентификацию, прокси-запросы, перенаправления и ошибки. Написание пользовательских обработчиков urllib2 — сложная тема, и в данной книге она не рассматривается.

Обработка аутентификации

Открывающий объект, заданный по умолчанию в urllib2, не выполняет аутентификацию. Для выполнения аутентификации следует вызвать функцию build_opener, чтобы создать открывающий объект с экземплярами HTTPBasicAuthHandler, ProxyBasicAuthHandler, HTTPDigestAuthHandler и/или ProxyDigestAuthHandler, в зависимости от того, какая именно аутентификация требуется — непосредственная или через прокси, и каким должен быть ее тип — Basic или Digest.

Для создания каждого из этих обработчиков аутентификации используйте экземпляр *x* класса HTTPPasswordMgrWithDefaultRealm в качестве единственного аргумента в конструкторе обработчика. Обычно для инстанциализации всех необходимых обработчиков аутентификации используют один и тот же экземпляр *x*. Чтобы записать информацию о пользователях и паролях для заданных контекстов аутентификации и URL-адресов, вызовите метод *x*.add_password один или несколько раз.

add_password *x*.add_password(*realm*, *URLs*, *user*, *password*)

Записывает в *x* пару (*user*, *password*) в качестве имени пользователя и пароля для указанного контекста *realm* и URL-адресов, заданных аргументом *URLs*. Аргумент *realm* — это строка, именующая контекст аутентификации. Аргументу *realm*, равному None, соответствует любой контекст, не указанный конкретно. Аргумент *URLs* — это строка или последовательность строк URL-адресов. URL-адрес *u* считается применимым для данных имени пользователя и пароля, если в аргументе *URLs* имеется такой элемент *u1*, что компоненты *location* для *u* и *u1* совпадают, а компонент *path* элемента *u1* является префиксом в таком же компоненте *u*. Другие компоненты (*scheme*, *query*, *fragment*) не влияют на применимость URL-адресов для целей аутентификации. Вот пример использования модуля urllib2 с базовой HTTP-аутентификацией.

```
import urllib2

x = urllib2.HTTPPasswordMgrWithDefaultRealm( )
x.add_password(None, 'http://myhost.com/',
'auser', 'apassword')
auth = urllib2.HTTPBasicAuthHandler(x)
opener = urllib2.build_opener(auth)
urllib2.install_opener(opener)

flob = urllib2.urlopen('http://myhost.com/index.html')
for line in flob.readlines(): print line,
```

Другие сетевые протоколы

Количество используемых сетевых протоколов *огромно*, и некоторые из них лучше всего поддерживаются стандартной библиотекой Python, но для подавляющего числа протоколов имеет смысл поиск соответствующих сторонних модулей на сайте PyPI (`https://pypi.python.org/pypi`).

Чтобы подключиться к серверу так, как если бы вы осуществляли вход на другой компьютер (или в отдельном сеансе на собственном узле), используйте защищенный протокол SSH (`https://en.wikipedia.org/wiki/Secure_Shell`), поддерживаемый сторонним модулем `paramiko` (`http://www.paramiko.org/`) или его высокоуровневой оберткой — сторонним модулем `spur` (`https://github.com/mwilliamson/spur.py`). Допуская определенную степень риска в плане безопасности, вы все еще можете использовать классический Telnet (`https://ru.wikipedia.org/wiki/Telnet`), поддерживаемый модулем `telnetlib` стандартной библиотеки (`https://docs.python.org/3/library/telnetlib.html`).

Среди других протоколов назовем следующие.

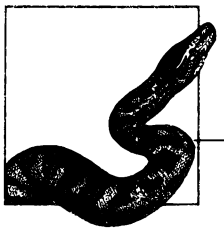
- NNTP (`https://ru.wikipedia.org/wiki/NNTP`). Предназначен для несколько устаревших серверов Usenet News, поддерживаемых модулем стандартной библиотеки `nntplib` (`https://docs.python.org/3/library/nntplib.html`).
- XML-RPC (`https://ru.wikipedia.org/wiki/XML-RPC`). Предоставляет простейшую функциональность удаленного вызова процедур, поддерживаемую модулем `xmlrpc.client` (`https://docs.python.org/3/library/xmlrpc.client.html`) или `xmlrpc` (`https://docs.python.org/2/library/xmlrpclib.html`) в версии v2.
- gRPC (`https://grpc.io/`). Предоставляет современную расширенную функциональность удаленного вызова процедур, поддерживаемую сторонним модулем `grpcio` (`https://pypi.python.org/pypi/grpcio`).
- NTP (`http://www.ntp.org/`). Предназначен для получения точного сетевого времени; поддерживается сторонним модулем `ntplib` (`https://pypi.python.org/pypi/ntplib/`).

- SNMP (`https://ru.wikipedia.org/wiki/SNMP`). Предназначен для управления устройствами в IP-сетях; поддерживается сторонним модулем `pysnmp` (`https://pypi.python.org/pypi/pysnmp`).

Этот список можно было бы продолжать! Подробно описать все эти протоколы в одной книге просто не представляется возможным. Поэтому рекомендуем вам использовать следующий стратегический подход: всякий раз, когда вы приступаете к написанию приложения, нуждающегося во взаимодействии с другими системами посредством определенного сетевого протокола, не бросайтесь сразу же реализовать собственные модули для его поддержки. Осмотритесь вокруг, проведите соответствующий поиск, и вам наверняка удастся подыскать уже существующие модули Python, отлично поддерживающие данный протокол[3].

Если вы обнаружите какую-то ошибку в этих модулях или решите, что в них не хватает каких-то средств, оповестите сообщество об обнаруженной ошибке или запросите требуемое средство (в идеальном случае предложите свое решение или найдите рекомендации по устранению проблемы, которые удовлетворили бы потребности вашего приложения). Иными словами, станьте активным членом сообщества свободного программного обеспечения, а не просто пассивным пользователем: вас с радостью там примут, и вам представится возможность не только получать помощь от других людей, но и самому помогать им.

[3] Еще более важно следующее: если вы считаете, что вам придется изобрести совершенно новый протокол и реализовать его поверх сокетов, дважды подумайте, прежде чем решиться на это, и сначала проведите соответствующий поиск — с большой долей вероятности один (или несколько) из огромного количества существующих интернет-протоколов окажется именно тем, что вам нужно!



20

Работа с протоколом HTTP

Когда браузер (или любой другой веб-клиент) запрашивает у сервера страницу, сервер может вернуть как статическое, так и динамическое содержимое. Обслуживание динамического содержимого неизменно связано с тем, чтобы веб-программы на стороне сервера генерировали и доставляли содержимое на лету, зачастую на основе информации, хранящейся в базе данных.

На заре веб-разработки стандартом программирования на стороне сервера служил CGI (Common Gateway Interface — общий интерфейс шлюза), который требовал запуска на сервере отдельной программы всякий раз, когда клиент запрашивал динамическое содержимое. Суммарное время, необходимое для запуска процесса, инициализации интерпретатора, подключения к базе данных и инициализации скриптов, достигает ощутимой величины. Поэтому CGI не мог обеспечить надлежащую масштабируемость приложений.

В наши дни веб-серверы поддерживают множество специфических для сервера способов снижения накладных расходов, обрабатывая динамическое содержимое из процессов, способных обслуживать несколько посещений страницы, а не запускать каждый раз новый CGI-процесс. Если вам приходится поддерживать существующие CGI-программы или портировать их на более современные платформы, обратитесь к онлайн-документации модулей `cgi` (`https://docs.python.org/3/library/cgi.html`) и `http.cookies` (`https://docs.python.org/3/library/http.cookies.html`) стандартной библиотеки[1].

С появлением систем, основанных на использовании микросервисов, протокол HTTP стал играть еще более фундаментальную роль в проектировании распределенных систем, поскольку он предлагает удобный способ транспортировки

[1] Одно из исторических наследий CGI связано с тем, что сервер предоставлял CGI-скрипту информацию о текущем HTTP-запросе посредством окружения операционной системы (в Python — `os.environ`). По сегодняшний день интерфейсы между веб-серверами и фреймворками приложений полагаются на "окружение", которое является по сути словарем, обобщая и развивая ту же фундаментальную идею.

JSON-содержимого между процессами, что требуется довольно часто. В настоящее время существуют тысячи общедоступных программных интерфейсов для передачи данных в Интернете посредством HTTP. И хотя базовые принципы HTTP-протокола остаются почти неизменными с момента его появления в середине 1990-х годов, его возможности были значительно расширены за прошедшие годы. Более подробную информацию вместе с отлично подобранными ссылками на справочные материалы по этой теме можно найти в книге *HTTP: The Definitive Guide* (`http://shop.oreilly.com/product/9781565925090.do`).

WSGI

WSGI (Web Server Gateway Interface — интерфейс шлюза веб-сервера) — это стандарт, по которому все современные фреймворки для разработки приложений на языке Python взаимодействуют с веб-серверами или шлюзами. При этом вовсе не предполагается, что программы вашего приложения взаимодействуют непосредственно с WSGI. Скорее, код ваших программ использует любой из фреймворков, работающих на высоком уровне абстракции, а фреймворк, в свою очередь, использует WSGI для общения с сервером.

О деталях WSGI (`https://www.python.org/dev/peps/pep-3333/`) вам придется заботиться лишь в том случае, если вы реализуете интерфейс WSGI для веб-сервера, который не предоставляет его (неужели такие еще существуют?!), или создаете новый фреймворк для Python[2]. Если так оно и есть, изучите документ **PEP 3333** (`https://www.python.org/dev/peps/pep-3333/`), содержащий спецификацию WSGI, а также документацию пакета `wsgiref` стандартной библиотеки (`https://docs.python.org/3/library/wsgiref.html`) и соответствующий материал на сайте `wsgi.org` (`http://wsgi.readthedocs.io/en/latest/`).

Некоторые концепции WSGI могут быть важными для вас, если вы используете облегченные фреймворки (т.е. те, которые обеспечивают достаточно близкое соответствие WSGI). WSGI — это *интерфейс*, и, как у любого интерфейса, у него имеются две стороны: сторона *веб-сервера/шлюза* и сторона *приложения/фреймворка*.

Задача стороны фреймворка заключается в следующем:

- во-первых, предоставить WSGI объект *приложения*, в качестве которого может выступать любой вызываемый объект (часто им может быть экземпляр класса, имеющего специальный метод `__call__`, но это является деталями

[2] Прошу вас отказаться от этой затеи. Как однажды справедливо заметил Тайтус Браун, Python печально известен тем, что количество фреймворков для него превышает количество имеющихся в нем ключевых слов. Один из авторов этой книги как-то посоветовал Гвидо, когда тот только приступил к работе над проектом Python 3, каким образом можно легко устранить это неравенство: для этого достаточно ввести в язык еще несколько сотен ключевых слов. Однако Гвидо по каким-то причинам не был в восторге от такого предложения.

реализации), соблюдающий соглашения, установленные в соответствующем документе PEP (`http://wsgi.readthedocs.io/en/latest/`);

- во-вторых, подключить объект приложения к серверу, используя для этого любые средства, документированные для данного сервера (зачастую это всего лишь несколько строк кода, конфигурационные файлы или просто соглашение относительно, например, того, что объект приложения WSGI должен быть указан в качестве атрибута `application` на верхнем уровне модуля).

Сервер вызывает объект приложения для каждого входящего HTTP-запроса, на что объект приложения реагирует соответствующим образом, чтобы сервер мог сформировать исходящий HTTP-ответ и отправить его клиенту, причем все это должно делаться с соблюдением вышеуказанных соглашений. Фреймворк, даже облегченный, избавляет вас от необходимости знания подобных деталей (за исключением того, что вам, возможно, придется создать и подключить экземпляр объекта приложения, в зависимости от специфики конкретного сервера).

Серверы WSGI

Обширный список серверов и адаптеров, которые могут быть использованы для выполнения фреймворков и приложений WSGI (как в целях разработки и тестирования, так и в производственных условиях) приведен в Интернете (`http://wsgi.readthedocs.io/en/latest/servers.html`), но и он не является полным. Например, в нем отсутствуют сведения о том, что среда выполнения Python, предлагаемая службой Google App Engine (`https://cloud.google.com/appengine/docs/python/`), также является WSGI-сервером, готовым к диспетчеризации WSGI-приложений в соответствии с параметрами конфигурационного файла *app.yaml*.

Если вы ищете WSGI-сервер для разработки или развертывания программ в производственных условиях совместно, скажем, с балансировщиком нагрузки на основе сервера `nginx` (`nginx.org`), то, по крайней мере в случае Unix-подобных систем, будете удовлетворены WSGI-сервером Green Unicorn (`http://docs.gunicorn.org/en/stable/`): предназначенный исключительно для Python, он отличается легковесностью и не поддерживает ничего, кроме WSGI. Его достойной альтернативой является WSGI-сервер Waitress (`https://docs.pylonsproject.org/projects/waitress/en/latest/`), также предназначенный исключительно для Python и предоставляющей только WSGI. В настоящее время он обеспечивает лучшую поддержку в Windows. Если вам нужна еще более широкая функциональность (например, поддержка Perl и Ruby наряду с Python, а также многие другие формы расширяемости), обратите внимание на более мощный и сложный сервер `uWSGI` (`https://uwsgi-docs.readthedocs.io/en/latest/`)[3].

[3] В настоящее время установка *uWSGI* в Windows требует компиляции с помощью Cygwin (*www.cygwin.com*).

В WSGI также существует понятие *промежуточного (связующего) программного обеспечения* (middleware), относящееся к подсистемам, которые реализуют как сторону сервера, так и приложения WSGI. Промежуточный объект "обертывает" WSGI-приложение. Он может селективно изменять запросы, параметры окружения и ответы, представляя себя серверу как "приложение". Допускается и широко распространено использование многослойных оберток, которые образуют "стек" промежуточных объектов, предлагающих услуги на уровне кода фактического приложения. Если вы захотите написать компонент промежуточного программного обеспечения, пригодный для использования несколькими фреймворками, то для этого вам необходимо стать настоящим экспертом в области WSGI.

Веб-фреймворки Python

Обзор большинства веб-фреймворков Python можно найти на вики-странице Python (`https://wiki.python.org/moin/WebFrameworks`). Это авторитетный источник, находящийся на официальном сайте `python.org` и курируемый сообществом, поэтому он будет постоянно обновляться. На момент выхода книги в этом списке числились 56 фреймворков[4], помеченных как "активные", плюс намного больше других с пометкой "разработка прекращена/неактивен". Кроме того, в списке приведены ссылки на отдельные вики-страницы, посвященные используемым в Python системам управления контентом (`https://wiki.python.org/moin/ContentManagementSystems`), веб-серверам (`https://wiki.python.org/moin/WebServers`), веб-компонентам (`https://wiki.python.org/moin/WebComponents`) и их библиотекам.

Полностековые и облегченные фреймворки

Веб-фреймворки Python можно грубо разделить на две основные категории: *полностековые*, пытающиеся предоставить всю функциональность, которая может потребоваться вам для создания веб-приложений, и *облегченные*, предоставляющие лишь тот минимум, который необходим вам в качестве удобного интерфейса к самому веб-серверу, и позволяющие самостоятельно выбирать отдельные, наиболее предпочтительные для вас компоненты для таких задач, как создание интерфейсов к базам данных (см. раздел "Python Database API (DBAPI) 2.0" в главе 11) и работа с шаблонами (раздел "Работа с шаблонами" в главе 22), а также других задач, охватываемых веб-компонентами, о которых пойдет речь в следующем разделе. Разумеется, эта таксономия, как и любая другая, не является ни точной, ни полной и требует здравых оценок, однако это первый шаг к тому, чтобы не заблудиться среди многих десятков веб-фреймворков Python.

[4] С учетом того, что в Python насчитывается немногим более 30 ключевых слов, смысл замечания Тайтуса Брауна о том, что Python имеет больше фреймворков, чем ключевых слов, сразу же становится очевидным.

Подробное рассмотрение полностековых фреймворков в рамках данной книги просто невозможно — каждый из них имеет слишком объемную структуру (иногда весьма разветвленную). Тем не менее какой-то один из них может оказаться наилучшим для конкретной специфики ваших приложений, поэтому мы упомянем о тех из них, которые пользуются наибольшей популярностью, и рекомендуем вам посетить сайты, ссылки на которые мы предоставим.

Когда следует использовать облегченные фреймворки

Как следует из самого определения "облегченный", если вам необходимо обращаться к базам данных, использовать шаблоны или другую функциональность, не имеющую прямого отношения к HTTP, то для этой цели вы будете подбирать отдельные компоненты (работе с базами данных и шаблонами посвящены другие главы). Однако, чем более упрощен ваш фреймворк, тем большее количество отдельных компонентов вам потребуется для достижения нужной функциональности, такой как аутентификация пользователей или сохранение состояния в промежутках между запросами данного пользователя. Многие пакеты промежуточного программного обеспечения WSGI (`http://wsgi.readthedocs.io/en/latest/libraries.html`) могут оказать вам помощь в решении подобных задач. Некоторые отличные пакеты довольно специализированы. Так, пакет `Barrel` (`https://pypi.python.org/pypi/barrel`) предназначен для управления доступом, пакет `Beaker` (`http://beaker.readthedocs.io/en/latest/`) — для сохранения состояния в форме упрощенных сеансов нескольких видов и т.п.

В то же время, если нам (авторам этой книги) требуется хороший пакет промежуточного программного обеспечения WSGI для какой-либо цели, мы практически всегда сначала обращаемся к библиотеке Werkzeug (`http://werkzeug.pocoo.org/`) — коллекции подобных компонентов, поражающей своим размахом и качеством. Мы не обсуждаем библиотеку Werkzeug в этой книге (по тем же причинам, по которым не рассматриваем другое промежуточное программное обеспечение), но настоятельно рекомендуем ознакомиться с ней.

Популярные полностековые фреймворки

До сих пор наиболее популярным полностековым фреймворком является Django (`https://www.djangoproject.com/`), отличающийся большими размерами и расширяемостью. Так называемые *приложения* Django в действительности являются повторно используемыми подсистемами, тогда как то, что обычно называют "приложением", в Django называется *проектом*. Django навязывает определенный способ мышления, но взамен предлагает мощные возможности и широкую функциональность.

Отличный альтернативный полностековый вариант — фреймворк web2py (`http://www.web2py.com/`), который предлагает примерно те же возможности, но проще в изучении и известен тем, что в нем предпринимаются все возможные меры для обеспечения обратной совместимости (если поддержка списка исправлений

и обновлений этого фреймворка будет продолжаться, то любое web2py-приложение, которое вы напишете сегодня, будет работать в будущем бесконечно долго). Качество подготовки его документации заслуживает отдельной похвалы[5].

Третьим достойным соперником является фреймворк TurboGears, который стартует как облегченный фреймворк, но достигает статуса "полностекового" путем интеграции других, независимых сторонних проектов, предоставляющих другие виды функциональности, необходимые для большинства веб-приложений, такие как интерфейсы для доступа к базам данных или средства для работы с шаблонами, чтобы вам не приходилось писать их заново. Другим, близким ему по духу "облегченным, но функционально насыщенным" фреймворком является Pyramid (`trypyramid.com`).

Некоторые популярные облегченные фреймворки

Как уже отмечалось, для Python разработано множество фреймворков, включая многочисленные облегченные фреймворки. Мы рассмотрим четыре фреймворка этой категории: Falcon, Flask, Bottle и webapp2. Обсуждая каждый из них, мы предоставляем пример крошечного приложения, которое приветствует пользователей и напоминает им о дате последнего посещения сайта, используя значения, сохраненные в cookie-файле `lastvisit`. Сначала мы покажем, как установить cookie-файл в основной функции "приветствия". Однако для приложения было бы желательно, чтобы cookie-файл `lastvisit` устанавливался независимо от способа "посещения" приложения. Рассмотренные ниже фреймворки позволяют не повторять установку cookie-файла в каждом используемом методе, а факторизовать код в строгом соответствии с принципом DRY (Don't Repeat Yourself — не повторяйся). В связи с этим мы также приводим пример того, как факторизовать соответствующий код в каждом из этих веб-фреймворков.

Falcon

Возможно, самым быстрым (а отсюда можно сделать вывод, что и самым легковесным) из фреймворков Python является Falcon. Согласно опубликованным на его сайте результатам оценочных тестов он работает в обеих версиях Python, v2 и v3, от 5 до 8 раз быстрее, чем Flask — самый популярный облегченный веб-фреймворк Python. Фреймворк Falcon отличается высокой переносимостью, обеспечивая дополнительное небольшое ускорение за счет поддержки Jython или Cython и огромный скачок в быстродействии — в целом в 27 раз превышающем быстродействие Flask — за счет поддержки PyPy. Кроме сайта проекта (`falconframework.org`), ознакомьтесь с источниками на сайте GitHub (`https://github.com/falconry/falcon`), а также с рекомендациями по установке на сайте PyPI (`https://pypi.python.org/pypi/falcon`) и обширной документацией (`http://falcon.readthedocs.io/en/latest/`).

[5] С различными точками зрения относительно достоинств и недостатков веб-фреймворка web2py можно ознакомиться на сайте Quora (`https://www.quora.com/Is-web2py-a-good-Python-web-framework`).

Чтобы запустить Falcon в Google App Engine локально на своем компьютере (или на серверах Google на сайте appspot.com), посетите полезную страницу Рафаэля Баррело на сайте GitHub (https://github.com/rafaelbarrelo/appengine-falcon-skeleton).

Основной класс, предоставляемый пакетом falcon, называется API. Экземпляр falcon.API является WSGI-приложением, поэтому распространенной идиомой является многократное присваивание.

```
import falcon

api = application = falcon.API()
```

При инстанциализации falcon.API можно передать нуль или несколько именованных параметров, описанных ниже.

media_type
: Тип содержимого, используемого в ответах, если не указано иное. По умолчанию используется значение 'application/json; charset=utf-8'.

middleware
: Объект *промежуточного программного обеспечения Falcon* или список таких объектов (раздел "Промежуточное программное обеспечение Falcon").

request_type
: Класс, используемый в запросах. По умолчанию это класс falcon.request.Request, но вы можете написать код его подкласса и использовать полученный подкласс.

response_type
: Класс, используемый в ответах. По умолчанию это класс falcon.request.Response, но вы можете написать код его подкласса и использовать полученный подкласс.

router
: Класс, используемый для маршрутизации. По умолчанию это класс falcon.routing.CompiledRouter, непригодный для наследования, но вы можете написать код класса собственного маршрутизатора, обеспечив его совместимость посредством неявной ("утиной") типизации, и использовать его.

Получив объект api, вы можете адаптировать его для своих потребностей, настроив, скажем, параметры обработки ошибок или добавив приемники данных (используя регулярные выражения для перехвата и диспетчеризации набора маршрутов, например играя роль прокси-сервера для доступа к другому сайту). Эти дополнительные темы (а также пользовательские типы запроса, ответа и маршрутизатора) в данной книге не рассматриваются: в документации Falcon (http://falcon.readthedocs.io/en/latest/index.html; в частности, в виде справочного материала в разделе *Classes and Functions*) все эти вопросы очень хорошо освещены.

Описанный ниже метод `add_route` относится к числу методов, которые вы почти неизбежно будете вызывать для объекта `api`, причем, как правило, неоднократно. Он необходим для добавления маршрутов, а значит, и для получения обслуживаемых запросов.

`add_route` `add_route(uri_template, resource)`

Аргумент *`uri_template`* — это строка URI, по которому следует направить запрос. Например, строка `'/'` соответствует маршрутизации запросов в корневой URL сайта. Аргумент *`template`* может включать одно или несколько полей выражений — идентификаторов, заключенных в фигурные скобки (`{}`). В этом случае *методы-ответчики* ресурса *`resource`* должны поддерживать аргументы с именами этих идентификаторов, указанными в том же порядке после обязательных аргументов для запроса и ответа.

Аргумент *`resource`* — это экземпляр *ресурса Falcon* (раздел "Ресурсы Falcon"). Обратите внимание на то, что передается конкретный экземпляр, используемый для обслуживания всех запросов по данному URI, а не сам класс. Будьте предельно внимательны при написании кода, особенно в отношении использования переменных экземпляра.

Ресурсы Falcon

Класс ресурсов Falcon — это класс, предоставляющий так называемые *методы-ответчики* (responder methods), чьи имена начинаются с префикса `on_`, за которым следует глагол HTTP в нижнем регистре. Например, метод `on_get` отвечает на самый распространенный HTTP-глагол `GET`.

Обычно каждый метод-ответчик имеет следующую сигнатуру, получая объект запроса `req` и объект ответа `resp` в качестве аргументов:

```
def on_get(self, req, resp): ...
```

Но если шаблон URI, определяющий маршрут к ресурсу, включает поля выражений, то методы-ответчики должны поддерживать дополнительные аргументы, имена которых совпадают с идентификаторами в полях выражений. Например, если запрос маршрутизируется к ресурсу посредством вызова `api.add_route('/foo/{bar}/{baz}', rsrc)`, то методы-ответчики экземпляра *`rsrc`* ресурсного класса должны иметь сигнатуры наподобие следующей:

```
def on_post(self, req, resp, bar, baz): ...
```

Ресурс не должен предоставлять методы-ответчики для всех глаголов HTTP. Если для обращения к ресурсу используется глагол HTTP, который не поддерживается данным ресурсом (например, в случае использования глагола POST в отношении маршрута, класс ресурса которого не имеет метода `on_post`), то Falcon обнаруживает это и отвечает на недействительный запрос сообщением с кодом HTTP-состояния: `'405 Method Not Allowed'`.

Falcon не предоставляет никакого класа, от которого наследовались бы классы ресурсов: вместо этого классы ресурсов целиком полагаются на *"утиную" типизацию*

(т.е. они всего лишь определяют методы-ответчики с подходящими именами и сигнатурами) для полной поддержки Falcon.

Объекты запросов Falcon

Класс `falcon.Request` предоставляет большое количество свойств и методов, подробно описанных в документации Falcon (`http://falcon.readthedocs.io/en/latest/api/request_and_response.html`). Чаще всего вы будете использовать методы, представленные в атрибуте `cookies` (`http://falcon.readthedocs.io/en/latest/api/cookies.html`) — словаре, ассоциирующем имена cookie-файлов со значениями, а также метод `get_header`, предназначенный для получения заголовка из запроса (возвращает значение `None`, если запрос не имеет такого заголовка), метод `get_param`, предназначенный для получения параметра запроса (возвращает значение `None`, если запрос не имеет такого параметра), и некоторые их вариации, такие как метод `get_param_as_int`, позволяющий получить целое значение для параметра запроса, который должен быть целочисленным.

Объекты ответов Falcon

Класс `falcon.Response` содержит большое количество свойств и методов, подробно описанных в документации Falcon (`http://falcon.readthedocs.io/en/latest/api/request_and_response.html`). Обычно вам придется устанавливать многие свойства (за исключением свойств, управляющих заголовками, отсутствие которых, как и использование значений по умолчанию, является для вас приемлемым) и вызывать многие методы. Перечень свойств класса `falcon.Response` приводится ниже.

`body`
: Присвойте этому свойству значение в виде строки Unicode, чтобы задать тело ответа в кодировке UTF-8. Если имеется тело ответа в виде байтовой строки, присвойте его свойству `data` для ускорения операций.

`cache_control`
: Присвойте этому свойству значение, чтобы установить заголовок `Cache-Control` (`https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9`). Если вы присваиваете значение в виде списка строк, Falcon объединит их, используя в качестве разделителя строку `', '`, для получения значения заголовка.

`content_location`
: Присвойте этому свойству значение, чтобы установить заголовок `Content-Location` (`https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.14`).

`content_range`
: Присвойте этому свойству значение, чтобы установить заголовок `Content-Range` (`https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.16`).

`content_type`

Присвойте этому свойству значение, чтобы установить заголовок `Content-Type` (`https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.17`). Если вы устанавливаете свойство `resp.data` для HTML-документа в кодировке ISO-8859-15, то типичным значением, которое вы присвоили бы атрибуту `resp.content_type`, могла бы, например, быть строка `'text/html; charset=ISO-8859-15'` (значение по умолчанию `api.media_type`).

`data`

Присвойте этому свойству значение в виде байтовой строки в качестве тела ответа.

`etag`

Присвойте этому свойству значение, чтобы установить заголовок `ETag`.

`last_modified`

Присвойте этому свойству значение, чтобы установить заголовок `Last-Modified`.

`location`

Присвойте этому свойству значение, чтобы установить заголовок `Location`.

`retry_after`

Присвойте этому свойству значение, чтобы установить заголовок `Retry-After`, что обычно делается в сочетании с установкой значения `'503 Service Unavailable'` для свойства `resp.status`.

`status`

Присвойте этому свойству значение строки состояния, которым по умолчанию является строка `'200 OK'`. Например, задайте значение `resp.status = '201 Created'`, если метод `on_post` создает новый объект. Подробные сведения обо всех кодах и строках состояния вы найдете в документе RFC 2616 (`https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`).

`stream`

Присвойте этому свойству значение в виде файлового объекта, метод `read` которого возвращает строку байтов или итератор, возвращающий байтовые строки, если это является для вас более удобным способом формирования тела ответа, чем присваивание значения свойству `resp.body` или `resp.data`.

`stream_len`

По желанию можете присвоить этому свойству целочисленное значение, определяющее количество байтов в `resp.stream`, если вы задаете значение последнего. Falcon использует это свойство для установки заголовка `Content-Length` (если оно не предоставлено, Falcon может использовать *блочное кодирование* или альтернативные варианты).

`vary`

Присвойте этому свойству значение, чтобы установить заголовок `Vary`.

Работая с облегченными фреймворками, вы должны четко осознавать свои действия!

Возможно, вы заметили, что установка значений многих свойств требует понимания протокола HTTP (иными словами, вы должны отдавать себе отчет в своих действиях), в то время как полностековый фреймворк пытается "провести вас за руку", чтобы вы все делали правильно, даже не понимая полностью смысла своих действий и почему они являются правильными. Впрочем, все это достигается за счет дополнительных затрат времени и ресурсов, а также принятия концептуальных основ полностекового фреймворка и изменения своего способа мышления. ("Платите деньги и берите что хотите!") Авторы данной книги — большие приверженцы облегченных фреймворков, но при этом мы понимаем, что в разработке веб-приложений не существует "царских путей", и другие разработчики могут отдать предпочтение полностековым фреймворкам с их богатыми и универсальными возможностями. Каждому свое!

Вернемся к экземплярам класса `falcon.Response`. Ниже описаны основные методы экземпляра *r* этого класса.

Таблица 20.1. Методы класса `falcon.Response`

Метод	Описание
`add_link`	*r*`.add_link(`*target*`, `*rel*`, `*title*`=None, `*title_star*`=None, `*anchor*`=None, `*hreflang*`=None, `*type_hint*`=None)` Добавляет в ответ заголовок `Link` (`https://tools.ietf.org/html/rfc5988`). Обязательными аргументами являются *target* — целевой URI ссылки (в случае необходимости Falcon преобразует этот аргумент в URI) и *rel* — строка с именем одной из многих связей ссылок (`https://www.iana.org/assignments/link-relations/link-relations.xhtml`), каталогизированных IANA, таких как `'terms-of-service'` или `'predecessor-version'`. Необходимость в других аргументах (если они передаются, то только в виде именованных аргументов) возникает лишь в редких случаях, и они подробно описаны в документации Falcon (`http://falcon.readthedocs.io/en/latest/api/request_and_response.html`) и документе RFC 5988 (`https://tools.ietf.org/html/rfc5988`). Если вы вызываете метод *r*`.add_link` многократно, то Falcon создает один заголовок `Link` с указанными вами различными ссылками, разделенными запятыми
`append_header`	*r*`.append_header(`*name*`, `*value*`)` Устанавливает или присоединяет HTTP-заголовок с заданными именем *name* и значением *value*. Если заголовок с именем *name* уже существует, то Falcon присоединяет к нему аргумент *value* через запятую. Это

Метод	Описание
	работает для большинства заголовков, но не для cookie-файлов — для них используйте метод `set_cookie`
`get_header`	`r.get_header(`*`name`*`)` Возвращает строковое значение для HTTP-заголовка с заданным именем *`name`* или значение `None`, если такой заголовок отсутствует
`set_cookie`	`r.set_cookie(`*`name`*`, `*`value`*`, `*`expires`*`=None, `*`max_age`*`=None, `*`domain`*`=None, `*`path`*`=None, `*`secure`*`=True, `*`http_only`*`=True)` Устанавливает в ответе заголовок `Set-Cookie`. Обязательными аргументами являются *`name`* (имя cookie-файла) и *`value`* в виде строки (значение cookie-файла). Другие аргументы (если они передаются, то только в виде именованных аргументов) могут потребоваться для обеспечения дополнительной безопасности (и сохранения cookie-файла в постоянном хранилище: если ни аргумент *`expires`*, ни аргумент *`max_age`* не заданы, то срок действия cookie-файла истекает по окончании сеанса браузера), и они подробно описаны в документации Falcon (`http://falcon.readthedocs.io/en/latest/api/request_and_response.html`) и документе RFC 6525 (`https://tools.ietf.org/html/rfc6265`). Если вы вызываете метод *`r.set_cookie`* многократно, то Falcon добавляет в ответ несколько заголовков `Set-Cookie`
`set_header`	`r.set_header(`*`name`*`, `*`value`*`)` Устанавливает HTTP-заголовок с заданными именем *`name`* и значением *`value`*. Если заголовок с таким именем уже существует, то он заменяется
`set_headers`	`r.set_headers(`*`headers`*`)` Устанавливает несколько HTTP-заголовков, как это было бы сделано с помощью нескольких вызовов `set_header`. Аргумент *`headers`* может быть либо итерируемым объектом, содержащим пары `(`*`name`*`, `*`value`*`)` или отображение с *`name`* в качестве ключа и *`value`* в качестве соответствующего значения
`unset_cookie`	`r.unset_cookie(`*`name`*`)` Удаляет из ответа cookie-файл с именем *`name`*, если он в нем имеется. Также добавляет в ответ заголовок `Set-Cookie`, запрашивая у браузера разрешение удалить cookie-файл из его кеша, если в нем сохранен данный cookie-файл

Промежуточное программное обеспечение Falcon

Класс промежуточного программного обеспечения Falcon предоставляет от одного до трех методов, предназначенных для проверки и, возможно, изменения входящих запросов и исходящих ответов. Список экземпляров таких классов передается

в качестве аргумента *middleware* во время инстанциализации `falcon.API`, и этим экземплярам предоставляется возможность исследовать и изменять все входящие запросы и исходящие ответы. Экземпляры получают эту возможность в порядке очередности от первого к последнему для входящих запросов и в обратном порядке, от последнего к первому, для исходящих ответов.

Falcon не предоставляет никакого базового класса для классов промежуточного ПО, от которого они могли бы наследоваться: эти классы целиком полагаются на "утиную" типизацию (т.е. просто определяют от одного до трех методов с подходящим именем и сигнатурой). Если класс промежуточного ПО не предоставляет эти методы, то Falcon трактует свои экземпляры, как если бы они предоставляли пустую реализацию каждого из недостающих методов, которая ничего не делает.

Тремя вышеупомянутыми методами, которые могут предоставляться классами промежуточного ПО Falcon, являются следующие.

process_request `process_request(self, req, resp)`

Вызывается перед маршрутизацией запроса. Если он изменяет аргумент *req*, то это может повлиять на выбор маршрута.

process_resource `process_resource(self, req, resp, resource)`

Вызывается после маршрутизации запроса, но перед обработкой ресурса. Аргумент *resource* может иметь значение *None*, если по данному маршруту не удалось найти ресурс, соответствующий запросу.

process_response `process_response(self, req, resp, resource)`

Вызывается после возврата управления методом-ответчиком, если таковой вызывается. Аргумент *resource* может иметь значение None, если по данному маршруту не удалось найти ресурс, соответствующий запросу.

Вышеизложенное иллюстрирует приведенный ниже простой пример.

Листинг 20.1. Пример приложения Falcon

```
import datetime, falcon
one_year = datetime.timedelta(days=365)

class Greeter(object):
    def on_get(self, req, resp):
        lastvisit = req.cookies.get('lastvisit')
        now = datetime.datetime.now()
        newvisit = now.ctime()
        thisvisit = '<p>This visit on {} UTC</p>'.format(newvisit)
        resp.set_cookie('lastvisit', newvisit,
                        expires=now+one_year, secure=False)
        resp_body = [
            '<html><head><title>Hello, visitor!</title>'
```

```
            '</head><body>']
        if lastvisit is None:
            resp_body.extend((
            '<p>Welcome to this site on your first visit!</p>',
            thisvisit,
            '<p>Please Refresh the web page to proceed</p>'))
        else:
            resp_body.extend((
            '<p>Welcome back to this site!</p>',
            '<p>You last visited on {} UTC</p>'.format(lastvisit),
            thisvisit))
        resp_body.append('</body></html>')
        resp.content_type = 'text/html'
        resp.stream = resp_body
        resp.stream_len = sum(len(x) for x in resp_body)

app = falcon.API()
greet = Greeter()
app.add_route('/', greet)
```

В этом примере продемонстрировано использование лишь некоторых из фундаментальных строительных блоков, предлагаемых веб-фреймворком Falcon, — API, одного класса ресурса и одного экземпляра, подготавливающего ответ. (В данном случае установка значения `resp.stream` вполне естественна, поскольку мы подготовили в `resp_body` список строковых фрагментов тела ответа. Использование инструкции `resp.body = ''.join(resp_body)` также было бы приемлемым вариантом.) В примере также показано, как посредством установки и использования cookie-файла можно обеспечить минимальную поддержку непрерывности состояния в промежутках между отдельными взаимодействиями с сервером из одного и того же браузера.

Если бы приложение включало несколько ресурсов и методов-ответчиков, то было бы целесообразно установить cookie-файл `lastvisit` независимо от того, посредством какого ресурса и метода было "посещено" приложение. Ниже приведен пример кода подходящего класса промежуточного ПО и показано, как инстанциализировать `falcon.API` экземпляром указанного класса.

```
class LastVisitSettingMiddleware(object):
    def process_request(self, req, resp):
        now = datetime.datetime.now()
        resp.set_cookie('lastvisit', now.ctime(),
                        expires=now+one_year, secure=False)

lastvisitsetter = LastVisitSettingMiddleware()

app = falcon.API(middleware=[lastvisitsetter])
```

Теперь установку cookie-файла в вызове `Greeter.on_get` можно удалить, и при этом приложение по-прежнему будет работать нормально.

Flask

Flask (`flask.pocoo.org`) — самый популярный облегченный фреймворк Python для разработки веб-приложений. Несмотря на свою легковесность, он включает сервер для разработки и отладчик, а также использует другие хорошо подобранные пакеты, такие как набор инструментов Werkzeug и шаблонизатор jinja2 (оба этих пакета первоначально были написаны Армином Ронахером, автором Flask).

В дополнение к сайту проекта просмотрите соответствующие источники на сайтах GitHub (`https://github.com/pallets/flask`) и PyPI (`https://pypi.python.org/pypi/Flask`) и ознакомьтесь с документацией фреймворка (`http://flask.pocoo.org/docs/0.12/`). О способах запуска Flask в Google App Engine (на локальном компьютере или на серверах Google на сайте `appspot.com`) можно прочитать на сайте GitHub (`https://github.com/googlearchive/appengine-flask-skeleton` и `https://github.com/kamalgill/flask-appengine-template` — более структурированный подход).

Основной класс, предоставляемый пакетом `flask`, называется `Flask`. Экземпляр `flask.Flask`, кроме того что сам является WSGI-приложением, обертывает WSGI-приложение как собственное свойство `wsgi_app`. Если требуется дополнительно обернуть WSGI-приложение в какое-то промежуточное ПО WSGI, используйте следующую идиому.

```
import flask

app = flask.Flask(__name__)
app.wsgi_app = some_middleware(app.wsgi_app)
```

Инстанциализируя класс `flask.Flask`, всегда передавайте ему в качестве первого аргумента имя приложения (достаточно использовать специальную переменную `__name__` модуля, в котором вы инстанциализируете этот класс). Если вы инстанциализируете `flask.Flask` из пакета (обычно это делается в методе *__init__.py*), то в этом случае можно воспользоваться инструкцией `__name__.partition('.')[0]`. Если необходимо, можно дополнительно передать такие необязательные именованные параметры, как `static_folder` и `template_folder`, позволяющие указать, где находятся статические файлы и шаблоны jinja2, однако необходимость в этом возникает лишь в редких случаях — обычно их местоположение, используемое по умолчанию (папки *static* и *templates* соответственно, находящиеся в одной папке со сценарием Python, который инстанциализирует класс `flask.Flask`), вполне вас устроит.

Экземпляр *app* класса `flask.Flask` содержит свыше 100 методов и свойств, многие из которых, такие как *функции представления* (обслуживают глаголы HTTP, применяемые к URL-адресам), *функции-перехватчики* (хуки, позволяющие изменить запрос до его обработки или ответ после того, как он был создан), обработчики ошибок и т.п., являются декораторами, связывающими функции с экземпляром *app* в различных ролях.

При создании экземпляра класса `flask.Flask` ему передается небольшое количество параметров (причем они не относятся к числу тех, которые обычно вам приходится вычислять в своем коде), и он предоставляет декораторы, которые понадобятся вам, когда вы, скажем, будете определять функции представления. Таким образом, обычная схема использования пакета `flask` предполагает создание экземпляра *app* в самом начале работы вашего приложения, чтобы декораторы *app*, а также другие методы и свойства были доступны к тому моменту, когда вы будете определять функции представления.

В связи с тем, что используется единственный глобальный объект приложения *app*, у вас может возникнуть вопрос: насколько потокобезопасными являются свойства и атрибуты экземпляра *app* в отношении доступа к ним, а также изменения и повторного связывания? Вам не о чем беспокоиться: все они — *прокси-объекты* к фактическим объектам, существующим в *контексте* конкретного запроса в конкретном потоке. Никогда не проверяйте тип этих свойств (в действительности их типы являются типами прокси-объектов), и все будет нормально; в Python проверка типов вообще не считается удачной идеей.

Flask также предоставляет много других вспомогательных функций и классов. Последние часто являются подклассами или обертками классов из других пакетов, обеспечивающими удобную интеграцию Flask. Например, классы `Request` и `Response` фреймворка Flask всего лишь добавляют немного вспомогательной функциональности, наследуя ее от соответствующих классов набора инструментов Werkzeug.

Объекты запросов Flask

Класс `flask.Request` содержит большое количество свойств, подробно описанных в документации Flask (`http://flask.pocoo.org/docs/0.12/api/#incoming-request-data`). Наиболее часто используемые из них описаны ниже.

`cookies`
: Словарь с cookie-значениями из запроса.

`data`
: Строка, представляющая тело запроса (для запросов POST и PUT).

`files`
: Экземпляр `MultiDict`, значениями которого являются файловые объекты, выгружаемые с запросом (для запросов POST и PUT), а ключами — имена файлов.

`headers`
: Экземпляр `MultiDict`, содержащий заголовки запроса.

`values`
: Экземпляр `MultiDict`, содержащий параметры запроса (взятые либо из строки запроса, либо, в случае запросов POST и PUT, из формы, являющейся телом запроса).

Тип MultiDict подобен типу dict, за исключением того, что он может иметь несколько значений для ключа. При индексировании и применении метода get к экземпляру *m* класса MultiDict возвращается одно из этих значений, выбираемое произвольно. Чтобы получить список значений для ключа (этот список пуст, если данный ключ отсутствует в *m*), вызовите метод *m*.getlist(key).

Объекты ответов Flask

Часто функции представления Flask просто возвращают строку (которая становится телом ответа): Flask прозрачно обертывает эту строку экземпляром *r* класса flask.Response, снимая с вас все заботы относительно класса ответа. Однако в некоторых ситуациях вы захотите изменить заголовки ответа. В этом случае выполните в функции представления инструкцию *r*=flask.make_response(*astring*), внесите в словарь *r*.headers типа MultiDict требуемые изменения и верните *r*. (Не используйте *r*.headers для установки cookie-файла; вместо этого выполните вызов *r*.set_cookie с аргументами, указанными для метода set_cookie в табл. 20.1.)

Некоторые из встроенных средств интеграции Flask с другими системами не требуют создания подклассов: например, интеграция шаблонизации неявно внедряет в контекст jinja2 глобальные объекты Flask config, request, session и g (последний является удобным универсальным глобальным хранилищем flask.g, играющим роль прокси-объекта к контексту приложения, в котором ваш код может хранить все, что вы решите "припасти" на период длительности обслуживаемого запроса), а также функции url_for (предназначена для преобразования конечной точки в соответствующий URL-адрес; то же самое, что flask.url_for) и get_flashed_messages (предназначена для поддержки концепции *всплывающих сообщений* Flask, которые в данной книге не рассматриваются; то же самое, что flask.get_flashed_messages). Flask предоставляет вашему коду удобные способы внедрения дополнительных фильтров, функций и значений в контекст jinja2 без создания каких-либо подклассов.

Большинство официально признанных или одобренных расширений Flask (http://flask.pocoo.org/extensions/; свыше 50 на момент выхода книги) применяют аналогичные подходы, предоставляя классы и вспомогательные функции для бесшовной интеграции других популярных систем с вашими Flask-приложениями.

Flask также вводит ряд собственных дополнительных концепций, таких как концепция *сигналов* (http://flask.pocoo.org/docs/0.12/signals/), обеспечивающих более свободное динамическое связывание в шаблоне проектирования "публикация/подписка", и концепция *эскизов* (blueprints; http://flask.pocoo.org/docs/0.12/blueprints/), предлагающих основную часть функциональности Flask-приложения, что облегчает рефакторинг крупных приложений гибкими способами. В данной книге эти передовые концепции не рассматриваются.

Ниже представлен простой пример использования фреймворка Flask.

Листинг 20.2. Пример приложения Flask

```
import datetime, flask
app = flask.Flask(__name__)
app.permanent_session_lifetime = datetime.timedelta(days=365)
app.secret_key = b'\xc5\x8f\xbc\xa2\x1d\xeb\xb3\x94;:d\x03'

@app.route('/')
def greet():
    lastvisit = flask.session.get('lastvisit')
    now = datetime.datetime.now()
    newvisit = now.ctime()
    template = '''
      <html><head><title>Hello, visitor!</title>
      </head><body>
      {% if lastvisit %}
        <p>Welcome back to this site!</p>
        <p>You last visited on {{lastvisit}} UTC</p>
        <p>This visit on {{newvisit}} UTC</p>
      {% else %}
        <p>Welcome to this site on your first visit!</p>
        <p>This visit on {{newvisit}} UTC</p>
        <p>Please Refresh the web page to proceed</p>
      {% endif %}
      </body></html>'''
    flask.session['lastvisit'] = newvisit
    return flask.render_template_string(
      template, newvisit=newvisit, lastvisit=lastvisit)
```

В этом примере продемонстрировано использование лишь некоторых из фундаментальных строительных блоков, предлагаемых фреймворком Flask: класса Flask, функции представления и рендеринга ответа (в данном случае посредством применения функции `render_template_string` к шаблону jinja2; на практике шаблоны обычно хранятся в отдельных файлах, визуализируемых с помощью функции `render_template`). В примере также показано, как обеспечить непрерывность состояния в промежутках между отдельными взаимодействиями с сервером из одного и того же браузера с помощью удобной переменной `flask.session`. (В другом возможном варианте этого примера, более близко имитирующем пример 20.1, можно было бы не использовать шаблонизатор jinja2, а поместить HTML-ответ в код Python и вместо сеанса использовать непосредственно cookie-файл. Однако в реальных Flask-приложениях использование jinja2 и сеансов является более предпочтительным вариантом.)

Если бы в этом представлении использовались несколько функций представления, то, вероятно, было бы целесообразно установить cookie-файл `lastvisit` в сеансе независимо от того, с помощью какого URL-адреса было "посещено" приложение.

Ниже приведен пример соответствующего кода, выполнение которого при каждом запросе гарантируется декоратором.

```
@app.after_request()
def set_lastvisit(response):
    now = datetime.datetime.now()
    flask.session['lastvisit'] = now.ctime()
    return response
```

Теперь можно удалить инструкцию `flask.session['lastvisit'] = newvisit` из функции представления `greet`, и при этом приложение по-прежнему будет работать нормально.

Bottle

Другим заслуживающим внимания веб-фреймворком является Bottle. Он реализован в виде единственного файла *bottle.py* (содержащего свыше 4000 строк!) и не имеет никаких других зависимостей, кроме стандартной библиотеки. Строго говоря, Bottle не требует процесса установки: вы можете просто загрузить его в свой текущий каталог — например, с помощью команды **`curl --location -Ohttp://bottlepy.org/bottle.py`**[6] — для получения последнего варианта версии, находящейся в стадии разработки (хотя мы рекомендуем установить стабильный выпуск, что, как обычно, можно сделать с помощью команды **`pip install bottle`**). В этом единственном *.py*-файле Bottle поддерживает обе версии, v2 и v3. Он включает веб-сервер, который можно использовать в целях локальной разработки, и даже простой автономный механизм шаблонизации, основанный на внедрении Python в ваши шаблоны (хотя наряду с этим он позволяет использовать также систему шаблонизации jinja2 или другой внешний механизм шаблонизации, установленный вами).

В дополнение к сайту проекта (`http://bottlepy.org/docs/stable/`) просмотрите соответствующий материал на сайте GitHub (`https://github.com/bottlepy/bottle`). Bottle поставляется с адаптерами для многих WGGI-серверов, но один полезный "каркас" для запуска `bottle` с использованием Google App Engine (на локальном компьютере или на серверах Google на сайте `appspot.com`) описан на странице GitHub (`https://github.com/GoogleCloudPlatform/appengine-bottle-skeleton`).

Основной класс, предоставляемый пакетом `bottle`, называется `Bottle`. Экземпляр `bottle.Bottle`, кроме того что он является WSGI-приложением, обертывает WSGI-приложение как собственный атрибут `wsgi`. Если требуется дополнительно обернуть WSGI-приложение в какое-то промежуточное ПО WSGI, используйте следующую идиому.

```
import bottle

app = bottle.Bottle()
app.wsgi = some_middleware(app.wsgi)
```

[6] Или `wget http://bottlepy.org/bottle.py`.

Инстанциализируя класс `bottle.Bottle`, вы можете, если это необходимо, передать именованные параметры `catchall` и `autojson` (по умолчанию оба они имеют значение `True`). Если `catchall` имеет значение `True`, то `Bottle` перехватывает и обрабатывает все исключения. Передав для него значение `False`, вы позволите исключениям распространяться (полезно, лишь если вы обертываете WSGI-приложение отладочным промежуточным ПО). Если параметр `autojson` равен `True`, то `Bottle` распознает, когда функция обратного вызова возвращает словарь, и в этом случае сериализует данный словарь в качестве тела ответа с помощью вызова `json.dumps` и устанавливает для заголовка `Content-Type` ответа значение `'application/json'`. Передав для него значение `False`, вы позволите таким словарям распространяться (полезно, лишь если вы обертываете WSGI-приложение промежуточным ПО, обрабатывающим эти словари).

Экземпляр *app* класса `bottle.Bottle` предоставляет десятки методов и свойств, в том числе декораторов, связывающих функции с экземпляром *app* в различных ролях, например в качестве *функций обратного вызова* (обслуживание глаголов HTTP, применяемых к заданным URL-адресам) или *функций-перехватчиков* (хуков, позволяющих изменить запрос до его обработки или ответ после того, как он был создан). Таким образом, обычная схема использования пакета `bottle` предполагает создание экземпляра *app* в самом начале работы вашего приложения, чтобы декораторы *app*, а также другие методы и свойства были доступны к тому моменту, когда вы будете определять функции обратного вызова. Зачастую вам даже не понадобится явно инстанциализировать класс `bottle.Bottle`: Bottle создает для вас "объект приложения по умолчанию", и функции уровня модуля делегируют свое выполнение одноименным методам этого объекта. Если вам нужен доступ к объекту приложения по умолчанию, вызовите функцию `bottle.default_app`. Работа с объектом приложения по умолчанию вполне допустима в случае простых веб-приложений небольшого размера, хотя мы рекомендуем явную инстанциализацию, как более соответствующую духу Python.

Маршрутизация запросов в Bottle

Для маршрутизации входящих запросов к функциям обратного вызова Bottle предоставляет функцию `route(`*`path`*`, `*`method`*`='GET', `*`callback`*`=None)`. Чаще всего вы будете использовать ее для декорирования функций обратного вызова, но с целью обеспечения большей гибкости вам также предоставляется возможность вызывать ее непосредственно, передавая функцию обратного вызова в качестве аргумента *`callback`*.

В качестве аргумента *`method`* можно использовать имя любого HTTP-метода (`'GET'`, `'DELETE'`, `'PATCH'`, `'POST'`, `'PUT'`) или список таких имен. Кроме того, Bottle предоставляет функции, являющиеся синонимами `route`, но уже с заданным конкретным значением для *метода*. Имена этих функций совпадают с именами соответствующих HTTP-методов (`get`, `delete`, `patch`, `post`, `put`).

Аргумент *path* — это строка, соответствующая URI, по которому должен быть направлен запрос. Например, строка `'/'` направляет запросы к корневому URL-адресу сайта. Эта строка может включать один или несколько *заполнителей* — идентификаторов, заключенных в угловые скобки <>. В таком случае функция обратного вызова должна принимать аргументы, имена которых совпадают с этими идентификаторами (если аргумент *path* не содержит заполнителей, Bottle вызывает функцию обратного вызова без аргументов). Каждый заполнитель сопоставляется с одним или несколькими символами URI вплоть до первого встретившегося в URI символа косой черты, но исключая его.

Например, если в качестве *path* задана строка `'/greet/<name>/first'`, а URI представлен строкой `'/greet/alex/first'`, то функция обратного вызова должна принимать аргумент с именем *name*, и Bottle вызывает ее, используя значение `'alex'` для этого аргумента.

С целью обеспечения большей гибкости за идентификатором заполнителя может следовать фильтр. Для произвольного идентификатора *ident*:

- `<`*ident*`:int>` сопоставляется с цифрами и преобразует их в целочисленное значение;
- `<`*ident*`:float>` сопоставляется с цифрами и необязательным символом "точка" (`.`) и преобразует их в значение с плавающей точкой;
- `<`*ident*`:path>` сопоставляется с символами, включая символы косой черты (нежадным способом, если далее еще имеются символы);
- `<`*ident*`:re:`*pattern*`>` сопоставляется с указанным вами произвольным шаблоном регулярного выражения. Bottle даже разрешает вашему коду использовать собственные нестандартные фильтры, но эта более сложная тема в данной книге не рассматривается.

Объекты запросов Bottle

Класс `bottle.Request` содержит большое количество свойств, подробно описанных в документации Bottle (`http://bottlepy.org/docs/dev/api.html#bottle.BaseRequest`). Наиболее часто используемые свойства описаны ниже.

`body`
: Файловый объект, который допускает поиск и представляет тело запроса (для запросов POST и PUT).

`cookies`
: Экземпляр `FormsDict` с cookie-значениями из запроса.

`files`
: Экземпляр `FormsDict`, значениями которого являются файловые объекты, причем все файлы выгружаются с запросом (для запросов POST и PUT), а ключами являются имена файлов.

`headers`
: Экземпляр `WSGIHeaderDict` с заголовками запроса.

`params`
: Класс `FormsDict`, содержащий параметры запроса (либо из строки запроса, либо, в случае запросов POST и PUT, из формы, являющейся телом запроса).

Класс `FormsDict` аналогичен типу `dict`, за исключением того, что он может иметь несколько значений для ключа. Индексирование или применение метода `get` к экземпляру *f* класса `FormsDict` возвращает одно из этих значений, выбираемое произвольно. Чтобы получить список значений для ключа (возможно, пустой в случае отсутствия данного ключа), вызовите метод *f*`.getall(`*key*`)`. Класс `WSGIHeaderDict` аналогичен типу `dict`, за исключением того, что его ключи и значения являются строками (байтовыми в v2, Unicode в v3), а ключи нечувствительны к регистру.

Подписанные cookie-файлы в `bottle.request.cookies` не декодируются: для корректного доступа к подписанному cookie-файлу в декодированной форме воспользуйтесь вызовом `bottle.request.get_cookie(`*key*`, `*secret*`)`, который возвращает значение `None`, если данный cookie-файл отсутствует или не подписан заданной строкой *secret*.

`bottle.request` — это потокобезопасный прокси-объект для доступа к обрабатываемому запросу.

Объекты ответов Bottle

Во многих случаях функции обратного вызова Bottle просто возвращают результат: Bottle прозрачно обертывает экземпляром *r* класса `bottle.Response` байтовую строку тела ответа (подготовленную в зависимости от типа результата), снимая с вас все заботы относительно класса ответа.

Результат может быть байтовой строкой (используемой в качестве тела ответа), строкой Unicode (кодировка которой соответствует заголовку `Content-Type`, по умолчанию — `utf8`), словарем (сериализованным с помощью вызова `json.dumps`; при этом также устанавливается тип содержимого `'application/json'`), файловым объектом (любой объект, имеющий метод `read`) или любым итерируемым объектом, элементы которого являются строками (байтовыми или Unicode).

Если результат представляет собой экземпляр класса `HTTPError` или `HTTPResponse`, то это указывает на ошибку, точно так же, как и возбуждение такого экземпляра. (Различие между этими двумя классами состоит в том, что если вы возвращаете или возбуждаете экземпляр `HTTPError`, то Bottle применяет обработчик ошибок, в то время как экземпляры `HTTPResponse` обходят обработку ошибок.)

Если ваша функция обратного вызова должна изменить заголовки ответа, то это можно сделать в объекте `bottle.response` перед тем, как вернуть тело ответа.

`bottle.response` — это потокобезопасный прокси-объект для доступа к подготавливаемому ответу.

Не используйте `bottle.response.headers` для установки cookie-файла. Вместо этого выполните вызов `bottle.response.set_cookie` с аргументами, указанными для метода `set_cookie` в табл. 20.1, и, возможно, необязательным аргументом `secret`, значение которого используется для создания криптографической *подписи* cookie-файла. Такая подпись защищает cookie-файл от подделки (хотя и не скрывает его содержимое), и Bottle использует эту возможность для того, чтобы позволить вам использовать объект любого типа в качестве значения cookie и сериализовать его с помощью модуля `pickle` (размер результирующей байтовой строки вызова `pickle.dumps` не должен превышать предельного значения 4 Кбайт, установленного для cookie-файлов).

Механизм шаблонов Bottle

Bottle предоставляет собственный механизм шаблонов Simple Templating System (`http://bottlepy.org/docs/dev/stpl.html`), который в данной книге не рассматривается. Однако использовать другие механизмы шаблонов, такие как jinja2 (раздел "Пакет `jinja2`" в главе 22), не составляет большого труда (при условии, что вы предварительно установили их отдельно в среде своей установки Python или в виртуальной среде).

По умолчанию Bottle загружает файлы шаблонов либо из текущего каталога, либо из его подкаталога *./views/* (вы можете задать другой каталог, внеся соответствующее изменение в список строк `bottle.TEMPLATE_PATH`). Чтобы создать объект шаблона, вызовите метод `bottle.template`: его первый аргумент — либо имя файла шаблона, либо строка шаблона. Все остальные позиционные аргументы являются словарями, кумулятивно устанавливающими переменные шаблона, и такими же являются все именованные аргументы, за исключением, возможно, аргумента `template_adapter`, позволяющего задать используемый механизм шаблонизации (значением по умолчанию является собственный шаблонизатор Bottle). Функция `bottle.jinja2template` — это та же самая функция, но задающая адаптер `jinja2`.

Создав объект шаблона, вы можете вызвать его метод `render` для получения визуализируемых им строк или, что встречается чаще, просто возвращаете объект шаблона из функции обратного вызова, и Bottle использует его для создания тела ответа.

Еще более простой способ использования шаблонов в функции обратного вызова предлагает декоратор `bottle.view` (`bottle.jinja2view`, чтобы использовать шаблонизатор `jinja2`), который принимает те же аргументы, что и функция `bottle.template`. Если функция декорирована таким способом и возвращает словарь, то этот словарь используется для установки или переопределения переменных шаблона, а Bottle действует так, как если бы функция вернула шаблон.

Другими словами, следующие три функции эквивалентны.

```
import bottle

@bottle.route('/a')
```

```
def a():
    t = bottle.template('tp.html', x=23)
    return t.render()

@bottle.route('/b')
def b():
    t = bottle.template('tp.html', x=23)
    return t

@bottle.route('/c')
@bottle.view('tp.html')
def c():
    return {'x': 23}
```

Если встроенной функциональности Bottle (плюс дополнительные возможности, добавленные вами самостоятельно посредством явного использования стандартной библиотеки и сторонних модулей) недостаточно, Bottle допускает расширение его собственной функциональности посредством расширений (плагинов). Bottle официально признает и перечисляет свыше десятка повторно используемых сторонних расширений (`http://bottlepy.org/docs/dev/plugins/index.html`) и документирует, как написать собственные расширения. Расширения Bottle в данной книге не рассматриваются.

Вышеизложенное иллюстрирует приведенный ниже простой пример.

Листинг 20.3. Пример приложения Bottle

```
import datetime, bottle
one_year = datetime.timedelta(days=365)

@bottle.route('/')
def greet():
    lastvisit = bottle.request.get_cookie('lastvisit')
    now = datetime.datetime.now()
    newvisit = now.ctime()
    thisvisit = '<p>This visit on {} UTC</p>'.format(newvisit)
    bottle.response.set_cookie('lastvisit', newvisit,
                               expires=now+one_year)
    resp_body = ['<html><head><title>Hello, visitor!</title>'
                 '</head><body>']
    if lastvisit is None:
        resp_body.extend((
          '<p>Welcome to this site on your first visit!</p>',
          thisvisit,
          '<p>Please Refresh the web page to proceed</p>'))
    else:
        resp_body.extend((
          '<p>Welcome back to this site!</p>',
```

```
            '<p>You last visited on {} UTC</p>'.format(lastvisit),
            thisvisit))
    resp_body.append('</body></html>')
    bottle.response.content_type = 'text/html'
    return resp_body

if __name__ == '__main__':
    bottle.run(debug=True)
```

В этом примере продемонстрировано использование лишь некоторых из многочисленных фундаментальных строительных блоков, предлагаемых веб-фреймворком Bottle, — используемого по умолчанию объекта *app*, одной функции обратного вызова и одного маршрута для подготовки ответа (в данном случае просто возвращается список строковых фрагментов тела ответа, что часто оказывается очень удобным). В примере также показано, как посредством установки и использования cookie-файла можно обеспечить минимальную поддержку непрерывности состояния в промежутках между отдельными взаимодействиями с сервером из одного и того же браузера.

Если бы приложение включало несколько маршрутов, то, несомненно, было бы целесообразно установить cookie-файл `lastvisit`, независимо от того, какой маршрут был использован для "посещения" приложения. Ниже приведен пример декорирования функции-перехватчика, выполняющейся после каждого запроса.

```
@bottle.hook('after_request')
def set_lastvisit():
    now = datetime.datetime.now()
    bottle.response.set_cookie('lastvisit', now.ctime(),
                               expires=now+one_year)
```

Теперь вызов `bottle.response.set_cookie` можно удалить из функции обратного вызова `greet`, и при этом приложение по-прежнему будет работать нормально.

webapp2

Последний из рассматриваемых, но не последний по значимости веб-фреймворк — webapp2 (`https://webapp2.readthedocs.io/en/latest/`). С Django и jinja2 его объединяет то, что он работает в связке с Google App Engine (но, разумеется, точно так же, как jinja2 и Django, может одинаково хорошо работать с другими веб-серверами). О том, как установить webapp2 с помощью Google App Engine, описано в основном руководстве Quick Start (`https://webapp2.readthedocs.io/en/latest/tutorials/quickstart.html`). Чтобы установить webapp2 независимо от Google App Engine, обратитесь к альтернативному руководству Quick Start (`https://webapp2.readthedocs.io/en/latest/tutorials/quickstart.nogae.html#tutorials-quickstart-nogae`). Кроме того, вы можете ознакомиться с исходным кодом фреймворка (`https://github.com/GoogleCloudPlatform/webapp2`) и прочитать соответствующий материал на сайте PyPI (`https://pypi.python.org/pypi/webapp2`).

Основной класс, предоставляемый webapp2, называется `WSGIApplication`. Экземпляр `webapp2.WSGIApplication` является WSGI-приложением, и вы инстанциализируете его, используя вплоть до трех именованных аргументов: *`routes`* — список определений маршрутизации, *`debug`* — булево значение, которое по умолчанию равно `False` (передача значения `True` включает режим отладки), и *`config`* — словарь с конфигурационной информацией приложения. Поскольку маршруты проще выражать, когда уже определены соответствующие обработчики, то при использовании webapp2 WSGI-приложение обычно инстанциализируют в конце модуля.

Объекты запросов webapp2

Класс `webapp2.Request` (текущий экземпляр доступен вашим методам-обработчикам как `self.request`) содержит большое количество свойств и методов, подробно описанных в документации webapp2. Наиболее часто используемыми из них являются три отображения: `cookies` — содержит имена cookie-файлов и соответствующие значения, `headers` — содержит имена заголовков и соответствующие значения (нечувствительные к регистру) и `params` — содержит имена параметров и соответствующие значения. Также имеются два удобных метода: `get` (как `params.get`, но при необходимости можно передать необязательный именованный аргумент `allow_multiple=True` для получения списка всех значений для данного параметра) и `get_all` (как `get` с аргументом `allow_multiple=True`).

Объекты ответов webapp2

Класс `webapp2.Response` (текущий экземпляр доступен вашим методам-обработчикам как `self.request`) предоставляет большое количество свойств и методов, подробно описанных в документации webapp2. Одно из свойств — отображение `headers`, которое вы будете часто использовать для установки заголовков ответа. Однако в конкретном случае установки cookie-файлов вместо этого свойства следует использовать метод `set_cookie`. Чтобы установить код состояния ответа, присвойте соответствующее значение свойству `status` — например, с помощью инструкции `self.response.status = '404 Not Found'`.

Обычно для создания тела ответа следует вызвать для ответа метод `write` (часто более одного раза). Альтернативный подход заключается в том, чтобы присвоить атрибуту `body` байтовую строку или же строку Unicode, которая кодируется с использованием кодировки UTF-8, если заголовок `Content-Type` заканчивается подстрокой `;charset=utf8` (или, что эквивалентно, если для атрибута `charset` установлено значение `'utf8'`).

Обработчики webapp2

Обработчики webapp2 — это классы, расширяющие класс `webapp2.RequestHandler`. Обработчики определяют такие методы, как `get` и `post`, имена которых повторяют (с использованием нижнего регистра букв) имена методов HTTP.

Эти методы получают доступ к текущему запросу через атрибут `self.request` и подготавливают ответ в виде атрибута `self.response`.

Обработчик может определить метод `dispatch`, который вызывается до определения соответствующего метода (например, `get` или `post`). Благодаря этому обработчик получает возможность изменить запрос, если в этом возникнет необходимость, выполнить пользовательскую диспетчеризацию запроса, а также (обычно после делегирования полномочий методу `dispatch` суперкласса) изменить результирующий ответ `self.response`. Подобные переопределения часто выполняются в общем базовом классе. Затем ваши фактические классы обработчиков могут наследовать необходимую функциональность от этого общего базового класса.

Маршрутизация в webapp2

Фреймворк webapp2 предоставляет класс `Route`, предназначенный для обработки сложных случаев маршрутизации запросов. Однако в большинстве случае вам будет достаточно рассматривать *маршрут* просто как пару `(template, handler)`.

Элемент *`template`* — это строка, которая может включать идентификаторы, заключенные в угловые скобки (`<>`). Такого рода компонент в угловых скобках совпадает с любой последовательностью символов (вплоть до символа косой черты, `/`, но не включая его) и передает совпавшую подстроку в качестве *именованного* аргумента методу обработчика.

Также возможен вариант, когда угловые скобки содержат символ `:` (двоеточие), за которым следует шаблон регулярного выражения. В подобных случаях компонент в угловых скобках сопоставляется с символами в соответствии с предоставленным шаблоном регулярного выражения и передает совпавшую подстроку в качестве *позиционного* аргумента методу обработчика.

Наконец, угловые скобки могут содержать идентификатор, за которым следуют символ `:` (двоеточие) и шаблон регулярного выражения. В подобных случаях компонент в угловых скобках сопоставляется с символами в соответствии с предоставленным шаблоном и передает совпавшую строку в качестве *именованного* аргумента методу обработчика.

Позиционные аргументы передаются лишь при отсутствии именованных аргументов. Если элемент *`template`* содержит компоненты обоих видов, то поиск соответствия тем компонентам, которые стали бы позиционными аргументами, выполняется, но соответствующие значения не передаются методу обработчика. Например, если аргументом *`template`* является строка `'/greet/<name>/<:.*>'`, то метод обработчика вызывается с именованным аргументом *`name`*, но без любого позиционного аргумента, который мог бы соответствовать замыкающей части URI.

Методы обработчиков, к которым ведут маршруты с элементом *`template`*, включающим компоненты в угловых скобках, должны принимать результирующие позиционные или именованные аргументы.

Обычно элементом *`handler`* является *класс обработчика* (раздел "Обработчики webapp2"). Однако возможны альтернативные варианты, из которых наиболее часто

используется задание аргумента *handler* в виде *строки*, идентифицирующей класс обработчика, например строки `'my.module.theclass'`. В этом случае соответствующий модуль не импортируется до тех пор, пока не возникнет необходимость в использовании данного конкретного класса: webapp2 импортирует его на лету (самое большее один раз) тогда и только тогда, когда он фактически потребуется.

Пакет `webapp2.extras`

Пакет `webapp2.extras` предоставляет несколько удобных модулей, бесшовно добавляя в webapp2 дополнительную функциональность, включая средства аутентификации и авторизации, интеграцию с jinja2 и другими механизмами шаблонизации, интернационализацию и *сеансы* (с возможными резервными хранилищами различного рода), упрощающие сохранение состояния по сравнению с тем, как это делается исключительно посредством одних лишь cookie-файлов.

Вышеизложенное иллюстрирует приведенный ниже простой пример.

Листинг 20.4. Пример приложения webapp2

```
import datetime, webapp2
one_year = datetime.timedelta(days=365)

class Greeter(webapp2.RequestHandler):
    def get(self):
        lastvisit = self.request.cookies.get('lastvisit')
        now = datetime.datetime.now()
        newvisit = now.ctime()
        thisvisit = '<p>This visit on {} UTC</p>'.format(newvisit)
        self.response.set_cookie('lastvisit', newvisit,
                                 expires=now+one_year)
        self.response.write(
            '<html><head><title>Hello, visitor!</title>'
            '</head><body>')
        if lastvisit is None:
            self.response.write(
              '<p>Welcome to this site on your first visit!</p>')
            self.response.write(thisvisit)
            self.response.write(
              '<p>Please Refresh the web page to proceed</p>')
        else:
            self.response.write(
              '<p>Welcome back to this site!</p>')
            self.response.write(
              '<p>You last visited on {} UTC</p>'
                                      .format(lastvisit))
            self.response.write(thisvisit)
        self.response.write('</body></html>')
```

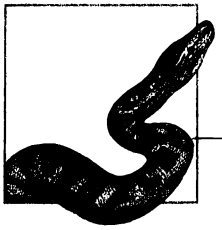
```
app = webapp2.WSGIApplication(
    [('/', Greeter),
    ]
)
```

В этом примере продемонстрировано использование лишь некоторых из многочисленных строительных блоков, предлагаемых веб-фреймворком webapp2, — объекта приложения, одного обработчика и одного маршрута для подготовки ответа (в данном случае просто выполняется запись в него). В примере также показано, как посредством установки и использования cookie-файла можно обеспечить минимальную поддержку непрерывности состояния в промежутках между отдельными взаимодействиями с сервером из одного и того же браузера.

Если бы приложение включало несколько маршрутов, то, несомненно, было бы целесообразно установить cookie-файл `lastvisit`, независимо от того, какой именно маршрут был использован для "посещения" приложения. Ниже приведен пример кода базового класса обработчика (от него должен наследоваться каждый обработчик!), который переопределяет метод `dispatch` и использует его переопределенную версию для установки cookie-файла.

```
class BaseHandler(webapp2.RequestHandler):
    def dispatch(self):
        webapp2.RequestHandler.dispatch(self)
        now = datetime.datetime.now()
        self.response.set_cookie('lastvisit', now.ctime(),
                                 expires=now+one_year)
class Greeter(BaseHandler): ...
```

Теперь вызов `self.response.set_cookie` можно удалить из метода `Greeter.get`, и при этом приложение по-прежнему будет работать нормально.



21

Электронная почта, MIME и другие сетевые кодировки

Данные, которые передаются по сети, — это потоки байтов, или, на сетевом жаргоне, — *октетов*. Разумеется, байты могут быть текстом в той или иной кодировке. Однако данные, которые необходимо передать по сети, часто имеют более сложную структуру, чем простой поток байтов или текста. MIME (Multipurpose Internet Mail Extensions — многоцелевые расширения интернет-почты) и другие стандарты кодирования заполняют эту брешь, структурируя способы представления информации в виде текстовых и байтовых данных. Довольно часто такие кодировки первоначально предназначались для электронной почты, но впоследствии стали применяться в веб-приложениях и во многих других сетевых системах. Python поддерживает разнообразные кодировки посредством многочисленных библиотечных модулей, таких как `base64`, `quopri` и `uu` (раздел "Преобразование двоичных данных в ASCII-текст"), а также модулей, входящих в пакет `email` (раздел "Обработка сообщений MIME и электронной почты").

Обработка сообщений MIME и электронной почты

Пакет `email` предоставляет набор средств для управления MIME-файлами, такими как сообщения электронной почты, публикации сетевых новостей (NNTP), файлы, передаваемые по протоколу HTTP, и т.п. Стандартная библиотека содержит и другие модули, реализующие отдельные части данной функциональности. Однако пакет `email` предлагает цельный и систематический подход к решению этих важных задач. Мы рекомендуем использовать пакет `email`, а не более старые модули, которые частично перекрываются его функциональностью. Пакет `email`, несмотря на свое название, не имеет ничего общего с получением или отправкой электронной почты: для этого предназначены модули `poplib` и `smtplib`, рассмотренные в разделе "Протоколы электронной почты" главы 19. Пакет `email` имеет дело с обработкой

MIME-сообщений (которые не обязательно являются почтовыми), когда они уже получены, или с их созданием, прежде чем они будут отправлены.

Функции пакета email

Пакет email предоставляет две функции-фабрики, которые возвращают экземпляр *m* класса email.message.Message. Эти функции основаны на функциональности класса email.parser.Parser, но с ними проще и удобнее работать. Поэтому модуль email.parser в книге не рассматривается.

message_from_string message_from_string(*s*)

Создает экземпляр *m* на основе анализа строки *s*.

message_from_file message_from_file(*f*)

Создает экземпляр *m* на основе анализа содержимого объекта *f*, подобного текстовому файлу, который должен быть открыт для чтения.

Кроме того, версия v3 предоставляет две аналогичные функции-фабрики, позволяющие создавать объекты сообщений из байтовых строк и двоичных файлов.

message_from_bytes message_from_bytes(*s*)

Создает экземпляр *m* на основе анализа байтовой строки *s*.

message_from_binary_file message_from_binary_file(*f*)

Создает экземпляр *m* на основе анализа содержимого объекта *f*, подобного двоичному файлу, который должен быть открыт для чтения.

Модуль email.message

Модуль email.message предоставляет класс Message. Все компоненты пакета email создают, изменяют или используют экземпляры Message. Экземпляр *m* класса Message моделирует MIME-сообщение, включая его заголовки и *полезную нагрузку* (содержащиеся данные). Чтобы создать первоначально пустой объект *m*, вызовите конструктор Message без аргументов. Чаще всего вы будете создавать экземпляр *m* посредством анализа почтового сообщения с помощью функций-фабрик message_from_string и message_from_file или косвенных средств, таких как классы, рассмотренные в разделе "Создание сообщений". Полезной нагрузкой экземпляра *m* может быть строка, другой экземпляр Message или (в случае составных сообщений) список других экземпляров Message.

В создаваемых почтовых сообщениях можно установить любые заголовки. Заголовки, предназначенные для самых разных целей, специфицированы в нескольких документах RFC. Наибольший интерес для нас представляет спецификация RFC 2822 (http://www.faqs.org/rfcs/rfc2822.html). Заголовки хранятся в экземпляре *m*

класса Message вместе с содержимым сообщения. Экземпляр *m* — это отображение с именами заголовков в качестве ключей и строками заголовков в качестве значений.

Чтобы было удобнее работать с экземпляром *m*, его семантика как отображения сделана отличающейся от семантики словаря. Ключи *m* нечувствительны к регистру. Экземпляр *m* хранит заголовки в том порядке, в каком они добавлялись, и этот же порядок соблюдается при выводе списков заголовков методами keys, values и items. Экземпляр *m* может иметь более одного заголовка с именем *key*: выражение *m*[*key*] возвращает один из таких заголовков, выбираемый произвольно (или значение None, если такого заголовка не существует), а del *m*[*key*] удаляет все подобные заголовки (отсутствие данного заголовка не приводит к ошибке).

Чтобы получить список всех заголовков с заданным именем, вызовите метод *m*.get_all(*key*). Вызов len(*m*) возвращает общее количество заголовков, включая дубликаты, а не только количество различных заголовков. Если заголовка с именем *key* не существует, *m*[*key*] возвращает значение None и не возбуждает исключение KeyError (т.е. ведет себя подобно *m*.get(*key*)). Инструкция del *m*[*key*] в этом случае ничего не делает, а вызов *m*.get_all(*key*) возвращает пустой список. В версии v2 непосредственное выполнение цикла по *m* невозможно, вместо этого следует использовать цикл по *m*.keys().

Экземпляр *m* класса Message предоставляет следующие атрибуты и методы, предназначенные для работы с заголовками и полезной нагрузкой экземпляра *m*.

add_header *m*.add_header(*_name*, *_value*, ****_params*)

Данный метод аналогичен инструкции *m*[*_name*]=*_value*, но ему также можно передать параметры заголовков в качестве именованных аргументов. В имени каждого именованного аргумента *pname=pvalue* метод add_header заменяет символы подчеркивания дефисами, а затем присоединяет к значению заголовка параметр в таком виде:

; *pname*="*pvalue*"

Если *pvalue* равно None, то метод add_header присоединяет только параметр '*;pname*'.

Если значение параметра содержит символы, отличающиеся от символов ASCII, укажите его в виде кортежа из трех элементов: (*CHARSET*, *LANGUAGE*, *VALUE*). Элемент *CHARSET* задает название кодировки, применяемой к значению. Элемент *LANGUAGE* обычно имеет значение None или '', но его можно использовать для указания используемого языка в соответствии с документом RFC 2231 (http://www.rfc-base.org/rfc-2231.html), а элемент *VALUE* — это строковое значение, которое содержит символы, отличные от символов ASCII.

as_string *m*.as_string(*unixfrom*=False)

Возвращает все сообщение в виде строки. Если аргумент *unixfrom* имеет значение *True*, то результат включает также первую строку, обычно начинающуюся со слова 'From ' и называемую *конвертом заголовка* сообщения. Метод __str__ класса — это то же самое, что и метод as_string экземпляра, но в версии v2 — с аргументом *unixfrom*, установленным в True.

attach *m*.attach(*payload*)

Добавляет сообщение *payload* в полезную нагрузку экземпляра *m*. Если до вызова данного метода полезная нагрузка экземпляра *m* была представлена значением None, то после него она представляется списком [*payload*], включающим один элемент. Если она была представлена списком сообщений, то вызов данного метода присоединяет *payload* к этому списку. Если же полезная нагрузка экземпляра *m* до вызова данного метода представляла собой объект другого типа, то вызов *m*.attach(*payload*) возбуждает исключение MultipartConversionError.

epilogue Атрибут *m*.epilogue может содержать либо значение None, либо строку, которая становится частью сообщения, следующей за последним вхождением строки границы в конце сообщения. Обычно почтовые программы не отображают этот текст. Атрибут epilogue — обычный атрибут экземпляра *m*, и ваша программа может получать к нему доступ при обработке экземпляра *m*, созданного любым способом, а также связывать его в процессе создания или изменения экземпляра *m*.

get_all *m*.get_all(*name*, *default*=None)

Возвращает список значений всех заголовков с именем *name* в том порядке, в каком они добавлялись в *m*. Если *m* не имеет заголовков с таким именем, то вызов get_all возвращает значение *default*.

get_boundary *m*.get_boundary(*default*=None)

Возвращает строковое значение параметра boundary заголовка Content-Type экземпляра *m*. Если экземпляр *m* не имеет заголовка Content-Type или этот заголовок не имеет параметра boundary, то вызов get_boundary возвращает значение *default*.

get_charsets *m*.get_charsets(*default*=None)

Возвращает список *L* строковых значений параметра charset заголовков Content-Type экземпляра *m*. Если экземпляр *m* представляет составное сообщение, то список *L* содержит по одному элементу на каждую часть сообщения; в противном случае *L* имеет длину 1. Для частей сообщения, не имеющих ни заголовка Content-Type, ни параметра charset, или если тип основного содержимого отличается от 'text', то соответствующим элементом *L* является значение *default*.

get_content_maintype *m*.get_content_maintype(*default*=None)

Возвращает тип основного содержимого экземпляра *m*: строку '*maintype*' в нижнем регистре, взятую из заголовка Content-Type. Например, если заголовок Content-Type имеет значение 'text/html', то вызов get_content_maintype возвращает строку 'text'. Если экземпляр *m* не имеет заголовка Content-Type, то вызов get_content_maintype возвращает значение *default*.

get_content_subtype *m*.get_content_subtype(*default*=None)

Возвращает подтип содержимого экземпляра *m*: строку '*subtype*' в нижнем регистре, взятую из заголовка Content-Type. Например, если заголовок Content-Type имеет значение 'text/html', то вызов get_content_subtype возвращает строку 'html'. Если экземпляр *m* не имеет заголовка Content-Type, то вызов get_content_subtype возвращает значение *default*.

get_content_type *m*.get_content_type(*default*=None)

Возвращает тип содержимого экземпляра *m*: строку 'maintype/subtype' в нижнем регистре, взятую из заголовка Content-Type. Например, если заголовок Content-Type имеет значение 'text/html', то вызов get_content_type возвращает строку 'text/html'. Если экземпляр *m* не имеет заголовка Content-Type, то вызов get_content_type возвращает значение *default*.

get_filename *m*.get_filename(*default*=None)

Возвращает строковое значение параметра filename заголовка Content-Disposition экземпляра *m*. Если экземпляр *m* не имеет заголовка Content-Disposition или этот заголовок не имеет параметра filename, то метод get_filename возвращает значение *default*.

get_param *m*.get_param(*param*, *default*=None, *header*='Content-Type')

Возвращает строковое значение параметра *param* заголовка *header* экземпляра *m*. Возвращает пустую строку для параметра, заданного только именем (без значения). Если экземпляр *m* не имеет заголовка *header* или этот заголовок не имеет параметра *param*, то вызов get_param возвращает значение *default*.

get_params *m*.get_params(*default*=None, *header*='Content-Type')

Возвращает параметры заголовка *header* экземпляра *m* в виде списка кортежей из двух строк, каждый из которых представляет имя и значение параметра. Использует пустую строку в качестве значения для параметров, заданных только именем (без значения). Если *m* не имеет заголовка *header*, то вызов get_params возвращает значение *default*.

get_payload *m*.get_payload(*i*=None, *decode*=False)

Возвращает полезную нагрузку экземпляра *m*. Если вызов *m*.is_multipart() возвращает значение False, то *i* должно быть равно None, и вызов m.get_payload() возвращает всю полезную нагрузку экземпляра *m* — строку или экземпляр Message. Если аргумент *decode* имеет значение True, а значением заголовка Content-Transfer-Encoding является либо строка 'quoted-printable', либо строка 'base64', то вызов *m*.get_payload также декодирует полезную нагрузку. Если аргумент *decode* имеет значение False или если заголовок Content-Transfer-Encoding отсутствует или имеет другие значения, то вызов *m*.get_payload возвращает полезную нагрузку в неизменной виде.

Если вызов *m*.is_multipart() возвращает значение True, то аргумент *decode* должен иметь значение False. Если *i* равно None, то вызов *m*.get_payload() возвращает полезную нагрузку *m* в виде списка. В противном случае вызов *m*.get_payload(*i*) возвращает *i*-й элемент полезной нагрузки или возбуждает исключение TypeError, если *i*<0 или имеет слишком большое значение.

get_unixfrom *m*.get_unixfrom()

Возвращает строку конверта заголовка для экземпляра *m* или значение None, если таковой отсутствует в *m*.

is_multipart *m*.is_multipart()

Возвращает значение True, если полезная нагрузка экземпляра *m* является списком. В противном случае возвращает значение False.

preamble Атрибут *m*.preamble может содержать значение None или строку, которая становится частью строковой формы сообщения, предшествующей первому вхождению строки границы. Почтовая программа отображает такой текст, только если она не поддерживает составные сообщения, поэтому вы можете использовать данный атрибут для того, чтобы предупредить пользователя о том, что сообщение является составным и для его просмотра требуется другая программа. Атрибут preamble — обычный атрибут экземпляра *m*: ваша программа может получать к нему доступ при обработке экземпляра *m*, созданного любым способом, а также связывать его в процессе создания или изменения экземпляра *m*.

set_boundary *m*.set_boundary(*boundary*)

Устанавливает для параметра boundary заголовка Content-Type экземпляра *m* значение *boundary*. Если экземпляр *m* не имеет заголовка Content-Type, то данный метод возбуждает исключение HeaderParseError.

set_payload *m*.set_payload(*payload*)

Задает для всей полезной нагрузки экземпляра значение *payload*, которое должно быть строкой или списком экземпляров Message в соответствии с заголовком Content-Type экземпляра *m*.

set_unixfrom *m*.set_unixfrom(*unixfrom*)

Задает строку конверта заголовка для экземпляра *m*. Аргумент *unixfrom* — полная строка конверта заголовка, включая начальный текст 'From ', но исключая завершающий символ '\n'.

walk *m*.walk()

Возвращает итератор, который совершает итерации по всем компонентам и подкомпонента экземпляра *m* для обхода дерева компонентов составного сообщения по принципу "сначала в глубину" (см. раздел "Рекурсия" в главе 3).

Модуль email.Generator

Модуль email.Generator предоставляет класс Generator, который можно использовать для генерации текстовой формы сообщения *m*. Иногда для этого достаточно вызовов *m*.as_string() и str(*m*), но класс Generator обеспечивает дополнительную гибкость. Вы можете создать экземпляр *g* класса Generator, передав конструктору один обязательный аргумент и два необязательных.

Generator class Generator(*outfp*, *mangle_from_*=False, *maxheaderlen*=78)

Аргумент *outfp* — это файл или файловый объект, который предоставляет метод write. Если аргумент *mangle_from_* имеет значение True, то экземпляр *g* присоединяет символ '>' в начале любой строки тела сообщения, которая начинается со слова 'From ' (с пробелом после него), с целью упрощения анализа текстовой формы сообщения. Экземпляр *g* обертывает каждую строку заголовка (при символах точки с запятой) в физические строки длиной не более чем *maxheaderlen* символов. Использование экземпляра *g* сводится к вызову g.flatten:

```
g.flatten(m, unixfrom=False)
```

В результате выполнения этого вызова экземпляр *m* помещается в виде текста в объект *outfp*, как если бы была выполнена (потребляющая больше памяти) инструкция outfp.write(*m*.as_string(*unixfrom*)).

Создание сообщений

Пакет email содержит модули, имена которых в версии v2 начинаются с префикса 'MIME', причем каждый модуль предоставляет подкласс класса Message, имя которого совпадает с именем модуля. В версии v3 указанные модули находятся в подпакете email.mime, а их имена записываются в нижнем регистре (например, email.mime.text в версии v3 вместо email.MIMEText в версии v2). Несмотря на то что классы в версиях v2 и v3 импортируются из разных модулей, они имеют одинаковые имена в обеих версиях.

Эти классы упрощают создание экземпляров Message различных MIME-типов. Перечень MIME-классов вместе с их описанием приведен ниже.

MIMEAudio class IMEAudio(*_audiodata*, *_subtype*=None, *_encoder*=None, ***_params*)_

Аргумент *audiodata* — это байтовая строка аудиоданных, упаковываемая в сообщение с MIME-типом 'audio/_subtype'. Если аргумент *_subtype* имеет значение None, то *_audiodata* должен позволять анализировать себя средствами модуля sndhdr стандартной библиотеки Python для определения его подтипа *subtype*; в противном случае класс MIMEAudio возбуждает исключение TypeError. Если аргумент *_encoder* имеет значение None, то класс MIMEAudio преобразует данные с использованием кодировки Base64, которая обычно является оптимальной. В противном случае аргумент *_encoder* должен быть вызываемым объектом с одним параметром *m*, представляющим создаваемое сообщение. Затем объект *_encoder* должен вызвать метод m.get_payload() для получения полезной нагрузки,

преобразовать ее, поместить обратно в преобразованном виде с помощью вызова *m*.set_payload и соответствующим образом установить заголовок 'Content-Transfer-Encoding' экземпляра *m*. Класс MIMEAudio передает словарь *_params*, содержащий имена именованных параметров и их значения, методу *m*.add_header для создания заголовка Content-Type экземпляра *m*.

MIMEBase class MIMEBase(*_maintype*, *_subtype*, ***_params*)

Базовый класс для всех MIME-классов, непосредственно расширяет класс Message. Его инстанциализация с помощью конструктора

```
m = MIMEBase(main,sub,**parms)
```

эквивалентна более длинной и менее удобной идиоме

```
m = Message()
m.add_header('Content-Type','{}/{}'.
  format(main,sub),**parms)
m.add_header('Mime-Version','1.0')
```

MIMEImage class MIMEImage(*_imagedata*, *_subtype*=None, *_encoder*=None, ***_params*)

Подобен MIMEAudio, но с основным типом 'image'. Использует модуль imghdr стандартной библиотеки Python для определения подтипа, если это необходимо.

MIMEMessage class MIMEMessage(*msg*, *_subtype*='rfc822')

Упаковывает объект *msg*, который должен быть экземпляром класса Message (или его подкласса), в качестве полезной нагрузки сообщения с MIME-типом 'message/_subtype'.

MIMEText class MIMEText(*_text*, *_subtype*='plain', *_charset*='us-ascii', *_encoder*=None)

Упаковывает текстовую строку *_text* в качестве полезной нагрузки сообщения с MIME-типом 'text/_subtype' и заданным набором символов. Если аргумент *_encoder* имеет значение None, то класс MIMEText не преобразует текст, что обычно является оптимальным вариантом. В противном случае аргумент *_encoder* должен быть вызываемым объектом с одним параметром *m*, представляющим создаваемое сообщение. Затем объект *_encoder* должен вызвать метод m.get_payload() для получения полезной нагрузки, преобразовать ее, поместить обратно в преобразованном виде с помощью вызова *m*.set_payload и соответствующим образом установить заголовок 'Content-Transfer-Encoding' экземпляра *m*.

Модуль email.encoders

Модуль email.encoders (в версии v3) содержит функции, которые принимают в качестве своего единственного аргумента сообщение *m*, не являющееся составным, преобразуют его полезную нагрузку и соответствующим образом устанавливают

заголовки сообщения. В версии v2 имя этого модуля начинается с прописной буквы: `email.Encoders`.

`encode_base64` `encode_base64(m)`

Использует стандарт кодирования Base64, оптимальный для преобразования произвольных двоичных данных.

`encode_noop` `encode_noop(m)`

Не выполняет никаких действий по отношению к полезной нагрузке и заголовкам *m*.

`encode_quopri` `encode_quopri(m)`

Использует кодировку Quoted Printable, оптимальную для текста, состоящего почти полностью из одних только символов ASCII (раздел "Модуль `quopri`").

`encode_7or8bit` `encode_7or8bit(m)`

Не выполняет никаких действий по отношению к полезной нагрузке сообщения *m* и устанавливает для заголовка `Content-Transfer-Encoding` значение `'8bit'`, если в любом байте полезной нагрузки *m* установлен старший бит. В противном случае для этого заголовка устанавливается значение `'7bit'`.

Модуль `email.utils`

Модуль `email.utils` (в версии v3) содержит несколько функций, используемых для обработки почты. В версии v2 имя этого модуля начинается с прописной буквы: `email.Utils`.

`formataddr` `formataddr(pair)`

Аргумент *pair* — это пара строк (*realname*, *email_address*). Функция `formataddr` возвращает строки для вставки в поля заголовков, такие как `To` и `Cc`. Если элемент *realname* имеет ложное значение (например, в виде пустой строки, `''`), то функция `formataddr` возвращает *email_address*.

`formatdate` `formatdate(timeval=None, localtime=False)` Аргумент *imeval* — это количество секунд, истекших с начала эпохи. Если *timeval* равен `None`, то функция `formatdate` использует текущее время. Если аргумент *localtime* равен `True`, то функция `formatdate` использует локальный часовой пояс; в противном случае она использует время UTC. Функция `formatdate` возвращает строку с заданным моментом времени, отформатированным в соответствии с документом RFC 2822.

`getaddresses` `getaddresses(L)`

Анализирует каждый элемент списка *L*, содержащего строки адресов, которые используются в таких полях заголовков, как `To` и `Cc`, и возвращает список пар строк (*name*, *email_address*). Если функции `getaddresses` не удается выполнить анализ элемента *L* как адреса, она использует пару (`None`, `None`) в качестве соответствующего элемента в возвращаемом списке.

mktime_tz `mktime_tz(t)`

Аргумент `t` — это кортеж из 10 элементов. Формат первых девяти элементов `t` совпадает с тем, который используется в модуле `time` (см. раздел "Модуль `time`" в главе 12). Элемент `t[-1]` представляет часовой пояс, выраженный в виде смещения в секундах от UTC (с противоположным знаком по сравнению со смещением `time.timezone`, определенным в документе RFC 2822). Если элемент `t[-1]` равен `None`, то `mktime_tz` использует локальный часовой пояс. Функция `mktime_tz` возвращает значение с плавающей точкой, которое выражает количество секунд, истекших с начала эпохи Unix, в UTC, и соответствует моменту времени `t`.

parseaddr `parseaddr(s)`

Анализирует строку `s`, которая содержит адрес, обычно указанный в таких заголовках, как `To` и `Cc`, и возвращает пару строк (*realname*, *email_address*). Если функции не удается выполнить анализ `s` как адреса, она возвращает пару пустых строк (`'', ''`).

parsedate `parsedate(s)`

Анализирует строку `s` в соответствии с правилами, изложенными в документе RFC 2822, и возвращает такой же кортеж `t` из девяти элементов, какой используется в модуле `time`, рассмотренном в разделе "Модуль `time`" в главе 12 (элементы `t[-3:]` несущественны). Функция `parsedate` также пытается проанализировать некоторые отклонения от требований RFC2822, используемые популярными почтовыми программами. Если функции `parsedate` не удается выполнить анализ `s`, то она возвращает значение `None`.

parsedate_tz `parsedate_tz(s)`

Подобна функции `parsedate`, но возвращает кортеж `t` из 10 элементов, в котором элемент `t[-1]` представляет часовой пояс, выраженный в виде смещения в секундах от UTC (с противоположным знаком по сравнению со смещением `time.timezone`, определенным в документе RFC 2822), как в аргументе, который принимает функция `mktime_tz`. Элементы `t[-4:-1]` несущественны. Если строка `s` не задает часовой пояс, то элемент `t[-1]` равен `None`.

quote `quote(s)`

Возвращает копию строки `s`, в которой каждая двойная кавычка (") заменяется символами `'\"'`, а каждый имеющийся символ обратной косой черты дублируется.

unquote `unquote(s)`

Возвращает копию строки `s`, в которой открывающие и закрывающие двойные кавычки ("), а также угловые скобки (<>) удаляются, если они заключают оставшуюся часть строки `s`.

Примеры использования пакета email

Пакет email облегчает как чтение, так и составление сообщений электронной почты и подобных им (но он не принимает участия в получении и передаче таких сообщений: для решения этих задач предназначены другие модули, рассмотренные в главе 19). Ниже приведен пример того, как использовать пакет email для чтения сообщения, которое может быть составным, и распаковки каждой его части в файл, сохраняемый в заданном каталоге.

```
import os, email

def unpack_mail(mail_file, dest_dir):
    ''' Получает файловый объект mail_file, открытый для чтения,
        и строку dest_dir, задающую путь к существующему каталогу,
        доступному для записи. Распаковывает каждую часть почтового
        сообщения из mail_file в файл, находящийся в dest_dir.
    '''
    with mail_file:
        msg = email.message_from_file(mail_file)
    for part_number, part in enumerate(msg.walk()):
        if part.get_content_maintype() == 'multipart':
            # Мы будем получать каждую составную часть далее
            # в цикле, поэтому 'multipart' не обрабатывается
        dest = part.get_filename()
        if dest is None: dest = part.get_param('name')
        if dest is None: dest = 'part-{}'.format(part_number)
        # На практике необходимо проверять, что dest
        # является допустимым именем файла для вашей ОС;
        # здесь эта проверка опущена
        with open(os.path.join(dest_dir, dest), 'wb') as f:
            f.write(part.get_payload(decode=True))
```

А вот пример выполнения противоположной задачи, заключающейся в упаковке всех файлов, находящихся непосредственно в заданном каталоге, в один файл, пригодный для отправки по электронной почте.

```
def pack_mail(source_dir, **headers):
    ''' Получает строку source_dir, задающую путь к существующему
        каталогу, доступному для чтения, и произвольные пары
        заголовков "имя/значение", переданных в виде именованных
        аргументов. Упаковывает все файлы, находящиеся в source_dir
        (предполагается, что это простые текстовые файлы), в почтовое
        сообщение, возвращаемое в виде MIME-строки.
    '''
    msg = email.Message.Message()
    for name, value in headers.items():
        msg[name] = value
    msg['Content-type'] = 'multipart/mixed'
    filenames = next(os.walk(source_dir))[-1]
    for filename in filenames:
```

```
        m = email.Message.Message()
        m.add_header('Content-type', 'text/plain', name=filename)
        with open(os.path.join(source_dir, filename), 'r') as f:
            m.set_payload(f.read())
        msg.attach(m)
    return msg.as_string()
```

Модули rfc822 и mimetools (v2)

Для обработки почтовых и аналогичных им сообщений лучше всего подходит пакет email. Однако некоторые другие модули (только в версии v2), рассмотренные в главах 19 и 20, используют экземпляры класса rfc822.Message или его подкласса mimetools.Message. В этом разделе обсуждается подмножество функциональности указанных классов, которое понадобится вам для эффективного использования упомянутых модулей в версии v2.

Экземпляр *m* класса Message в любом из этих модулей, входящих только в версию v2, является отображением с именами заголовков в качестве ключей и соответствующими строковыми значениями заголовков в качестве значений. Ключи и значения являются строками, причем ключи не чувствительны к регистру. Экземпляр *m* поддерживает все методы отображений, за исключением методов clear, copy, popitem и update. В методах get и setdefault в качестве значения по умолчанию используется '' вместо None. Кроме того, экземпляр *m* предоставляет вспомогательные методы (в частности, предназначенные для объединения операций получения значения заголовка и его анализа как даты или адреса). Для подобных целей мы рекомендуем применять функции модуля email.utils (см. раздел "Модуль email.utils"): используйте *m* просто как отображение.

Экземпляр *m* класса mimetools.Message содержит следующие дополнительные методы.

getmaintype *m*.getmaintype()

Возвращает тип основного содержимого экземпляра *m* из заголовка Content-Type в нижнем регистре. Если *m* не имеет заголовка Content-Type, то функция getmaintype возвращает строку 'text'.

getparam *m*.getparam(*param*)

Возвращает значение параметра *param* заголовка Content-Type экземпляра *m*.

getsubtype *m*.getsubtype()

Возвращает подтип содержимого экземпляра *m*, взятый из заголовка Content-Type, в нижнем регистре. Если *m* не имеет заголовка Content-Type, то функция getsubtype возвращает строку 'plain'.

gettype *m*.gettype()

Возвращает тип содержимого экземпляра *m*, взятый из заголовка Content-Type, в нижнем регистре. Если *m* не имеет заголовка Content-Type, то функция gettype возвращает строку 'text/plain'.

Преобразование двоичных данных в ASCII-текст

Некоторые средства передачи данных (например, электронная почта) поддерживают только ASCII-текст. Чтобы обеспечить передачу произвольных двоичных данных в таких сообщениях, данные необходимо преобразовывать в последовательность ASCII-символов. Стандартная библиотека Python предоставляет модули, поддерживающие стандарты кодирования Base64, Quoted Printable и UU.

Модуль base64

Модуль `base64` поддерживает кодировки, заданные в документе RFC 3548 как Base16, Base32 и Base64, каждая из которых обеспечивает компактный способ преобразования произвольных двоичных данных в ASCII-текст, не пытаясь получить конечный результат в удобочитаемой для человека форме. Модуль `base64` содержит 10 функций — 6 для Base64 и по 2 для Base32 и Base16, описание которых приведено ниже.

b64decode `b64decode(s, altchars=None, validate=False)`

Декодирует байтовую строку *s*, содержащую двоичные данные в формате Base64, и возвращает декодированную байтовую строку. Аргумент *altchars*, если его значение не равно `None`, должен быть байтовой строкой, содержащей по крайней мере два символа (избыточные символы игнорируются), задающих два нестандартных символа для использования вместо символов + и / (что может быть полезным для безопасной обработки URL-адресов и обеспечения нечувствительности данных в формате Base64 к файловой системе). Передача аргумента *validate* допускается только в версии v3: если его значение равно `True` и строка *s* содержит недопустимые для кодировки Base64 символы, то возбуждается исключение (по умолчанию такие байты просто игнорируются и отбрасываются). Исключение возбуждается также в случае несоблюдения правил стандарта Base64, касающихся дополнения строки *s* служебными символами.

b64encode `b64encode(s, altchars=None)`

Преобразует байтовую строку *s* и возвращает байтовую строку с соответствующими данными в формате Base64. Аргумент *altchars*, если его значение не равно `None`, должен быть байтовой строкой, содержащей по крайней мере два символа (избыточные символы игнорируются), задающих два нестандартных символа для использования вместо символов + и / (что может быть полезным для безопасной обработки URL-адресов и обеспечения нечувствительности данных в формате Base64 к файловой системе).

standard_b64decode `standard_b64decode(s)`

Аналогична функции `b64decode(s)`.

standard_b64encode `standard_b64encode(s)`

Аналогична функции `b64encode(s)`.

urlsafe_b64decode `urlsafe_b64decode(s)`

Аналогична функции `b64decode(s, '-_')`.

urlsafe_b64encode `urlsafe_b64encode(s)`

Аналогична функции `b64encode(s, '-_')`.

Ниже приведено описание аналогичных функций для кодировок Base16 и Base32.

b16decode `b16decode(s, casefold=False)`

Декодирует байтовую строку *s*, содержащую двоичные данные в формате Base16, и возвращает декодированную байтовую строку. Если аргумент *casefold* имеет значение `True`, то строка *s* может включать символы в нижнем регистре, которые в этом случае обрабатываются аналогично их эквивалентам в верхнем регистре. По умолчанию наличие в строке *s* символов в нижнем регистре приводит к возбуждению исключения.

b16encode `b16encode(s)`

Преобразует байтовую строку *s* и возвращает байтовую строку с соответствующими данными в формате Base16.

b32decode `b32decode(s, casefold=False, map01=None)`

Декодирует байтовую строку *s*, содержащую двоичные данные в формате Base32, и возвращает декодированную байтовую строку. Если аргумент *casefold* имеет значение `True`, то строка *s* может включать символы в нижнем регистре, которые в этом случае обрабатываются аналогично их эквивалентам в верхнем регистре. По умолчанию наличие в строке *s* символов в нижнем регистре приводит к возбуждению исключения. Если аргумент *map01* имеет значение `None`, то наличие символов `0` и `1` среди входных данных является недопустимым. Если же значение этого аргумента не равно `None`, то он должен быть односимвольной строкой, указывающей на то, в какую букву (`'l'` или `'L'`) должна отображаться цифра `1`, тогда как цифра `0` всегда отображается в букву `'O'` в верхнем регистре.

b32encode `b32encode(s)`

Преобразует байтовую строку *s* и возвращает байтовую строку с соответствующими данными в формате Base32.

В версии v3 данный модуль содержит также функции, позволяющие работать с нестандартными, но популярными кодировками Base85 и Ascii85, которые, несмотря на то что они не кодифицированы ни в одном документе RFC и не совместимы между собой, могут сократить объем выходных данных до 15% за счет использования расширенных алфавитов для преобразования байтовых строк. Более подробную информацию об этих функциях можно найти в онлайн-документации (`https://docs.python.org/3/library/base64.html#base64.a85encode`).

Модуль quopri

Модуль quopri поддерживает кодировку, определенную в документе RFC 1521 как *Quoted Printable* (QP). Кодировка QP может представлять любые двоичные данные в виде ASCII-текста, но в основном она предназначена для кодирования преимущественно текстовых данных с небольшим количеством символов с установленным старшим битом (т.е. символов, выходящих за пределы диапазона ASCII). Для таких данных кодировка позволяет получать компактные результаты в удобочитаемой для человека форме. Модуль quopri содержит следующие четыре функции.

decode decode(*infile*, *outfile*, *header*=False)

Читает двоичный файловый объект *infile* путем вызова метода *infile*.readline до тех пор, пока не будет достигнут конец файла (т.е. до тех пор, пока вызов *infile*.readline не вернет пустую строку), декодирует прочитанный ASCII-текст в кодировке QP и записывает декодированные данные в файловый объект *outfile*. Если аргумент *header* имеет значение True, то функция decode также декодирует символы подчеркивания (_) в пробелы (в соответствии с документом RFC 1522).

decodestring decodestring(*s*, *header*=False)

Декодирует байтовую строку *s*, содержащую ASCII-текст в кодировке QP, и возвращает байтовую строку с декодированными данными. Если аргумент *header* имеет значение True, то функция decodestring также декодирует символы подчеркивания (_) в пробелы.

encode encode(*infile*, *outfile*, *quotetabs*, *header*=False)

Читает данные из файлового объекта *infile* путем вызова метода *infile*.readline до тех пор, пока не будет достигнут конец файла (т.е. до тех пор, пока вызов *infile*.readline не вернет пустую строку), преобразует прочитанные данные с использованием кодировки QP и записывает преобразованный ASCII-текст в двоичный файловый объект *outfile*. Если аргумент *quotetabs* имеет значение True, то функция encode также кодирует пробелы и символы табуляции. Если аргумент *header* имеет значение True, то функция encode преобразует пробелы в символы подчеркивания (_).

encodestring encodestring(*s*, *quotetabs*=False, *header*=False)

Преобразует байтовую строку *s*, содержащую произвольные байты, и возвращает байтовую строку, содержащую ASCII-текст в кодировке QP. Если аргумент *quotetabs* имеет значение True, то функция encodestring также кодирует пробелы и символы табуляции. Если аргумент header имеет значение True, то функция encodestring преобразует пробелы в символы подчеркивания (_).

Модуль uu

Модуль uu поддерживает классический метод кодирования/декодирования данных *Unix-to-Unix* (UU), реализованный в Unix утилитами uuencode и uudecode.

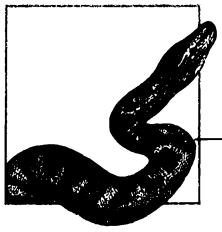
UU-данные начинаются со строки begin, представляющей имя файла и права доступа к преобразуемому файлу, и заканчиваются строкой end. Поэтому UU-кодирование позволяет встраивать закодированные данные в структурированный текст, в то время как кодирование Base64 рассчитывает на существование других индикаторов начала и конца закодированных данных. Модуль uu содержит две функции, описание которых приведено ниже.

decode decode(*infile*, *outfile*=None, *mode*=None)

Читает файловый объект *infile* путем вызова метода *infile*.readline до тех пор, пока не будет достигнут конец файла (т.е. до тех пор, пока вызов *infile*.readline не вернет пустую строку) или пока не встретится завершающая строка (строка 'end' в окружении произвольного количества пробелов). Функция decode декодирует прочитанный UU-текст и записывает декодированные данные в файловый объект *outfile*. Если аргумент *outfile* имеет значение None, то функция decode создает файл, указанный в строке begin UU-формата, с битами разрешений, заданными с помощью аргумента *mode* (с битами разрешений, заданными в строке begin, если аргумент *mode* равен None). В этом случае функция decode возбуждает исключение, если такой файл уже существует.

encode encode(*infile*, *outfile*, *name*='-', *mode*=0o666)

Читает файловый объект infile путем вызова метода infile.read (45 байтов за один раз — это количество байтов, которое UU кодирует в 60 символов каждой выходной строки) до тех пор, пока не будет достигнут конец файла (т.е. до тех пор, пока вызов infile.read не вернет пустую строку). Функция кодирует прочитанные данные в формат UU и записывает закодированный текст в файловый объект *outfile*. Функция *encode* также записывает UU-строку begin перед закодированным текстом и UU-строку end после закодированного текста. В строке begin функция encode указывает *name* в качестве имени файла и *mode* в качестве прав доступа к файлу.



22

Структурированный текст: HTML

В большинстве веб-документов используется *язык гипертекстовой разметки* — HTML (HyperText Markup Language). Здесь под *разметкой* подразумевается вставка в текстовый документ специальных лексем, известных как *теги* (дескрипторы), для структуризации текста. Теоретически HTML является реализацией общего стандарта SGML (Standard General Markup Language — стандартный обобщенный язык разметки). Но на практике во многих веб-документах HTML используется непоследовательно и недостаточно корректно. На протяжении многих лет в браузерной среде эволюционировали эвристические подходы, компенсирующие эти недостатки, но даже и это не спасает от некорректного отображения в браузере неправильно размеченных веб-страниц (не вините в этом браузер, особенно современный: в 9 случаях из 10 вина лежит на авторе веб-страницы).

HTML не способен ни на что большее, кроме как представлять документы в браузере[1]. Полное и точное извлечение содержащейся в документе информации посредством ее восстановления из тех данных, на основе которых обычно формируется представление документа, часто оказывается невыполнимой задачей. Ситуацию еще более усугубляет тот факт, что HTML пытался эволюционировать в более строгий стандарт XHTML, который аналогичен традиционному HTML, но определен в терминах XML и более точен по сравнению с HTML. Для обработки правильно оформленного XHTML-документа можно использовать инструменты, описанные в главе 23. Однако на момент выхода данной книги XHTML не сумел получить всеобщего признания и был вытеснен новейшей версией (не использующей XML) — HTML5.

Несмотря на все трудности, из HTML-документов нередко удается извлечь по крайней мере часть полезной информации. Стандартная библиотека Python (в версиях v2 и v3) пытается облегчить эту задачу, предоставляя модули `sgmllib`,

[1] За исключением, возможно, последней версии языка (HTML5) при ее надлежащем применении.

htmllib и HTMLParser в версии v2 и пакет html в версии v3, которые предназначены для анализа HTML-документов как в целях их представления, так и, что более типично, в целях извлечения информации. Однако если вы имеете дело с подпорченными веб-страницами, то вашей последней надеждой и спасителем может оказаться сторонний пакет BeautifulSoup (раздел "Сторонний пакет BeautifulSoup"). В этой главе мы уделим основное внимание пакету BeautifulSoup, игнорируя большинство средств стандартной библиотеки Python, предлагающих "конкурентные" возможности.

К числу распространенных задач относятся генерация HTML-разметки и внедрение кода Python в HTML. Стандартная библиотека Python не поддерживает ни первое, ни второе, но можно воспользоваться средствами форматирования строк Python, а также сторонними модулями. Пакет BeautifulSoup позволяет изменять HTML-деревья документов (поэтому вы можете построить такое дерево чисто программным путем, причем даже с нуля). Альтернативный подход заключается в использовании шаблонов, предлагаемых, например, сторонним модулем jinja2 (раздел "Пакет jinja2").

Модуль html.entities (v2: htmlentitydefs)

Модуль html.entities стандартной библиотеки Python (в версии v2 он называется htmlentitydefs) предоставляет несколько атрибутов, все из которых являются отображениями. Эти атрибуты будут вам полезны независимо от того, какой общий подход вы выберете для анализа, редактирования или генерации HTML-разметки, включая пакет BeautifulSoup.

codepoint2name
: Отображает кодовые точки Unicode в имена сущностей HTML. Например, entities.codepoint2name[228] — это 'auml', поскольку символ 228 Unicode, ä (буква a с умлаутом), кодируется в HTML как 'ä'.

entitydefs
: В версии v3 — отображение имен сущностей HTML в эквивалентные односимвольные строки Unicode. Например, entities.entitydefs['auml'] — это 'ä', а entities.entitydefs['sigma'] — это 'σ'. В версии v2 эта эквивалентность ограничена символами из набора Latin-1 (ISO-8859-1), тогда как для остальных символов используются ссылки HTML. Поэтому в версии v2 htmlentitydefs.entitydefs['auml'] — это '\xe4', а htmlentitydefs.entitydefs['sigma'] — это 'σ'.

html5 *(только в версии v3)*
: Предусмотренный только в версии v3 атрибут html5 — это отображение именованных ссылок на символы HTML5 в эквивалентные односимвольные строки. Например, entities.html5['gt;'] — это '>'. Замыкающий символ "точка с запятой" имеет значение — некоторые, хотя далеко не все, именованные ссылки на символы HTML5 можно использовать и без него, и в таком случае

в отображении `entities.html5` присутствуют оба ключа (как с символом `;`, так и без него).

`name2codepoint`
: Отображение имен сущностей HTML в кодовые точки Unicode. Например, `entities.name2codepoint['auml']` — это 228.

Сторонний пакет BeautifulSoup

Пакет BeautifulSoup позволяет анализировать HTML-документ, даже если он плохо сформирован. BeautifulSoup компенсирует вероятные нарушения правил разметки HTML, используя простую эвристику, и на удивление часто преуспевает в решении данной задачи. Мы настоятельно рекомендуем использовать версию 4 BeautifulSoup, известную как `bs4`, которая одинаково хорошо поддерживает обе версии Python, v2 и v3. В этой книге мы рассматриваем только `bs4` (в частности, версию 4.5).

Установка и импорт BeautifulSoup

Модуль BeautifulSoup можно установить, например, из командной строки с помощью команды **`pip install beautifulsoup4`**, но при его импорте следует использовать инструкцию `import bs4`. Если у вас установлены две версии Python, v2 и v3, то вам, возможно, понадобится явно использовать `pip3` для установки модуля в версию v3 и/или `pip2` для установки в версию v2.

Класс `BeautifulSoup`

Модуль `bs4` содержит класс `BeautifulSoup`, который инстанциализируется вызовом конструктора с одним или двумя аргументами. Первый из них, *`htmltext`*, — это либо файловый объект (содержимое которого читается для получения разметки HTML, подлежащей синтаксическому анализу), либо строка (представляющая анализируемый текст), а второй — необязательный аргумент *`parser`*.

Какой парсер используется в BeautifulSoup

Если вы не передаете аргумент *`parser`*, BeautifulSoup "осматривается", пытаясь выбрать наилучший из доступных синтаксических анализаторов (в подобном случае вы можете получить предупреждение). Если у вас не установлен никакой другой анализатор, то BeautifulSoup по умолчанию использует анализатор `html.parser` из стандартной библиотеки Python (чтобы задать его явно, укажите строку `'html.parser'` — имя данного модуля в версии v3; эта же строка сработает и в версии v2, в которой используется модуль `HTMLParser`). Чтобы расширить возможности управления и не зависеть, например, от различий между анализаторами, упомянутыми в документации BeautifulSoup (`https://www.crummy.com/software/BeautifulSoup/bs4/doc/`), передайте имя библиотеки того анализатора, который хотите использовать, в качестве второго аргумента конструктору при инстанциализации

BeautifulSoup. Если не оговорено иное, в следующих примерах используется анализатор `html.parser`, заданный в Python по умолчанию.

К примеру, если вы установили сторонний пакет `html5lib` (чтобы анализировать HTML-код так же, как это делают все известные браузеры, хотя и медленнее), то можете использовать следующий вызов:

```
soup = bs4.BeautifulSoup(thedoc, 'html5lib')
```

Если вы передаете `'xml'` в качестве второго аргумента, то у вас должен быть установлен[2] сторонний пакет `lxml`, о котором говорится в разделе "Модуль `ElementTree`" главы 23, и тогда BeautifulSoup анализирует документ как XML, а не как HTML. В этом случае атрибут `is_xml` объекта *soup* имеет значение *True*, в противном случае — значение `False`. (Если вы установили модуль `lxml`, то можете использовать его и для анализа HTML, передав строку `'lxml'` конструктору в качестве второго аргумента).

```
>>> import bs4
>>> s = bs4.BeautifulSoup('<p>hello', 'html.parser')
>>> sx = bs4.BeautifulSoup('<p>hello', 'xml')
>>> sl = bs4.BeautifulSoup('<p>hello', 'lxml')
>>> s5 = bs4.BeautifulSoup('<p>hello', 'html5lib')
>>> print(s, s.is_xml)
<p>hello</p> False
>>> print(sx, sx.is_xml)
<?xml version="1.0" encoding="utf-8"?>
<p>hello</p> True
>>> print(sl, sl.is_xml)
<html><body><p>hello</p></body></html> False
>>> print(s5, s5.is_xml)
<html><head></head><body><p>hello</p></body></html> False
```

Различия в поведении парсеров в отношении коррекции HTML-ввода

В данном примере построитель дерева `'html.parser'` вставляет конечный тег `</p>`, отсутствующий в HTML-вводе. Также показано, что другие построители идут дальше в отношении исправления некорректного HTML-ввода, добавляя требуемые теги, такие как `<body>` и `<html>`, в зависимости от парсера.

BeautifulSoup, Unicode и кодировка

BeautifulSoup использует Unicode на основании выводов или предположений о кодировке[3], когда на вход подается байтовая строка или двоичный файл. Для вы-

[2] Документация BeautifulSoup предоставляет подробную информацию относительно установки различных анализаторов.

[3] В соответствии с объяснениями, приведенными в документации BeautifulSoup (`https://www.crummy.com/software/BeautifulSoup/bs4/doc/`), где также показаны различные способы корректировки или переопределения догадок BeautifulSoup.

вода метод prettify возвращает дерево в строковом представлении Unicode, включая теги с атрибутами, дополнительные пробелы и символы новой строки для отображения вложенных структур посредством выделения элементов отступами. Если вы хотите, чтобы вместо этого возвращалась байтовая строка, соответствующая определенной кодировке, передайте методу prettify название кодировки в качестве аргумента. Если не хотите "приукрашивать" ("prettify") результат, используйте метод encode для получения байтовой строки и метод decode для получения строки Unicode. Например, в версии v3:

```
>>> s = bs4.BeautifulSoup('<p>hello', 'html.parser')
>>> print(s.prettify())
<p>
hello
</p>
>>> print(s.decode())
<p>hello</p>
>>> print(s.encode())
b'<p>hello</p>'
```

(В версии v2 последняя строка имела бы вид hello , поскольку в этой версии байтовые строки не требуют использования нотации b'...'.)

Навигационные классы модуля bs4

Экземпляр *b* класса BeautifulSoup предоставляет атрибуты и методы, которые обеспечивают навигацию по анализируемому дереву HTML, возвращая экземпляры классов Tag и NavigableString (а также подклассы NavigableString: CData, Comment, Declaration, Doctype и ProcessingInstruction, отличающиеся лишь способом их генерации при выводе).

Терминология навигационных классов

Когда мы говорим "экземпляры NavigableString", мы включаем сюда экземпляры любого подкласса этого класса; когда говорим "экземпляры Tag" — включаем сюда экземпляры BeautifulSoup, поскольку последний является подклассом класса Tag. Мы также называем экземпляры навигационных классов *элементами* или *узлами* дерева.

Каждый экземпляр "навигационного" (предоставляющего доступный для обхода узел) класса позволяет продолжить обход дерева или углубиться в структуру узла для получения дополнительной информации с использованием практически того же набора навигационных атрибутов и методов поиска, который предоставляется самим экземпляром *b*. Однако существуют и некоторые отличия: экземпляры Tag могут иметь HTML-атрибуты и узлы в дереве HTML, тогда как экземпляры NavigableString не могут иметь их (экземпляры NavigableString всегда имеют

одну текстовую строку, родительский элемент Tag и нуль или большее количество сестринских элементов, т.е. других дочерних элементов того же родительского тега).

Все экземпляры навигационных классов имеют атрибут name: это строка тега для экземпляров Tag, строка '[document]' для экземпляров BeautifulSoup и значение None для экземпляров NavigableString.

Экземпляры класса Tag позволяют получать доступ к своим HTML-атрибутам по индексу или получать сразу все атрибуты в виде словаря посредством атрибута .attrs экземпляра.

Индексация экземпляров Tag

Если *t* — экземпляр Tag, то конструкция наподобие *t*['foo'] осуществляет поиск HTML-атрибута foo среди HTML-атрибутов *t* и возвращает строку для HTML-атрибута foo. Если *t* не имеет HTML-атрибута foo, то обращение *t*['foo'] возбуждает исключение KeyError. Точно так же, как в случае словаря, чтобы получить для отсутствующего HTML-атрибута значение аргумента *default* вместо исключения, вызовите метод *t*.get('foo', *default*=None).

Некоторые атрибуты, такие как class, определены в стандарте HTML таким образом, что им разрешается иметь несколько значений (например, <body class="foo bar">...</body>). В подобных случаях индексирование возвращает список значений. Например, значением soup.body['class'] будет ['foo', 'bar']. (Напомним, что в случае отсутствия атрибута вы получите исключение KeyError. Чтобы получить вместо исключения значение по умолчанию, используйте не индексирование, а метод get.)

Для получения словаря, отображающего имена атрибутов в значения (или в некоторых случаях, определенных стандартом HTML, в список значений), используйте атрибут *t*.attrs.

```
>>> s = bs4.BeautifulSoup('<p foo="bar" class="ic">baz')
>>> s.get('foo')
>>> s.p.get('foo')
'bar'
>>> s.p.attrs
{'foo': 'bar', 'class': ['ic']}
```

Как проверить, имеет ли экземпляр Tag определенный атрибут

Не используйте инструкцию if 'foo' in t: для того, чтобы проверить, есть ли среди HTML-атрибутов экземпляра *t* класса Tag атрибут 'foo' — оператор in, применяемый к экземплярам Tag, выполняет поиск среди дочерних элементов Tag, а не среди его атрибутов. Вместо этого используйте инструкцию if 'foo' in t.attrs: или if t.has_attr('foo'):.

В процессе работы с экземпляром NavigableString часто возникает необходимость в получении доступа к фактической текстовой строке, которую он содержит. Если у вас имеется экземпляр Tag, вам может потребоваться доступ к содержащейся в нем уникальной строке или, если таких строк несколько, ко всем строкам, возможно, без окружающих их пробелов. Перейдем к рассмотрению того, как это делается.

Получение фактической строки

Если имеется экземпляр *s* класса NavigableString и вам нужно сохранить или обработать его текст без дополнительной навигации по нему, вызовите функцию str(*s*) (или, в версии v2, unicode(*s*)). Вы также можете использовать вызов метода *s*.encode(*codec*='utf8') для получения байтовой строки и метода *s*.decode() для получения текста (Unicode). Это даст вам фактическую строку без каких-либо ссылок на дерево BeautifulSoup, затрудняющих сборку мусора (экземпляр *s* поддерживает все методы строк Unicode, поэтому вызывайте их непосредственно, если они делают все, что вам нужно).

Если существует экземпляр *t* класса Tag, вы сможете получить единственный содержащийся в нем экземпляр NavigableString с помощью вызова *t*.string (поэтому вызов *t*.string.decode() мог бы вернуть фактический текст, который вам нужен). Вызов *t*.string работает лишь в том случае, если *t* имеет единственный дочерний элемент, являющийся экземпляром NavigableString или экземпляром Tag, единственным дочерним элементом которого является экземпляр NavigableString. В противном случае метод *t*.string возвращает значение None.

В качестве итератора по *всем* содержащимся (допускающим обход в качестве узлов дерева HTML) строкам используйте вызов *t*.strings (вызов ''.join(*t*.strings) мог бы дать вам желаемую строку). Чтобы игнорировать пробелы, окружающие каждую содержащуюся строку, используйте итератор t.stripped_strings (он также пропускает строки, полностью состоящие из пробелов).

Альтернативный способ заключается в вызове метода *t*.get_text() — он возвращает единственную строку (Unicode) со всем текстом, содержащимся в потомках экземпляра *t*, в порядке обхода узлов дерева (эквивалентный ему способ — обращение к атрибуту *t*.text). Вы также можете передать в качестве единственного позиционного аргумента строку-разделитель (по умолчанию — пустая строка, ''). Передача именованного параметра strip=True обеспечивает удаление пробелов, окружающих каждую строку, и пропуск всех строк, состоящих исключительно из пробелов.

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> print(soup.p.string)
None
>>> print(soup.p.b.string)
bold
>>> print(soup.get_text())
Plain bold
>>> print(soup.text)
```

```
Plain bold
>>> print(soup.get_text(strip=True))
Plainbold
```

Простейшим и наиболее элегантным способом перемещения вниз по HTML-дереву или поддереву в `bs4` является использование синтаксиса ссылок на атрибуты в Python (при условии, что каждый из указанных вами тегов является уникальным, или если вас интересует первый встретившийся тег с данным именем на каждом иерархическом уровне).

Ссылки на атрибуты экземпляров `BeautifulSoup` и `Tag`

Если существует экземпляр *t* класса `Tag`, то конструкция наподобие `t.foo.bar` осуществляет поиск первого вхождения тега `foo` среди потомков экземпляра *t*, получает для него экземпляр *ti* класса `Tag`, ищет первое вхождение тега `bar` среди его потомков и возвращает экземпляр `Tag` для тега `bar`.

Описанный подход обеспечивает компактный и элегантный способ навигации вниз по дереву узлов, если вам известно, что среди потомков данного экземпляра имеется единственное вхождение интересующего вас тега, или если вас удовлетворит первое найденное вхождение данного тега. Но будьте бдительны: если на каком-либо уровне просмотра не удается найти искомый тег, то ссылка на атрибут имеет значение `None`, и любая попытка ссылки на этот атрибут приведет к возбуждению исключения `AttributeError`.

Остерегайтесь опечаток в ссылках на атрибуты экземпляров `Tag`

Вследствие описанного поведения BeautifulSoup любые опечатки, которые вы можете допустить в ссылках на атрибуты экземпляра `Tag`, приведут к получению этими ссылками значения `None`, а не к возбуждению исключения `AttributeError`. Поэтому будьте особенно внимательны!

Модуль `bs4` предлагает и более общие способы навигации вниз по узлам иерархического дерева, а также по горизонтали. В частности, экземпляр каждого навигационного класса имеет атрибуты, идентифицирующие один родственный узел или, в случае множественного числа, итератор, возвращающий все узлы той же степени родства.

Атрибуты `contents`, `children` и `descendants`

Для экземпляра *t* класса `Tag` можно получить список всех его дочерних элементов `t.contents` или итератор по всем дочерним элементам `t.children`. Чтобы получить итератор по всем *потомкам* (дочерним элементам, дочерним элементам дочерних элементов и т.д.), используйте атрибут `t.descendants`.

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> list(t.name for t in soup.p.children)
```

```
[None, 'b']
>>> list(t.name for t in soup.p.descendants)
[None, 'b', None]
```

Имена None соответствуют узлам NavigableString. Лишь первый из них является дочерним узлом тега *p*, но оба являются потомками данного тега.

Атрибуты parent и parents

Родительским узлом экземпляра *n* любого навигационного класса является *n*.parent. Итератором по всем предкам, располагающимся выше в структуре дерева, является *n*.parents. Это относится и к экземплярам NavigableString, поскольку они также могут иметь родительские узлы. Атрибут *b*.parent экземпляра *b* класса BeautifulSoup имеет значение None, поэтому b.parents представляет пустой итератор.

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> soup.b.parent.name
'p'
```

Атрибуты next_sibling, previous_sibling, next_siblings и previous_siblings

Непосредственным левым сестринским (имеющим того же родителя) элементом экземпляра *n* любого навигационного класса является n.previous_sibling, а непосредственным правым сестринским элементом — *n*.next_sibling. Любой из этих атрибутов может иметь значение None, если для *n* отсутствует соответствующий сестринский элемент. Итератором по всем левым сестринским элементам, выполняющим итерации в направлении справа налево по узлам дерева, является *n*.previous_siblings. Итератором по всем правым сестринским элементам, выполняющим итерации в направлении слева направо по узлам дерева, является *n*.next_siblings (любой из этих итераторов или оба одновременно могут быть пустыми). Это относится и к экземплярам NavigableString, поскольку они также могут иметь сестринские узлы. Для экземпляра *b* класса BeautifulSoup оба атрибута, *b*.previous_sibling и *b*.next_sibling, имеют значение None, и оба его итератора по сестринским элементам являются пустыми.

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> soup.b.previous_sibling, soup.b.next_sibling
('Plain ', None)
```

Атрибуты next_element, previous_element, next_elements и previous_elements

Для экземпляра *n* любого навигационного класса анализируемым узлом, непосредственно предшествующим ему, является *n*.previous_element, а анализируемым узлом, непосредственно следующим за ним, является *n*.next_element. Любой из этих узлов или оба одновременно могут иметь значение None, если *n* является первым или

последним анализируемым узлом соответственно. Итератором по всем предыдущим элементам в обратном направлении по дереву узлов является *n*.previous_elements. Итератором по всем последующим элементам в прямом направлении по дереву узлов является *n*.next_elements (любой из этих итераторов или оба одновременно могут быть пустыми). Экземпляры класса NavigableString также имеют эти атрибуты. Для экземпляра *b* класса BeautifulSoup атрибуты *b*.previous_element и *b*.next_element имеют значение None, и оба итератора по его элементам являются пустыми.

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> soup.b.previous_element, soup.b.next_element
('Plain ', 'bold')
```

Как показывает предыдущий пример, тег b не имеет следующего сестринского элемента next_sibling (поскольку он является последним дочерним элементом своего родителя). Однако в примере также показано, что у тега b имеется следующий элемент next_element — анализируемый узел, следующий непосредственно за данным тегом (которым в данном случае является содержащаяся в нем строка 'bold').

Методы find... модуля bs4 (поисковые методы)

Каждый из навигационных классов модуля bs4 предлагает несколько методов, имена которых начинаются с префикса find. Это так называемые *поисковые методы*. Они предназначены для определения местонахождения узлов дерева, удовлетворяющих заданным условиям.

Поисковые методы существуют парами — один из методов каждой пары обходит все соответствующие узлы дерева и возвращает список узлов, удовлетворяющих заданным условиям, тогда как второй прекращает обход и возвращает единственный узел, удовлетворяющий заданным условиям, как только его удается найти (или значение None, если такой узел не найден). Поэтому вызов последнего из упомянутых методов эквивалентен вызову первого с аргументом limit=1 и с индексацией результирующего одноэлементного списка для получения его единственного элемента, но работает немного быстрее и более элегантен.

Например, для любого экземпляра *t* класса Tag и любой группы позиционных и именованных аргументов, представленных символами ..., всегда выполняется следующая эквивалентность:

```
just_one = t.find(...)
other_way_list = t.find_all(..., limit=1)
other_way = other_way_list[0] if other_way_list else None
assert just_one == other_way
```

Описание вышеупомянутых пар поисковых методов приводится ниже.

find, find_all

```
b.find(...)
b.find_all(...)
```

Осуществляют поиск *потомков* `b`, за исключением случая, когда вы передаете `recursive=False` в качестве одного из именованных аргументов (эта возможность доступна только для данных двух методов и недоступна для любого другого поискового метода), и в этом случае метод ищет только *дочерние* элементы `b`. Эти методы недоступны для экземпляров класса `NavigableString`, поскольку они не имеют потомков. Все остальные поисковые методы доступны для экземпляров классов `Tag` и `NavigableString`.

Поскольку необходимость в использовании метода `find_all` возникает довольно часто, модуль `bs4` предлагает элегантную сокращенную форму его вызова: вызвать тег — это все равно что вызвать его метод `find_all`. Таким образом, вызов `b(...)` — это то же самое, что вызов `b.find_all(...)`.

Другое полезное сокращение, о котором уже упоминалось в разделе "Ссылки на атрибуты экземпляров `BeautifulSoup` и `Tag`": выражение `b.foo.bar` эквивалентно вызову `b.find('foo').find('bar')`.

find_next, find_all_next

```
b.find_next(...)
b.find_all_next(...)
```

Осуществляют поиск элементов, следующих за `b`.

find_next_sibling, find_next_siblings

```
b.find_next_sibling(...)
b.find_next_siblings(...)
```

Осуществляют поиск сестринских элементов `b`, следующих за ним.

find_parent, find_parents

```
b.find_parent(...)
b.find_parents(...)
```

Осуществляют поиск родительских элементов `b`.

find_previous, find_all_previous

```
b.find_previous(...)
b.find_all_previous(...)
```

Осуществляют поиск элементов, предшествующих `b`.

find_previous_sibling, find_previous_siblings

```
b.find_previous_sibling(...)
b.find_previous_siblings(...)
```

Осуществляют поиск сестринских элементов `b`, предшествующих ему.

Аргументы поисковых методов

Каждый из поисковых методов имеет три необязательных аргумента: *name*, *attrs* и *string*. Аргументы *name* и *string* — *фильтры*, описанные далее в этом разделе; аргумент *attrs* — словарь, также описанный далее. Кроме того, методы `find` и `find_all` (но не другие поисковые методы) могут вызываться с необязательным аргументом *recursive*=`False`, позволяющим ограничить поиск потомков только дочерними элементами.

Любой поисковый метод, возвращающий список (т.е. такой, имя которого записывается во множественном числе или начинается с префикса `find_all`), имеет необязательный именованный аргумент `limit`, значением которого, если он передается, является целое число, определяющее максимальную длину возвращаемого списка.

Вслед за этими необязательными аргументами каждый поисковый метод может иметь любое количество произвольных именованных аргументов, именем каждого из которых может быть любой идентификатор (за исключением имени одного из специальных аргументов поисковых методов), а значением — фильтр.

Фильтры

Фильтр применяется к *целевому объекту*, которым может быть имя тега (передаваемое в качестве аргумента *name*), атрибут `string` экземпляра `Tag` или текстовое содержимое экземпляра `NavigableString` (передаваемые в качестве аргумента *string*) либо какой-нибудь атрибут экземпляра `Tag` (передаваемый в качестве значения именованного аргумента или в аргументе *attrs*). В качестве фильтров могут выступать следующие объекты.

Строка Unicode
: Фильтр срабатывает, если строка в точности совпадает с целевым объектом.

Байтовая строка
: Декодируется в строку Unicode с использованием кодировки `utf8`, и фильтр срабатывает, если результирующая строка Unicode в точности совпадает с целевым объектом.

Объект регулярного выражения (сокращенно — РВ, получаемый в результате вызова `re.compile`, описанного в разделе "Регулярные выражения и модуль `re`" главы 9)
: Фильтр срабатывает, если успешно выполняется поисковый метод РВ, вызванный с целевым объектом в качестве аргумента.

Список строк
: Фильтр срабатывает, если любая из строк в точности совпадает с целевым объектом (строки, являющиеся байтовыми, декодируются в строки Unicode с использованием кодировки `utf8`).

Объект функции

Фильтр срабатывает, если функция, вызванная с экземпляром `Tag` или `NavigableString` в качестве аргумента, возвращает значение `True`.

Значение `True`

Фильтр всегда срабатывает.

Синонимом фразы "фильтр срабатывает" могла бы служить фраза "целевой объект совпадает с фильтром".

Каждый поисковый метод находит соответствующие узлы, которые совпадают со всеми фильтрами (т.е. для каждого узла-кандидата ко всем его фильтрам неявно применяется логическая операция И).

name

Чтобы найти все экземпляры `Tag`, имя которых совпадает с фильтром, передайте фильтр поисковому методу в качестве первого позиционного аргумента или именованного аргумента `name=`*`filter`*.

```
soup.find_all('b') # или soup.find_all(name='b')
# возвращает все экземпляры тега 'b' в документе
soup.find_all(['b', 'bah'])
# возвращает все экземпляры тегов 'b' и 'bah' в документе
soup.find_all(re.compile(r'^b'))
# возвращает все экземпляры тегов в документе, начинающихся с 'b'
soup.find_all(re.compile(r'bah'))
# возвращает все экземпляры тегов в документе, включающие
# строку 'bah'
def child_of_foo(tag):
  return tag.parent == 'foo'
soup.find_all(name=child_of_foo)
# возвращает все экземпляры тегов, родительским элементом которых
# является элемент с именем 'foo'
```

string

Чтобы найти узлы тегов, в которых текст, содержащийся в атрибуте `.string`, совпадает с фильтром, или узлы экземпляров `NavigableString`, текст которых совпадает с фильтром, передайте фильтр с помощью аргумента `string=`*`filter`*.

```
soup.find_all(string='foo')
# возвращает все экземпляры NavigableString с текстом 'foo'
soup.find_all('b', string='foo')
# возвращает все экземпляры дескриптора 'b', атрибут .string
# которых содержит текст 'foo'
```

attrs

Чтобы найти узлы тегов с атрибутами, значения которых совпадают с фильтрами, используйте словарь *d* с именами атрибутов в качестве ключей и фильтрами в качестве соответствующих значений. Затем передайте его поисковому методу в качестве второго позиционного аргумента или именованного аргумента `attrs=`*d*.

Существует один специальный случай, когда в качестве значения в словаре *d* вместо фильтра используется значение `None`. В этой ситуации осуществляется поиск узлов, в которых соответствующий атрибут отсутствует.

В специальном случае, когда значением именованного аргумента `attrs` является не словарь, а фильтр *f*, поиск осуществляется так, как если бы аргумент `attrs` имел значение `{'class': `*f*`}`. (Эта сокращенная форма удобна по той причине, что поиск тегов, принадлежащих определенному CSS-классу, является довольно распространенной задачей.)

Вы не сможете совместить обе эти специальные возможности в одном вызове: для поиска тегов, не принадлежащих определенному CSS-классу, требуется явно передать аргумент `attrs={'class': None}` (т.е. использовать первую специальную возможность, но не одновременно со второй).

```
soup.find_all('b', {'foo': True, 'bar': None})
# возвращает все экземпляры тега 'b', имеющие атрибут 'foo',
# но не имеющие атрибута 'bar'
```

Соответствие тегам, принадлежащим одновременно нескольким классам CSS

В отличие от большинства атрибутов, атрибут `'class'` тега может иметь несколько значений. В HTML эти значения записываются в одной строке через пробел (например, `'<p class='foo bar baz'>...'`), а в `bs4` — в виде списка строк (например, значение *t*`['class']` является списком `['foo', 'bar', 'baz']`).

В случае использования фильтрации по CSS-классу в любом из поисковых методов фильтр совпадает с тегом, если он совпадает с любым из нескольких классов CSS такого тега.

Для поиска тегов по соответствию нескольким CSS-классам можно написать пользовательскую функцию и передать ее в качестве фильтра поисковому методу или же, если вам не нужна дополнительная функциональность поисковых методов, можете обойтись без них и использовать метод *t*`.select`, рассмотренный в разделе "Селекторы CSS в модуле `bs4`", работая с синтаксисом селекторов CSS.

Другие именованные аргументы поисковых методов

Именованные аргументы с именами сверх тех, которые известны поисковым методам, передаются для дополнения ограничений, указанных в аргументе `attrs`, если таковые имеются. Например, вызов поискового метода с аргументом *foo*`=`*bar* аналогичен вызову с `attrs={'`*foo*`': `*bar*`}`.

Селекторы CSS в модуле bs4

В модуле bs4 теги предоставляют методы select и select_one, которые примерно эквивалентны методам find_all и find, но принимают в качестве единственного аргумента строку, являющуюся CSS-селектором (https://www.w3schools.com/cssref/css_selectors.asp), и возвращают список узлов тегов, удовлетворяющих этому селектору, или, соответственно, первый такой узел.

Модуль bs4 предоставляет лишь подмножество богатой функциональности селекторов CSS, которые в этой книге не обсуждаются. (Для получения полной информации, касающейся CSS, рекомендуем обратиться к книге *CSS: полный справочник, 4-е издание* Эрика Мейера.) В большинстве случаев поисковые методы, рассмотренные в разделе "Методы find... модуля bs4 (поисковые методы)", предлагают лучшие решения, однако в некоторых особых случаях вызов метода select может избавить вас от небольших хлопот, связанных с написанием пользовательской функции фильтра.

```
def foo_child_of_bar(t):
    return t.name=='foo' and t.parent and t.parent.name=='bar'
soup(foo_child_of_bar)
# возвращает теги с именем 'foo', являющиеся дочерними элементами
# тегов с именем 'bar'
soup.select('foo < bar')
# точный эквивалент, не требующий использования функции фильтра
```

Пример анализа HTML-кода с помощью BeautifulSoup

В следующем примере модуль bs4 используется для решения типичной задачи: извлечение веб-страницы, анализ страницы и вывод содержащихся в ней гиперссылок HTTP. Ниже приведен код для версии v2.

```
import urllib, urlparse, bs4

f = urllib.urlopen('http://www.python.org')
b = bs4.BeautifulSoup(f)

seen = set()
for anchor in b('a'):
    url = anchor.get('href')
    if url is None or url in seen: continue
    seen.add(url)
    pieces = urlparse.urlparse(url)
    if pieces[0]=='http':
        print(urlparse.urlunparse(pieces))
```

В версии v3 код остается тем же, но в этом случае функция urlopen находится в модуле urllib.request, а функции urlparse и urlunparse — в модуле urllib.parse, а не в модулях urllib и urlparse соответственно, как в версии v2 (см. раздел

"Доступ к ресурсам с помощью URL" в главе 19). Это проблемы, не связанные с самим пакетом BeautifulSoup.

В рассматриваемом примере сначала вызывается экземпляр класса `bs4.BeautifulSoup` (эквивалентно вызову его метода `find_all`) для получения всех экземпляров определенного тега (в данном случае тега `'<a>'`), а затем — метод `get` экземпляров класса `Tag` для получения значения атрибута (в данном случае `'href'`) или значения `None`, если нужный атрибут отсутствует.

Генерация HTML-кода

В поставку Python не входят ни инструменты, специально предназначенные для генерации HTML-кода, ни инструменты, обеспечивающие внедрение кода на языке Python непосредственно в HTML-страницы. Использование *шаблонов* (раздел "Работа с шаблонами") позволяет упростить разработку и сопровождение приложений за счет разделения логики и представления. Альтернативный подход заключается в том, чтобы использовать модуль `bs4` для создания HTML-документов в коде на языке Python путем изменения минимальной начальной заготовки документа. Поскольку эти изменения основываются на *парсинге* некоторой начальной HTML-разметки средствами модуля `bs4`, то результирующий вывод зависит от выбора используемого для этих целей парсера (см. раздел "Какой парсер используется в BeautifulSoup").

Изменение и создание HTML-кода с помощью модуля `bs4`

Изменить имя тега экземпляра `t` класса `Tag` можно путем присваивания соответствующего значения атрибуту `t.name`. Вы можете изменять атрибуты экземпляра `t`, работая с ним как с отображением: используйте индексацию для добавления (изменения) атрибута или оператор `del` для его удаления. Например, инструкция `del t['foo']` удаляет атрибут `foo`. Если вы присвоите некоторое строковое значение `str` атрибуту `t.string`, то предыдущее значение атрибута `t.contents` (теги и/или строки — все поддерево потомков экземпляра `t`) отбрасывается и заменяется новым экземпляром `NavigableString`, текстовым содержимым которого является строка `str`.

Если `s` — экземпляр `NavigableString`, то можно заменить его текстовое содержимое: вызов `s.replace_with('other')` заменяет текст экземпляра `s` строкой `'other'`.

Создание и добавление новых узлов

Очень важно иметь возможность изменения существующих узлов, однако ключевую роль при создании HTML-документов с нуля играют операции создания новых узлов и их добавления в дерево.

Чтобы создать экземпляр `NavigableString`, достаточно вызвать класс, передав ему текстовое содержимое в качестве единственного аргумента:

```
s = bs4.NavigableString(' some text ')
```

Чтобы создать новый экземпляр Tag, вызовите метод new_tag экземпляра BeautifulSoup с именем тега в качестве единственного позиционного аргумента и, возможно, с именованными аргументами, задающими атрибуты.

```
t = soup.new_tag('foo', bar='baz')
print(t)
<foo bar="baz"></foo>
```

Чтобы добавить узел в качестве дочернего элемента экземпляра Tag, используйте метод append экземпляра Tag, который добавит узел после всех дочерних элементов, если таковые существуют.

```
t.append(s)
print(t)
<foo bar="baz"> some text </foo>
```

Если вы хотите, чтобы новый узел был вставлен в позицию с определенным индексом среди дочерних элементов экземпляра *t*, вызовите метод *t*.insert(*n*, *s*), который поместит *s* в позицию с индексом *n* в *t*.contents (методы *t*.append и *t*.insert работают так, как если бы *t* был списком своих дочерних элементов).

Если у вас имеется элемент *b*, принадлежащий навигационному классу, и вы хотите добавить новый узел *x* в качестве предыдущего сестринского узла по отношению к *b*, вызовите метод *b*.insert_before(*x*). Если же требуется вставить *x* в качестве следующего сестринского узла по отношению к *b*, вызовите метод *b*.insert_after(*x*).

Если вы хотите обернуть новый родительский узел *t* вокруг *b*, вызовите метод *b*.wrap(*t*) (который при этом возвращает обертывающий тег).

```
print(t.string.wrap(soup.new_tag('moo', zip='zaap')))
<moo zip="zaap"> some text </moo>
print(t)
<foo bar="baz"><moo zip="zaap"> some text </moo></foo>
```

Замена и удаление узлов

Вызов метода t.replace_with для любого тега *t* приводит к замене *t* и всего его предыдущего содержимого переданным аргументом и возвращает *t* вместе с его исходным содержимым.

```
soup = bs4.BeautifulSoup(
       '<p>first <b>second</b> <i>third</i></p>', 'lxml')
i = soup.i.replace_with('last')
soup.b.append(i)
print(soup)
<html><body><p>first <b>second<i>third</i></b> last</p></body></html>
```

Вы можете вызвать для любого тега *t* метод *t*.unwrap(), который заменяет *t* его содержимым и возвращает *t* пустым, т.е. без содержимого.

```
empty_i = soup.i.unwrap()
print(soup.b.wrap(empty_i))
<i><b>secondthird</b></i>
print(soup)
<html><body><p>first <i><b>secondthird</b></i> last</p></body></html>
```

Вызов метода `t.clear()` удаляет содержимое `t`, уничтожает его и оставляет `t` пустым (но он по-прежнему занимает свою первоначальную позицию в дереве). Вызов `t.decompose()` удаляет и уничтожает как `t`, так и его содержимое.

```
soup.i.clear()
print(soup)
<html><body><p>first <i></i> last</p></body></html>
soup.p.decompose()
print(soup)
<html><body></body></html>
```

Наконец, вызов метода `t.extract()` удаляет `t` и его содержимое, но — не выполняя их фактического уничтожения — возвращает `t` вместе с его первоначальным содержимым.

Создание HTML-документа с помощью модуля `bs4`

Ниже приведен пример генерации HTML-документа с помощью методов модуля `bs4`, предназначенных для построения деревьев. В частности, следующая функция принимает последовательность "строк таблицы" (последовательностей) и возвращает строку в виде таблицы HTML, отображающей их значения.

```
def mktable_with_bs4(s_of_s):
  tabsoup = bs4.BeautifulSoup('<table>', 'html.parser')
  tab = tabsoup.table
  for s in s_of_s:
    tr = tabsoup.new_tag('tr')
    tab.append(tr)
    for item in s:
      td = tabsoup.new_tag('td')
      tr.append(td)
      td.string = str(item)
  return tab
```

Вот так выглядят результаты работы этой функции.

```
example = (
  ('foo', 'g>h', 'g&h'),
  ('zip', 'zap', 'zop'),
)
print(mktable_with_bs4(example))
# вывод:
<table><tr><td>foo</td><td>g&gt;h</td><td>g&amp;h</td></tr>
<tr><td>zip</td><td>zap</td><td>zop</td></tr></table>
```

Обратите внимание на то, что модуль `bs4` автоматически "экранирует" строки, содержащие символы разметки, такие как <, > и &. Например, на основе строки `'g>h'` формируется HTML-разметка в виде `'g>h'`.

Работа с шаблонами

Во многих случаях для генерации HTML-документов лучше всего воспользоваться подходом, основанным на *шаблонизации* контента. Начните с *шаблона* — текстовой строки (часто прочитанной из файла, базы данных и т.п.), которая представляет собой почти допустимый HTML-документ, но включает маркеры, так называемые *заполнители*, вместо которых должен быть вставлен динамически генерируемый текст. Ваша программа генерирует требуемый текст и подставляет его в шаблон.

В простейшем случае можно использовать маркеры вида `{`*имя*`}`. Установите динамически сгенерированный текст в качестве значения для ключа `'`*имя*`'` в словаре *d*. Всю остальную работу позволяет выполнить метод форматирования строк `.format` (см. раздел "Форматирование строк" в главе 8): если *t* — строка шаблона, то результат вызова `t.format(d)` представляет собой копию шаблона с надлежащим образом подставленными значениями.

В общем случае вы будете использовать не только простую подстановку значений, но также условную подстановку и циклы, что значительно расширит ваши возможности форматирования результирующего HTML-документа и управления его представлением. Действуя в духе отделения всего, что связано с бизнес-логикой приложения, от того, что связано с его представлением, вы сможете вынести решение задач представления в шаблоны. И здесь вам на выручку придут сторонние пакеты, специально предназначенные для этих целей. Существует множество возможных вариантов, но все авторы данной книги, которые в прошлом использовали и даже разрабатывали подобные пакеты (`https://www.safaribooksonline.com/library/view/python-cookbook/0596001673/ch03s23.html`), в настоящее время отдают предпочтение пакету `jinja2` (`http://jinja.pocoo.org/docs/dev/`), о котором пойдет речь в следующем разделе.

Пакет `jinja2`

Для серьезных задач, связанных с применением шаблонов, мы рекомендуем использовать пакет `jinja2`, доступный для загрузки на сайте PyPI (`https://pypi.python.org/pypi/Jinja2`), который, как и другие сторонние пакеты Python, можно легко установить с помощью команды **`pip install jinja2`**.

Пакет `jinja2` снабжен великолепной документацией, охватывающей:

- собственно язык создания шаблонов (`http://jinja.pocoo.org/docs/dev/templates/`), концептуально смоделированный по типу Python, но с многочисленными отличиями, обусловленными необходимостью поддержки его встраивания в HTML и удовлетворения специфических потребностей, связанных с представлением документа;

- API — ваш код на языке Python использует его для подключения к пакету `jinja2`, в случае необходимости расширяя (`http://jinja.pocoo.org/docs/dev/api/#custom-filters`) или дополняя (`http://jinja.pocoo.org/docs/dev/extensions/`) его;
- другие темы, такие как установка и интернационализация, ограниченная среда выполнения ("песочница") и портирование кода из других механизмов шаблонизации, не говоря уже о многочисленных ценных подсказках и полезных приемах.

В этом разделе мы обсудим лишь небольшое подмножество функциональности пакета `jinja2`, достаточное только для того, чтобы вы могли приступить к работе сразу же после его установки. Мы настоятельно рекомендуем вам изучить документацию `jinja` для получения огромного количества прекрасно изложенной дополнительной информации.

Класс `jinja2.Environment`

Используя `jinja2`, вы всегда будете иметь дело с экземпляром `Environment`. В некоторых случаях вы сможете обойтись используемым по умолчанию типичным "разделяемым окружением", но поступать так не рекомендуется. Лишь в весьма нестандартных ситуациях, когда вы получаете шаблоны из других источников (или с другим синтаксисом языка построения шаблонов), вам потребуется определить несколько окружений. Обычно инстанциализируют один экземпляр *`env`* класса `Environment`, пригодный для всех шаблонов, рендеринг которых вам потребуется.

В процессе создания экземпляра *`env`* его можно адаптировать многими способами, передавая конструктору именованные аргументы (в частности, позволяющие изменять свойства ключевых элементов синтаксиса языка шаблонов, таких как начальный и конечный разделители блоков, переменные, комментарии и пр.). Однако существует один именованный аргумент, который в реальных приложениях вы будете передавать почти всегда: `loader=...`.

Используемый окружением загрузчик (`loader`) определяет, откуда должны загружаться шаблоны. Обычно они будут загружаться из некоторого каталога файловой системы или из базы данных (в последнем случае вам придется написать пользовательский подкласс `jinja2.Loader`), но существуют и другие возможности.

Загрузчик нужен для того, чтобы шаблоны могли использовать некоторые мощные возможности `jinja2`, такие как *наследование шаблонов* (в данной книге эти возможности не рассматриваются).

Вы можете дополнить создаваемый экземпляр *`env`* пользовательскими фильтрами, тестами, расширениями и т.п., но все эти элементы могут быть добавлены позже, и в данной книге они не рассматриваются.

В последующих примерах предполагается, что экземпляр *`env`* инстанциализировался с единственным аргументом `loader=jinja2.FileSystemLoader('/`*`путь/к/`*

шаблонам') и его возможности дополнительно не расширялись. В действительности, для простоты, мы даже не будем использовать загрузчик. Однако в реальности загрузчик задается практически всегда, тогда как другие опции — в редких случаях.

Вызов `env.get_template(name)` извлекает, компилирует и возвращает экземпляр `jinja2.Template` на основе того, что возвращает вызов `env.loader(name)`. В следующем примере с целью упрощения мы используем редко рекомендуемый вызов `env.from_string(s)` для создания экземпляра `jinja2.Template` из строки `s` (в идеальном случае — строки Unicode, хотя для этого подойдет также байтовая строка, преобразованная с использованием кодировки `utf8`).

Класс `jinja2.Template`

Экземпляр `t` класса `jinja2.Template` имеет множество атрибутов и методов, но в реальности вы будете практически всегда использовать описанный ниже метод `render`.

`render` `t.render(...context...)`

Аргументы `context` — это те же аргументы, которые вы могли бы передать конструктору словаря (экземпляра отображения), и/или аргументы, расширяющие и потенциально переопределяющие связи между ключами и значениями отображения.

Вызов `t.render(context)` возвращает строку (Unicode), являющуюся результатом применения аргументов `context` к шаблону `t`.

Создание HTML-документа с помощью пакета `jinja2`

Ниже приведен пример использования шаблона `jinja2` для генерации HTML-документа. В частности, как и в предыдущем примере (раздел "Создание HTML-документа с помощью модуля `bs4`"), следующая функция принимает последовательность "строк таблицы" (последовательностей) и возвращает HTML-таблицу, отображающую их значения.

```
TABLE_TEMPLATE = '''\
<table>
{% for s in s_of_s %}
  <tr>
  {% for item in s %}
    <td>{{item}}</td>
  {% endfor %}
  </tr>
{% endfor %}
</table>'''

def mktable_with_jinja2(s_of_s):
    env = jinja2.Environment(
        trim_blocks=True,
        lstrip_blocks=True,
```

```
        autoescape=True)
    t = env.from_string(TABLE_TEMPLATE)
    return t.render(s_of_s=s_of_s)
```

Данная функция создает окружение с опцией `autoescape=True`, которая устанавливает автоматическое "экранирование" строк, содержащих символы разметки, такие как <, > и &. Например, если `autoescape=True`, то `'g>h'` преобразуется в `'g>h'`.

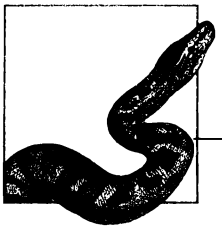
Опции `trim_blocks=True` и `lstrip_blocks=True` — исключительно "косметические" и обеспечивают всего лишь аккуратное форматирование строки шаблона и сформированной строки HTML. Разумеется, браузеру в процессе визуализации HTML-разметки совершенно безразлично, аккуратно ли отформатирована сама строка HTML.

Как правило, вы будете всегда создавать окружение с использованием опции `loader=...`, которая будет загружать шаблоны из файлов или из других хранилищ с помощью вызовов таких методов, как *t = env*`.get_template(`*template_name*`)`. В данном примере, чтобы предоставить все в одном месте, мы опускаем загрузчик и вместо этого создаем шаблон на основе строки посредством вызова метода `env.from_string`. Обратите внимание на то, что пакет `jinja2` не учитывает специфику HTML и XML, поэтому использование только его одного не гарантирует допустимость сгенерированного содержимого, которое следует тщательно проверять, если необходимо обеспечить соответствие полученного документа требованиям стандарта.

Из многих десятков средств, предлагаемых языком шаблонов `jinja2`, в этом примере применены лишь два, которые используются наиболее часто: циклы (т.е. блоки, заключенные в конструкции вида `{% for ...%}` и `{% endfor %}`) и подстановка параметров (встроенные выражения, заключенные в пары фигурных скобок `{{` и `}}`).

Ниже приведен пример, иллюстрирующий применение определенной выше функции.

```
example = (
  ('foo', 'g>h', 'g&h'),
  ('zip', 'zap', 'zop'),
)
print(mktable_with_jinja2(example))
# вывод:
<table>
  <tr>
    <td>foo</td>
    <td>g&gt;h</td>
    <td>g&amp;h</td>
  </tr>
  <tr>
    <td>zip</td>
    <td>zap</td>
    <td>zop</td>
  </tr>
</table>
```



23

Структурированный текст: XML

XML (eXtensible Markup Language — *расширяемый язык разметки*) — широко используемый формат обмена данными. Поверх самого XML сообщество XML в тесном сотрудничестве с Консорциумом Всемирной паутины (World Wide Web Consortium, W3C) стандартизировало множество других технологий, таких как языки описания структуры XML-документов, пространства имен, XPath, XLink, XPointer и XSLT.

Индустриальные консорциумы определили поверх XML специфические для соответствующих отраслей языки разметки, предназначенные для обмена данными между приложениями. XML, языки разметки на основе XML и другие связанные с XML технологии часто используются для кросс-языкового и кросс-платформенного обмена данными между приложениями в конкретных отраслях.

По историческим причинам сложилось так, что стандартная библиотека Python включает несколько модулей с перекрывающейся функциональностью, которые поддерживают XML и находятся в пакете `xml`. Поскольку не все они рассматриваются в данной книге, для более подробного ознакомления с соответствующими модулями обратитесь к онлайн-документации (`https://docs.python.org/3/library/xml.html`).

В данной книге (и в частности, в данной главе) рассматривается лишь один из подходов к обработке XML-документов, наиболее соответствующий духу Python, который основан на использовании модуля `ElementTree`, чьи элегантность, быстродействие, общность, множественность реализаций и выдержанная в духе Python архитектура делают его наиболее предпочтительным модулем для XML-приложений на языке Python. Более полные практические руководства и подробное описание модуля `xml.etree.ElementTree` вы найдете в онлайн-документации (`https://docs.python.org/3/library/xml.etree.elementtree.html`), а также на сайте создателя модуля `ElementTree` Фредрика Лунда, известного под псевдонимом "the effbot" (`http://effbot.org/zone/element-index.htm`)[1].

[1] Алекс Мартелли слишком скромен, чтобы упомянуть об этом, но в период примерно с 1995 по 2005 год он и Фредрик, а вместе с ними и Тим Петерс были Python-ботами. Известные своими энциклопедическими и доскональными знаниями языка, *effbot*, *martellibot* и *timbot* создали програмное обеспечение, представляющее огромную ценность для многих людей.

В данной книге предполагается, что читателю известны основы самого языка XML. Если вам необходимо узнать больше об XML, обратитесь к книге *XML in a Nutshell* (`http://shop.oreilly.com/product/9780596007645.do`).

Анализ документов XML, полученных из ненадежных источников, подвергает ваше приложение риску многих возможных атак. В этой книге вопросы безопасности отдельно не рассматриваются. Для получения дополнительной информации обратитесь к онлайн-документации (`https://docs.python.org/3/library/xml.html#xml-vulnerabilities`), где вы найдете ссылки на сторонние модули, позволяющие защитить приложение, если вам приходится анализировать XML-документы, которые получены из источников, не заслуживающих полного доверия. В частности, если вы хотите использовать реализацию `ElementTree`, в которой предусмотрена защита от парсинга XML-документов из ненадежных источников, обратите внимание на модуль `defusedxml.ElementTree` (`https://pypi.python.org/pypi/defusedxml#defusedxml-elementtree`) и его C-аналог, модуль `defusedxml.cElementTree` (`https://pypi.python.org/pypi/defusedxml#defusedxml-celementtree`), входящие в состав стороннего пакета `defusedxml` (`https://pypi.python.org/pypi/defusedxml`).

Модуль `ElementTree`

Python и сторонние дополнения предлагают несколько альтернативных реализаций функциональности `ElementTree`. Той реализацией, на наличие которой в стандартной библиотеке вы всегда можете рассчитывать, является модуль `xml.etree.ElementTree`. В версии v2 в большинстве ситуаций можно использовать более быструю C-реализацию `xml.etree.cElementTree`. В версии v3 обычный импорт модуля `xml.etree.ElementTree` обеспечивает доступ к самой быстрой реализации. Сторонний пакет `defusedxml`, который упоминался в предыдущем разделе, предлагает несколько более медленную, но вместе с тем более безопасную реализацию, если вам приходится анализировать XML-документы, полученные из ненадежных источников. Другой сторонний пакет, `lxml` (`http://lxml.de`), предлагает лучшее быстродействие и дополнительную функциональность посредством модуля `lxml.etree` (`http://lxml.de/api.html`).

Традиционным способом получения любой доступной реализации `ElementTree`, которую вы предпочитаете, является использование инструкции `from...import...as`:

```
from xml.etree import cElementTree as et
```

Допускается вставлять несколько таких инструкций с применением синтаксиса `try...except ImportError` (это защитная мера при поиске наилучшей из доступных реализаций) и использованием `et` (некоторые предпочитают вариант `ET` в верхнем регистре) в качестве имени модуля в оставшейся части вашего кода.

Модуль `ElementTree` содержит один фундаментальный класс, который представляет *узел* в *дереве*, естественным образом отображающем XML-документ, — класс

Element. Кроме того, модуль ElementTree содержит другие важные классы, основным из которых является класс, представляющий все дерево и располагающий методами для ввода и вывода данных, а также многочисленными вспомогательными методами, эквивалентными методам своего *корневого* узла Element, — т.е. класс ElementTree. В дополнение к ним модуль ElementTree содержит несколько служебных функций и менее важных вспомогательных классов.

Класс Element

Класс Element представляет узел в дереве, отображающем весь XML-документ, и является центральным элементом всей экосистемы ElementTree. Каждый элемент ведет себя, с одной стороны, как отображение с *атрибутами*, переводящими строковые ключи в строковые значения, а с другой — как последовательность с *дочерними элементами*, являющимися другими элементами (иногда называемыми "подэлементами" данного элемента). Кроме того, каждый элемент предлагает несколько дополнительных атрибутов и методов. Каждый экземпляр e класса Element имеет четыре атрибута данных, или свойства, описание которых приведено ниже.

attrib Словарь, содержащий все атрибуты узла XML, в котором ключами являются строки, содержащие имена атрибутов (а соответствующими значениями, как правило, также являются строки). Например, выполнив парсинг XML-фрагмента bc, вы получите экземпляр e, атрибутом e.attrib которого является {'x': 'y'}.

По возможности избегайте использования словаря attrib экземпляров Element

Обычно лучше всего не обращаться к словарю e.attrib, поскольку реализации может потребоваться создавать его на лету, когда вы обратитесь к нему. Как будет показано далее, сам экземпляр e предлагает некоторые типичные методы отображений, которые вы можете вызывать для e.attrib. Использование собственных методов e позволяет применять интеллектуальную оптимизацию кода, обеспечивающую повышение производительности кода по сравнению с той, которую вы получили бы, используя фактический словарь e.attrib.

tag XML-тег узла (строка), также известный как "*тип* элемента". Например, выполнив парсинг XML-фрагмента bc, вы получите экземпляр e со свойством e.tag, установленным в 'a'.

tail Произвольные данные (строка), следующие непосредственно за элементом. Например, выполнив парсинг XML-фрагмента bc, вы получите экземпляр e со свойством e.tail, установленным в 'c'.

text Произвольные данные (строка), находящиеся непосредственно "внутри" элемента. Например, выполнив парсинг XML-фрагмента bc, вы получите экземпляр e со свойством e.text, установленным в 'b'.

Экземпляр `e` имеет методы, действующие подобно отображению и позволяющие избежать явного запроса словаря `e.attrib`.

clear `e.clear()`

Вызов `e.clear()` оставляет `e` "пустым", за исключением его тега, удаляя все атрибуты и дочерние элементы и устанавливая для свойств `text` и `tail` значение `None`.

get `e.get(key, default=None)`

Подобен методу `e.attrib.get(key, default)`, но потенциально работает намного быстрее. Вы не сможете использовать обращение `e[key]`, поскольку индексирование `e` используется для доступа к дочерним элементам, а не атрибутам.

items `e.items()`

Возвращает список кортежей `(name, value)` для всех атрибутов в произвольном порядке.

keys `e.keys()`

Возвращает список имен всех атрибутов в произвольном порядке.

set `e.set(key, value)`

Устанавливает для атрибута `key` значение `value`.

Другие методы экземпляра `e` (включая индексирование с помощью синтаксиса `e[i]` и определение длины `e`, как в вызове `len(e)`) имеют дело с дочерними элементами `e` как с последовательностью или — в некоторых случаях, указанных в оставшейся части данного раздела, — со всеми потомками (элементы в поддереве с корнем в `e`, также называемые подэлементами `e`).

Не полагайтесь на неявное булево преобразование экземпляра `Element`

Во всех версиях вплоть до Python 3.6 тестирование экземпляра `e` класса `Element` дает значение `False`, если `e` не имеет дочерних элементов, что соответствует обычному правилу неявного булевого преобразования для контейнеров Python. Однако в документации сказано, что в одной из будущих версий v3 это поведение может быть изменено. В целях совместимости с будущими версиями, если вы хотите проверить, не содержит ли `e` дочерние элементы, выполняйте явную проверку `if len(e) == 0:` — не используйте обычную идиому Python `if not e:`.

Упомянутые методы экземпляра `e`, имеющие дело с дочерними элементами или потомками, описаны ниже. Спецификация XPath в данной книге не рассматривается. Для получения более подробной информации по этой теме обратитесь к онлайн-документации (`https://docs.python.org/3/library/xml.etree.elementtree.html#elementtree-xpath`).

append `e.append(se)`

Добавляет подэлемент *se* (который должен быть экземпляром `Element`) в конец последовательности дочерних элементов *e*.

extend `e.extend(ses)`

Добавляет каждый элемент итерируемого объекта *ses* (каждый элемент должен быть экземпляром `Element`) в конец последовательности дочерних элементов *e*.

find `e.find(match, namespaces=None)`

Возвращает первого потомка, соответствующего аргументу *match*, который может быть именем тега или выражением XPath в пределах подмножества, поддерживаемого текущей реализацией `ElementTree`. В отсутствие потомков, соответствующих аргументу *match*, возвращает значение `None`. Только в версии v3 аргумент *namespaces* — необязательное отображение с префиксами пространства имен XML в качестве ключей и соответствующими полными именами пространств имен XML в качестве значений.

findall `e.findall(match, namespaces=None)`

Возвращает список всех потомков, соответствующих аргументу *match*, который может быть именем тега или выражением XPath в пределах подмножества, поддерживаемого текущей реализацией `ElementTree`. В отсутствие потомков, соответствующих аргументу *match*, возвращает `[]`. Только в версии v3 аргумент *namespaces* — необязательное отображение с префиксами пространства имен XML в качестве ключей и соответствующими полными именами пространств имен XML в качестве значений.

findtext `e.findtext(match, default=None, namespaces=None)`

Возвращает текст первого потомка, соответствующего аргументу *match*, который может быть именем тега или выражением XPath в пределах подмножества, поддерживаемого текущей реализацией `ElementTree`. Результат может быть пустой строкой `''`, если первый потомок, соответствующий аргументу *match*, не имеет текста. В отсутствие потомков, соответствующих аргументу *match*, возвращает значение *default*. Только в версии v3 аргумент *namespaces* — необязательное отображение с префиксами пространства имен XML в качестве ключей и соответствующими полными именами пространств имен XML в качестве значений.

insert `e.insert(index, se)`

Добавляет подэлемент *se* (который должен быть экземпляром `Element`) с индексом *index* в пределах последовательности дочерних элементов *e*.

iter `e.iter(tag='*')`

Возвращает итератор, совершающий обход всех потомков *e* в порядке "сначала в глубину". Если аргумент *tag* не равен `'*'`, то возвращает только подэлементы, атрибут `tag` которых равен аргументу *tag*. Не изменяйте поддерево, корнем которого является *e*, в процессе циклического перебора с использованием `e.iter`.

iterfind `e.iterfind(match, namespaces=None)`

Возвращает итератор, выполняющий обход (в порядке "сначала в глубину") всех потомков, соответствующих аргументу *match*, который может быть именем тега или выражением XPath в пределах подмножества, поддерживаемого текущей реализацией `ElementTree`. В отсутствие потомков, соответствующих аргументу *match*, результирующий итератор является пустым. Только в версии v3 аргумент *namespaces* — необязательное отображение с префиксами пространства имен XML в качестве ключей и соответствующими полными именами пространств имен XML в качестве значений.

itertext `e.itertext(match, namespaces=None)`

Возвращает итератор, выполняющий обход (в порядке "сначала в глубину") атрибутов `text` (но не `tail`) всех потомков, соответствующих аргументу *match*, который может быть именем тега или выражением XPath в пределах подмножества, поддерживаемого текущей реализацией `ElementTree`. В отсутствие потомков, соответствующих аргументу *match*, результирующий итератор является пустым. Только в версии v3 аргумент *namespaces* — необязательное отображение с префиксами пространства имен XML в качестве ключей и соответствующими полными именами пространств имен XML в качестве значений.

remove `e.remove(se)`

Удаляет потомка, который является элементом *se* (см. проверки тождественности в табл. 3.2).

Класс `ElementTree`

Класс `ElementTree` представляет дерево, отображающее XML-документ. Основной ценностью экземпляра *et* класса `ElementTree` являются описанные ниже методы для группового парсинга (ввода) и записи (вывода) дерева в целом.

Таблица 23.1. Методы экземпляра `ElementTree`, предназначенные для парсинга и записи

Метод	Описание
parse	*et*`.parse(`*source*`, `*parser*`=None)` Аргумент *source* может быть файлом, открытым для чтения, или именем файла для открытия и чтения (для парсинга строки оберните ее экземпляром класса `io.StringIO`, описанного в разделе "Файлы в памяти: функции `io.StringIO` и `io.BytesIO`" главы 10), который содержит текст XML. Метод *et*`.parse` анализирует этот текст, создает дерево экземпляров `Element` в качестве нового содержимого *et* (отбрасывая предыдущее содержимое *et*, если оно было) и возвращает корневой элемент дерева. Аргумент *parser* — необязательный экземпляр парсера. По умолчанию метод *et*`.parse` использует экземпляр класса `XMLParser`, предоставляемый модулем `ElementTree`. В этой книге класс `XMLParser` не рассматривается. Для получения более подробной информации обратитесь к онлайн-документации (`https://docs.python.org/3/library/xml.etree.elementtree.html#xmlparser-objects`)

Метод	Описание
write	*et*.write(*file*, *encoding*='us-ascii', *xml_declaration*=None, *default_namespace*=None, *method*='xml', *short_empty_elements*=True) Аргумент *file* может быть файлом, открытым для записи, или именем файла для открытия и записи (для записи в строку передайте в качестве аргумента *file* экземпляр класса io.StringIO, рассмотренного в разделе "Файлы в памяти: функции io.StringIO и io.BytesIO" главы 10). Метод *et*.write записывает в этот файл текст, который представляет XML-документ для дерева, являющегося содержимым *et*. Аргумент *encoding* должен записываться в соответствии со стандартом, например 'iso-8859-1', а не 'latin-1', даже если сам Python допускает обе формы записи этой кодировки. В версии v3 вы можете передать в качестве *encoding* значение 'unicode' для вывода текстовых (Unicode) строк, если метод *file*.write принимает такие строки. В противном случае метод *file*.write должен принимать байтовые строки, и именно строки этого типа выводит метод *et*.write, используя символьные ссылки XML для символов, отсутствующих в кодировке. Например, в случае используемой по умолчанию кодировки ASCII символ e с акцентом, é, выводится как é. Допускается передать для аргумента *xml_declaration* значение False, чтобы не выводить декларацию в результирующем тексте, или значение True, чтобы она выводилась. По умолчанию декларация выводится только в том случае, если кодировка не является одной из следующих: 'us-ascii', 'utf-8' или (только в версии v3) 'unicode'. Необязательный аргумент *default_namespace* задает пространство имен по умолчанию для конструкций xmlns. Значение 'text' аргумента *method* означает вывод только атрибутов text и tail каждого узла (но не tag). Значение 'html' аргумента *method* означает вывод документа в формате HTML (который, например, опускает закрывающие теги, необязательные в HTML, такие как </br>). Значение по умолчанию — 'xml', которому соответствует вывод в формате XML. В версии v3 вы можете передать для аргумента *short_empty_elements* (задаваемого только как именованный, а не позиционный) значение False, чтобы всегда использовались начальный (открывающий) и конечный (закрывающий) теги, причем даже для элементов, не имеющих ни текста, ни подэлементов. По умолчанию для таких пустых элементов используется сокращенная форма XML. Например, пустой элемент с тегом a по умолчанию выводится как <a/>, но в версии v3 — как <a>, если вы передаете для аргумента short_empty_elements значение False

Кроме того, экземпляр *et* класса ElementTree предоставляет метод getroot — вызов *et*.getroot() возвращает корень дерева — и вспомогательные методы find, findall, findtext, iter и iterfind, каждый из которых в точности эквивалентен вызову тех же методов для корня дерева, т.е. для результата вызова метода *et*.getroot().

Функции, предоставляемые модулем ElementTree

Модуль ElementTree содержит также несколько функций, описанных в табл. 23.2.

Таблица 23.2. Функции модуля ElementTree

Функция	Описание
Comment	Comment(*text*=None) Возвращает экземпляр Element, который после его вставки в качестве узла в дерево ElementTree будет выводиться как комментарий с заданной строкой текста, заключенной между символами '<!--' и '-->'. XMLParser пропускает XML-комментарии в анализируемом им документе, поэтому данная функция является единственным способом получения узлов комментариев
ProcessingInstruction	ProcessingInstruction(*target*, *text*=None) Возвращает экземпляр Element, который после его вставки в качестве узла в дерево ElementTree будет выводиться как инструкция обработки XML с заданными строками *target* и *text*, заключенными между символами '<?' и '?>'. XMLParser пропускает инструкции обработки XML в анализируемом им документе, поэтому данная функция является единственным способом получения узлов инструкций обработки
SubElement	SubElement(*parent*, *tag*, *attrib*={}, ***extra*) Создает экземпляр Element с заданным тегом *tag*, а также атрибутами из словаря *attrib* и другими атрибутами, переданными в качестве именованных аргументов посредством аргумента *extra*, и присоединяет его в качестве крайнего справа дочернего элемента родительского экземпляра Element. Возвращает созданный экземпляр Element
XML	XML(*text*, *parser*=None) Анализирует XML-разметку из строки *text* и возвращает экземпляр Element. Необязательный аргумент *parser* задает экземпляр парсера. По умолчанию функция XML использует экземпляр класса XMLParser, предоставляемый модулем ElementTree. В этой книге класс XMLParser не рассматривается. Для получения более подробной информации обратитесь к онлайн-документации (https://docs.python.org/3/library/xml.etree.elementtree.html#xmlparser-objects)
XMLID	XMLID(*text*, *parser*=None) Анализирует XML-разметку из строки *text* и возвращает кортеж из двух элементов: экземпляра Element и словаря, отображающего атрибуты *id* на единственный экземпляр Element, имеющий каждый из этих атрибутов (XML запрещает

Функция	Описание
	дублирование атрибутов *id*). Необязательный аргумент *parser* задает экземпляр парсера. По умолчанию функция XMLID использует экземпляр класса XMLParser, предоставляемый модулем ElementTree. В этой книге класс XMLParser не рассматривается. Для получения более подробной информации обратитесь к онлайн-документации (https://docs.python.org/3/library/xml.etree.elementtree.html#xmlparser-objects)
dump	dump(*e*) Записывает аргумент *e*, который может быть экземпляром класса Element или класса ElementTree, в качестве XML-разметки в стандартный вывод sys.stdout. Данная функция предназначена для использования в целях отладки
fromstring	fromstring(*text*, *parser*=None) Анализирует XML-разметку из строки *text* и возвращает экземпляр Element, как и рассмотренная выше функция XML
fromstringlist	fromstringlist(*sequence*, *parser*=None) Аналогична вызову fromstring(''.join(*sequence*)), но может выполняться немного быстрее за счет исключения явного вызова функции join
iselement	iselement(*e*) Возвращает значение True, если *e* — экземпляр класса Element
iterparse	iterparse(*source*, *events*=['end'], *parser*=None) Аргумент *source* может быть файлом, открытым для чтения, или именем файла для открытия и чтения, который содержит документ XML в виде текста. Функция iterparse возвращает итератор, который в процессе парсинга документа и инкрементного создания функцией iterparse соответствующего экземпляра ElementTree возвращает кортежи (*event*, *element*), где элемент кортежа *event* — одна из строк, указанных в аргументе *events* (который должен быть строкой 'start', 'end', 'start-ns' или 'end-ns'). Элемент кортежа *element* — экземпляр Element для событий 'start' и 'end', значение None для события 'end-ns' и кортеж из двух строк (*namespace_prefix*, *namespace_uri*) для события 'start-ns'. Необязательный аргумент *parser* задает экземпляр парсера. По умолчанию функция iterparse использует экземпляр класса XMLParser, предоставляемый модулем ElementTree. В этой книге

Функция	Описание
	класс `XMLParser` не рассматривается. Для получения более подробной информации обратитесь к онлайн-документации (`https://docs.python.org/3/library/xml.etree.elementtree.html#xmlparser-objects`).
	Функция `iterparse` обеспечивает возможность итеративного парсинга большого XML-документа, по возможности избегая сохранения результирующего дерева `ElementTree` целиком в памяти. Более подробно функция `iterparse` рассмотрена в разделе "Итеративный парсинг XML"
`parse`	`parse(`*`source`*`, `*`parser`*`=None)`
	Аналогична методу `parse` экземпляра `ElementTree`, описанного в табл. 23.1, за исключением того, что данная функция возвращает экземпляр `ElementTree`, который она создает
`register_namespace`	`register_namespace(`*`prefix`*`, `*`uri`*`)`
	Регистрирует строку *`prefix`* в качестве пространства имен для строки *`uri`*. Элементы в пространстве имен сериализуются с использованием этого префикса
`tostring`	`tostring(e, `*`encoding`*`='us-ascii, `*`method`*`='xml', `*`short_empty_elements`*`=True)`
	Возвращает строку с XML-представлением поддерева, корнем которого является экземпляр *`e`* класса `Element`. Аргументы имеют тот же смысл, что и в методе `write` экземпляра `ElementTree`, описанного в табл. 23.1
`tostringlist`	`tostringlist(e, `*`encoding`*`='us-ascii, `*`method`*`='xml', `*`short_empty_elements`*`=True)`
	Возвращает список строк с XML-представлением поддерева, корнем которого является элемент *`e`* класса `Element`. Аргументы имеют тот же смысл, что и в методе `write` экземпляра `ElementTree`, описанного в табл. 23.1

Кроме того, модуль `ElementTree` предоставляет классы `QName`, `TreeBuilder` и `XMLParser`, которые в данной книге не рассматриваются. В версии v3 данный модуль дополнительно предоставляет класс `XMLPullParser`, рассмотренный в разделе "Итеративный парсинг XML".

Анализ XML-разметки с помощью парсера `ElementTree.parse`

В повседневной работе экземпляр `ElementTree` чаще всего создают посредством парсинга содержимого файла или файлового объекта, обычно с помощью функции `parse` или метода `parse` экземпляров класса `ElementTree`.

Для последующих примеров в этой главе мы используем простой XML-файл, который доступен по адресу `http://www.w3schools.com/xml/simple.xml`. Его корневым элементом является `'breakfast_menu'`, а дочерними элементами корневого элемента являются элементы с тегом `'food'`. Каждый элемент `'food'` имеет дочерний элемент с тегом `'name'`, текстом которого является название блюда, и дочерний элемент с тегом `'calories'`, текстом которого является строковое представление целочисленного значения калорийности порции данного блюда. Упрощенное представление интересующего нас содержимого XML-файла в этом примере выглядит так.

```
<breakfast_menu>
  <food>
    <name>Belgian Waffles</name>
    <calories>650</calories>
  </food>
  <food>
    <name>Strawberry Belgian Waffles</name>
    <calories>900</calories>
  </food>
  <food>
    <name>Berry-Berry Belgian Waffles</name>
    <calories>900</calories>
  </food>
  <food>
    <name>French Toast</name>
    <calories>600</calories>
  </food>
  <food>
    <name>Homestyle Breakfast</name>
    <calories>950</calories>
  </food>
</breakfast_menu>
```

Поскольку XML-документ находится по заданному URL-адресу, мы начинаем с получения файлового объекта с интересующим нас содержимым и передаем его для анализа. В версии v2 проще всего это делается следующим образом.

```
import urllib
from xml.etree import ElementTree as et

content = urllib.urlopen('http://www.w3schools.com/xml/simple.xml')
tree = et.parse(content)
```

В версии v3 для этого проще всего воспользоваться модулем `request`.

```
from urllib import request
from xml.etree import ElementTree as et
```

```
content = request.urlopen('http://www.w3schools.com/xml/simple.xml')
tree = et.parse(content)
```

Выбор элементов из дерева элементов

Предположим, мы хотим записать в стандартное устройство вывода калорийность и названия различных блюд в порядке возрастания их калорийности с дополнительной сортировкой в алфавитном порядке. Код для этой задачи выглядит одинаково в версиях v2 и v3.

```
def bycal_and_name(e):
  return int(e.find('calories').text), e.find('name').text

for e in sorted(tree.findall('food'), key=bycal_and_name):
  print('{} {}'.format(e.find('calories').text,
                       e.find('name').text))
```

Выполнение этой функции дает следующий результат.

```
600 French Toast
650 Belgian Waffles
900 Berry-Berry Belgian Waffles
900 Strawberry Belgian Waffles
950 Homestyle Breakfast
```

Изменение дерева элементов

Как только построено дерево элементов `ElementTree` (посредством парсинга или иным способом), его можно изменить — вставляя, удаляя и/или изменяя узлы (элементы), — с помощью различных методов классов `ElementTree` и `Element` и функций модуля. Предположим, нашей программе стало известно, что в меню было добавлено новое блюдо: *buttered toast* (гренки) — два поджаренных ломтика белого хлеба с маслом (180 калорий), а блюда, названия которых содержат слово "berry" (ягода) в любом регистре, удалены из меню. Код для изменения части дерева, связанной с упомянутыми блюдами, может выглядеть примерно так.

```
# Добавить Buttered Toast в меню
menu = tree.getroot()
toast = et.SubElement(menu, 'food')
tcals = et.SubElement(toast, 'calories')
tcals.text = '180'
tname = et.SubElement(toast, 'name')
tname.text = 'Buttered Toast'

# Удалить из меню все, что связано со словом 'berry'
for e in menu.findall('food'):
    name = e.find('name').text
    if 'berry' in name.lower():
        menu.remove(e)
```

Как только мы вставим этот "редактирующий" код между кодом парсинга и кодом селективного вывода, результат примет следующий вид.

```
180 Buttered Toast
600 French Toast
650 Belgian Waffles
950 Homestyle Breakfast
```

В некоторых случаях простота изменения дерева элементов может быть решающим фактором, заслуживающим того, чтобы все дерево хранилось в памяти.

Построение дерева элементов с нуля

Иногда ваша задача заключается не в том, чтобы начать с существующего XML-документа, а в том, чтобы создать требуемый XML-документ на основе данных, которые ваш код получает из различных источников, таких как CSV-документ или база данных.

Код для таких задач аналогичен только что представленному коду, предназначенному для изменения существующего дерева элементов. Остается лишь добавить небольшой фрагмент для построения первоначально пустого дерева `ElementTree`.

Предположим, у вас имеется CSV-файл *menu.csv*, двумя столбцами которого, разделенными запятыми, являются калорийность и название различных блюд, причем каждому блюду отводится строка. Ваша задача — создать XML-файл *menu.xml*, аналогичный тому, который анализировался в предыдущем примере. Ниже представлен один из возможных способов решения этой задачи.

```
import csv
from xml.etree import ElementTree as et

menu = et.Element('menu')
tree = et.ElementTree(menu)

with open('menu.csv') as f:
    r = csv.reader(f)
    for calories, namestr in r:
        food = et.SubElement(menu, 'food')
        cals = et.SubElement(food, 'calories')
        cals.text = calories
        name = et.SubElement(food, 'name')
        name.text = namestr

tree.write('menu.xml')
```

Итеративный парсинг XML

Для задач, сфокусированных на выборе элементов из существующего XML-документа, в некоторых случаях не требуется создавать все дерево `ElementTree`

в памяти — фактор, приобретающий особенно важное значение, если XML-документ очень большой. (Тот крохотный документ, с которым мы работаем в примере, не относится к этой категории, однако напрягите свое воображение и представьте аналогичный документ, также ориентированный на меню, однако предназначенный для учета блюд, количество которых исчисляется миллионами.)

Напомним, что мы по-прежнему хотим записывать в стандартный вывод калорийность и названия блюд, но на этот раз — только 10 блюд с наименьшей калорийностью в порядке увеличения калорийности и с дополнительной сортировкой в алфавитном порядке. В данном случае файл *menu.xml*, который для простоты предполагается локальным, насчитывает миллионы блюд, поэтому мы не будем хранить его целиком в памяти, поскольку совершенно очевидно, что нам не потребуется полный доступ сразу ко всем блюдам.

Ниже приведен код, который, казалось бы, решает поставленную задачу.

```
import heapq
from xml.etree import ElementTree as et

# Инициализировать кучу фиктивными записями
heap = [(999999, None)] * 10

for _, elem in et.iterparse('menu.xml'):
    if elem.tag != 'food': continue
    # Отобрано блюдо, получить его калорийность и название
    cals = int(elem.find('calories').text)
    name = elem.find('name').text
    heapq.heappush(heap, (cals, name))

for cals, name in heap:
    print(cals, name)
```

Простой подход, но расходующий память

Продемонстрированный подход работает, однако объем потребляемой памяти остается почти таким же, как и в случае подхода, основанного на полном парсинге файла с помощью метода `et.parsewould`!

Почему этот простой подход по-прежнему потребляет так много памяти? Причина в том, что метод `iterparse` создает в памяти все дерево `ElementTree`, инкрементно наращивая его, хотя при этом и использует для обратной связи всего лишь такие события, как (по умолчанию) `'end'`, которое означает следующее: "Только что завершен анализ данного элемента".

Чтобы обеспечить заметную экономию памяти, мы можем по крайней мере отбрасывать содержимое каждого элемента по завершении его обработки, а для этого необходимо сразу же после вызова `heapq.heappush` добавить вызов `addelem.clear()`, чтобы сделать только что обработанный элемент пустым.

Такой подход действительно позволяет сэкономить немного памяти, но не в полной мере, поскольку в конечном счете корень дерева окажется связанным с огромным списком пустых дочерних узлов. Чтобы добиться ощутимой экономии памяти, нам нужно получать также события `'start'`, что позволит отслеживать корень создаваемого дерева `ElementTree`, т.е. перемещать в него начало цикла.

```
root = None
for event, elem in et.iterparse('menu.xml'):
    if event == 'start':
        if root is not None: root = elem
        continue
    if elem.tag != 'food': continue  # и т.д, как раньше
```

Затем сразу же за вызовом `heapq.heappush` нужно добавить вызов `root.remove(elem)`. Такой подход позволяет сэкономить максимально возможный объем памяти и по-прежнему обеспечивает успешное решение поставленной задачи.

Парсинг XML в асинхронном цикле

Несмотря на то что метод `iterparse` в случае его корректного применения обеспечивает экономию памяти, он все еще недостаточно хорошо подходит для использования в асинхронном цикле, рассмотренном в главе 18. Это объясняется тем, что метод `iterparse` блокирует вызовы метода `read` для файлового объекта, передаваемого в качестве первого аргумента. При асинхронной обработке такое блокирование вызовов недопустимо.

В версии v2 экземпляр `ElementTree` не предлагает никакого решения этой проблемы. В версии v3 такое решение существует — в частности, в виде класса `XMLPullParser`. (В версии v2 вы можете получить эту функциональность с помощью стороннего пакета `lxml`, используя модуль `lxml.etree`.)

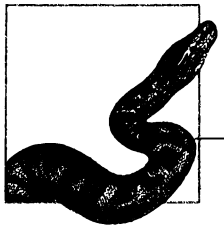
При использовании асинхронного подхода (см. главу 18) типичная задача заключается в написании "фильтрующего" компонента, которому подаются порции байтов по мере их поступления от некоторого потокового источника и который после их полного анализа возвращает события. Ниже приведен пример кода, реализующего подобный "фильтрующий" компонент с помощью класса `XMLPullParser`.

```
from xml.etree import ElementTree as et

def filter(events=None):
    pullparser = et.XMLPullParser(events)
    data = yield
    while data:
        pullparser.feed(data)
        for tup in pullparser.read_events():
            data = yield tup
    pullparser.close()
    for tup in pullparser.read_events():
        data = yield tup
```

Здесь предполагается, что фильтр используется посредством вызовов `.send(chunk)` для его результата (передача новых порций байтов осуществляется по мере их поступления) и возвращает кортежи `(event, element)`, которые обрабатываются вызывающим кодом в цикле. Поэтому фильтр фактически преобразует асинхронный поток байтов в асинхронный поток пар `(event, element)`, потребляемых итерационным способом, — типичный шаблон проектирования в современном программировании на языке Python.

V

Расширение, распространение, миграция v2/v3



24

Модули расширения и внедрение классического Python в другие программы

CPython выполняется на переносимой виртуальной машине, написанной на языке C. Встроенные объекты Python — числа, последовательности, словари, множества и файлы — также написаны на языке C, как и некоторые модули стандартной библиотеки Python. Современные платформы поддерживают динамически загружаемые библиотеки, хранящиеся в файлах с расширениями *.dll* (Windows), *.so* (Linux) и *.dylib* (Mac): эти двоичные файлы создаются в процессе сборки интерпретатора Python. Вы можете написать собственные расширения для Python на языке C, используя прикладной программный интерфейс Python/C, рассмотренный в этой главе, для создания и развертывания динамических библиотек, возможность использования которых в сценариях на языке Python и в интерактивных сеансах обеспечивает инструкция `import` (см. раздел "Инструкция `import`" в главе 6).

Под *расширением* Python подразумевается создание модулей, которые код на языке Python может импортировать для получения доступа к предоставляемой ими функциональности. Под *внедрением* (встраиванием) Python подразумевается выполнение кода на языке Python из приложений, написанных на других языках программирования. Чтобы такой подход приносил максимальную пользу, встроенный код на языке Python в свою очередь должен иметь возможность доступа к какой-то части функциональности вашего приложения. Поэтому на практике встраивание всегда предполагает определенное расширение, а также некоторые операции, специфические для встраивания. Тремя основными причинами, которые могут побудить вас к расширению Python, являются следующие:

- повторная реализация некоторой функциональности (первоначально реализованной на языке Python) на языке более низкого уровня в надежде получить более высокую производительность;
- необходимость обеспечить коду на языке Python доступ к существующей функциональности, предоставляемой библиотеками, написанными на языках более низкого уровня (или, во всяком случае, вызываемой из них);
- необходимость предоставить коду на языке Python доступ к некоторой существующей функциональности приложения в процессе встраивания Python с качестве языка написания сценариев для данного приложения.

Механизмы встраивания и расширения Python описаны в его стандартной документации. В ней вы найдете подробное практическое руководство (`https://docs.python.org/3/extending/index.html`) и обширный справочный материал (`https://docs.python.org/3/c-api/index.html`). Для понимания многих деталей лучше всего самостоятельно исследовать отлично документированный исходный код соответствующих функций на языке C. Загрузите исходный код дистрибутивного пакета Python и изучите исходный код ядра Python и модулей C-расширений, а также примеры расширений, предоставленные для учебных целей.

В этой главе обсуждаются основы расширения и встраивания Python с использованием языка C, а также упоминаются, без их углубленного обсуждения, другие способы расширения Python. Обратите внимание на ряд отличных сторонних модулей, таких, как Cython, рассмотренный в разделе "Cython", CFFI, о котором говорится в разделе "Расширение Python без использования C API", а также Numba (`http://numba.pydata.org/`) и SWIG (`www.swig.org`), не рассматриваемые в данной книге. Все они, как сказано в онлайн-документации, "предлагают как более простые, так и более совершенные подходы к созданию расширений на языках C и C++ для Python".

Для расширения или встраивания Python с использованием C API (или большинства вышеупомянутых сторонних модулей) вам потребуется знание языка C. В этой книге язык C не обсуждается. В Интернете вы найдете массу ресурсов, посвященных изучению этого языка (только не путайте C и C++: несмотря на схожесть это все-таки разные языки). В главе предполагается, что язык C известен читателю.

Расширение Python с помощью C API

Модуль расширения Python с именем *x* хранится в файле динамической библиотеки с тем же именем (*x.pyd* в Windows; *x.so* на большинстве Unix-подобных платформ) в подходящем каталоге (часто в подкаталоге *site-packages* каталога стандартной библиотеки Python). Обычно модуль расширения *x* создается на основе файла *x.c* (или, что более распространено, файла *xmodule.c*), содержащего исходный код на языке C, общая структура которого выглядит примерно так.

```
#include <Python.h>

/* опущено: тело модуля x */

PyMODINIT_FUNC
PyInit_x(void)
{
    /* опущено: код инициализации модуля x */
}
```

В версии v2 функция инициализации модуля называется `initx`. При написании расширений, предназначенных для компиляции в обеих версиях, v2 и v3, используйте конструкции препроцессора C наподобие `#if PY_MAJOR_VERSION >= 3`. Например, функция инициализации модуля для такого переносимого расширения может начинаться следующим кодом.

```
PyMODINIT_FUNC
#if PY_MAJOR_VERSION >= 3
PyInit_x(void)
#else
initx(void)
#endif
{
```

Если вы создали и установили модуль расширения, то инструкция Python `import x` загрузит динамическую библиотеку, а затем определит местонахождение и вызовет функцию инициализации модуля, которая выполнит все необходимые действия по инициализации объекта модуля *x*.

Создание и установка C-расширений Python

Самым простым и продуктивным способом создания и установки модуля C-расширения Python является использование инструмента распространения `distutils` (`https://docs.python.org/2.7/distutils/setupscript.html#describing-extension-modules`) или его улучшенной сторонней версии — `setuptools` (`https://pypi.org/project/setuptools/`). Поместите в тот каталог, в котором находится файл *x.c*, файл *setup.py*, содержащий следующие инструкции:

```
from setuptools import setup, Extension
setup(name='x', ext_modules=[Extension('x',sources=['x.c'])])
```

Далее, в предположении, что данный каталог установлен в качестве текущего, можете выполнить в командной строке следующую команду, чтобы создать и установить модуль, который после этого можно использовать в вашей установке Python:

```
C:\> python setup.py install
```

(Обычно, чтобы избежать влияния глобального состояния вашей установки Python, это лучше делать в виртуальном окружении, используя модуль `venv`, описанный в разделе "Окружения Python" главы 6, однако для простоты мы опускаем этот шаг в данной главе.)

Модуль `setuptools` осуществляет компиляцию и связывание кода, используя соответствующие команды и флаги компилятора и компоновщика, и копирует результирующую динамическую библиотеку в подходящий каталог, точное расположение которого зависит от вашей установки Python. (В зависимости от деталей этой установки вам могут потребоваться привилегии администратора или суперпользователя. Например, на компьютерах Mac или Linux можно выполнить команду **`sudo python setup.pyinstall`**, хотя, вероятно, больше всего вас устроит использование модуля `venv`). После этого ваш код на языке Python (выполняющийся в случае необходимости в виртуальном окружении) может получить доступ к результирующему модулю с помощью инструкции `import x`.

Какой компилятор C вам нужен

Обычно для компиляции C-расширений Python вам понадобится тот же C-компилятор, который использовался для создания расширяемой версии Python. Для большинства платформ Linux это означает бесплатный компилятор *gcc*, который обычно поставляется вместе с платформой или может быть бесплатно загружен для нее. Можете также рассмотреть вариант использования компилятора `clang` (`clang.llvm.org`), который, согласно всеобщему мнению, предлагает улучшенные сообщения об ошибках. На компьютерах Macintosh компилятор *gcc* (фактически играющий роль препроцессора для `clang`) поставляется вместе с бесплатной интегрированной средой разработки XCode компании Apple, которую можно загрузить и установить отдельно от приложения App Store этой же компании.

Для Windows идеально подойдет продукт компании Microsoft, называемый Microsoft Visual Studio: VS2015 для версии v3 и VS2010 для версии v2. Однако могут подойти и другие версии.

Совместимость C-расширений с различными версиями Python

Вообще говоря, успешная компиляция и выполнение C-расширения в одной версии Python еще не является гарантией его выполнения в другой версии. Например, можно с уверенностью сказать, что версия расширения, скомпилированная для Python 3.4, будет выполняться в версии Python 3.4, но дать гарантии того, что она будет выполняться в версиях 3.3 или 3.5, нельзя. В Windows не стоит даже пытаться выполнить расширение, используя другую версию Python. В других системах, таких как Linux или macOS, конкретное расширение, возможно, будет работать с несколькими версиями Python, но вы можете получить предупреждение во время импортирования модуля, и самое разумное, что можно сделать в подобной ситуации, — не игнорировать предупреждение: скомпилируйте расширение заново

соответствующим образом. (Начиная с версии Python 3.5 предоставляется возможность компилировать расширения, сохраняя совместимость с более поздними версиями.)

С другой стороны, совместимость на уровне исходного С-кода в пределах старшего номера версии почти всегда сохраняется (однако это не относится к совместимости расширений с версиями v2 и v3).

Обзор модулей С-расширений Python

Как правило, ваша С-функция `PyInit_x` будет иметь следующую общую структуру.

```
PyMODINIT_FUNC
PyInit_x(void)
{
    PyObject* m = PyModule_Create(&x_module);
    // x_module -- это экземпляр структуры PyModuleDef, описывающей
    // сам модуль и подключающей его методы (функции)

    // Далее: вызовите PyModule_AddObject(m, "somename", someobj),
    // чтобы добавить исключения, другие классы или константы модуля.
    // Наконец, когда все сделано:
    return m;
}
```

С этого момента речь будет идти главным образом о версии v3. Информацию о незначительных отличиях для версии v2 вы найдете в онлайн-руководстве (`https://docs.python.org/2.7/extending/extending.html`) и справочнике (`https://docs.python.org/2.7/c-api/index.html`).

Более подробные сведения приведены в разделе "Модуль инициализации". Структура `x_module` выглядит примерно так.

```
static struct PyModuleDef x_module = {
   PyModuleDef_HEAD_INIT,
   "x",       /* имя модуля */
   x_doc,     /* строка документирования модуля, может быть NULL */
   -1,        /* размер состояния модуля для каждого интерпретатора
                 либо -1, если модуль хранит состояние во внешних
                 переменных. */
   x_methods /* массив определений методов модуля */
};
```

Здесь `x_methods` — массив структур `PyMethodDef`. Каждая структура `PyMethodDef` в `x_methods` описывает С-функцию, которую ваш модуль *x* делает доступной для кода на языке Python, импортирующего *x*. Каждая такая функция на языке С имеет следующую общую структуру.

```
static PyObject*
func_with_named_args(PyObject* self, PyObject* args, PyObject* kwds)
{
    /* Опущено: тело функции, которая получает доступ к аргументам
       через функцию PyArg_ParseTupleAndKeywords из C API, возвращающую
       значение PyObject* либо NULL в случае ошибки */
}
```

Или слегка упрощенный вариант.

```
static PyObject*
func_with_positional_args_only(PyObject* self, PyObject* args)
{
    /* Опущено: тело функции, которая получает доступ к аргументам
       через функцию PyArg_ParseTuple из C API, возвращающую
       значение PyObject* либо NULL в случае ошибки */
}
```

О том, как функции на языке C получают доступ к аргументам, передаваемым им кодом на языке Python, рассказано в разделе "Доступ к аргументам". О том, как подобные функции создают объекты Python, речь пойдет в разделе "Создание значений Python", а о том, как они возбуждают исключения и распространяют их в код на языке Python, который вызывает эти функции, см. в главе 5. Если ваш модуль определяет новые типы Python, также называемые классами, то ваш код на языке C определяет один или несколько экземпляров структуры `PyTypeObject`. Эта тема обсуждается в разделе "Определение новых типов".

Простой пример использования этих концепций приведен в разделе "Пример простого расширения". А вот как может выглядеть простейший модуль в стиле "Hello World".

```
#include <Python.h>

static PyObject*
hello(PyObject* self)
{
    return Py_BuildValue("s", "Hello, Python extensions world!");
}

static char hello_docs[] =
    "hello(): return a popular greeting phrase\n";

static PyMethodDef hello_funcs[] = {
    {"helloworld", (PyCFunction)hello, METH_NOARGS, hello_docs},
    {NULL}
};

static struct PyModuleDef hello_module = {
   PyModuleDef_HEAD_INIT,
```

```
    "hello",
    hello_docs,
    -1,
    hello_funcs
};

PyMODINIT_FUNC
PyInit_hello(void)
{
    return PyModule_Create(&hello_module);
}
```

Функции `Py_BuildValue` передается строка C в кодировке UTF-8, а возвращаемым результатом является экземпляр строки `str` Python, для которого в версии v2 также используется кодировка UTF-8. Как уже отмечалось ранее, мы рассматриваем примеры для версии v3. О небольших отличиях инициализации модулей в версии v2 вы можете прочитать в онлайн-документации (`https://docs.python.org/2.7/howto/cporting.html`). В этом тривиальном расширении вам достаточно защитить все определение модуля конструкцией `#if PY_MAJOR_VERSION >= 3 / #endif` (поскольку в версии v2 отсутствует тип `PyModuleDef`) и соответствующим образом изменить определение функции инициализации модуля.

```
PyMODINIT_FUNC
#if PY_MAJOR_VERSION >= 3
PyInit_hello(void)
{
    return PyModule_Create(&hello_module);
#else
inithello(void)
{
    Py_InitModule3("hello",
        hello_funcs, hello_docs);
#endif
}
```

Сохраните этот код в файле *hello.c* и создайте модуль посредством сценария *setup.py* с помощью пакета `distutils`.

```
from setuptools import setup, Extension
setup(name='hello',
      ext_modules=[Extension('hello',sources=['hello.c'])])
```

После выполнения команды *python setup.py install* вы сможете использовать вновь установленный модуль в своих сценариях или в интерактивном сеансе Python.

```
>>> import hello
>>> print hello.hello()
Hello, Python extensions world!
>>>
```

Возвращаемые значения функций C API

Все функции C API в Python возвращают значение типа `int` или `PyObject*`. Большинство функций, возвращающих тип `int`, возвращает значение `0` в случае успешного выполнения и `-1` для индикации ошибок. Некоторые функции возвращают результаты, являющиеся истиной или ложью: эти функции возвращают значение `0` для указания ложного результата и целочисленное значение, не равное `0`, для указания истинного результата и никогда не индицируют ошибок. Функции, возвращающие тип `PyObject*`, возвращают значение `NULL` в случае возникновения ошибок. Подробную информацию о том, как функции на языке C обрабатывают и возбуждают ошибки, см. в главе 5.

Модуль инициализации

Функция `PyInit_x` должна содержать как минимум вызов функции `Py_Module_Create` — или, начиная с версии 3.5, функции `PyModuleDef_Init` (`https://docs.python.org/3/c-api/module.html#c.PyModuleDef_Init`) — с единственным параметром: адресом структуры `PyModuleDef`, подробно определяющей модуль. Кроме того, она может содержать один или несколько вызовов функций, приведенных в табл. 24.1, все из которых возвращают значение `-1` в случае ошибки и значение `0` в случае успешного выполнения.

Таблица 24.1. Функции модуля инициализации

Функция	Описание
`PyModule_AddIntConstant`	`int PyModule_AddIntConstant(PyObject* module, char* name, long value)` Добавляет в модуль *module* атрибут *name*, имеющий целочисленное значение *value*
`PyModule_AddObject`	`int PyModule_AddObject(PyObject* module, char* name, PyObject* value)` Добавляет в модуль *module* атрибут *name*, имеющий значение *value*, и захватывает ссылку на значение, о чем рассказано в разделе "Подсчет ссылок"
`PyModule_AddStringConstant`	`int PyModule_AddStringConstant(PyObject* module, char* name, char* value)` Добавляет в модуль *module* атрибут *name*, имеющий строковое значение *value* (в кодировке UTF-8)

Иногда для инициализации нового модуля вам может потребоваться доступ к содержимому другого модуля. Если бы вы писали код на языке Python, то вам было бы достаточно импортировать этот модуль, а затем обращаться к его атрибутам. При написании C-расширений для Python это делается почти так же просто: вызовите

функцию PyImport_Import для другого модуля, а затем функцию PyModule_GetDict для получения словаря __dict__ другого модуля.

PyImport_Import `PyObject* PyImport_Import(PyObject* name)`

Импортирует модуль, указанный в строковом объекте Python *name*, и возвращает новую ссылку на объект модуля подобно вызову `__import__(name)` в Python. Функция PyImport_Import обеспечивает наиболее высокоуровневый, простейший и чаще всего используемый способ импортирования модуля.

PyModule_GetDict `PyObject* PyModule_GetDict(PyObject* module)`

Возвращает *заимствованную* ссылку (раздел "Подсчет ссылок") на словарь модуля *module*.

Структура PyMethodDef

Чтобы добавить в модуль функции (или неспециальные методы новых типов, о чем пойдет речь в разделе "Определение новых типов"), вы должны описать эти функции или методы в массиве структур PyMethodDef и завершить его *сигнальной меткой* (т.е. структурой, поля которой заполнены значениями 0 или NULL). Структура PyMethodDef определяется следующим образом.

```
typedef struct {
    char* ml_name;       /* имя функции или метода Python */
    PyCFunction ml_meth; /* указатель на C-функцию реализации */
    int ml_flags;        /* флаг, описывающий, как передавать
                            аргументы */
    char* ml_doc;        /* строка документирования функции
                            или метода */
} PyMethodDef
```

Если функция на языке C не имеет в точности сигнатуру `PyObject* function(PyObject* self, PyObject* args)`, то второе поле должно быть приведено к типу PyCFunction. Указанная сигнатура корректна, если значение ml_flags равно METH_O (указывает на то, что функция принимает один аргумент) или METH_VARARGS (указывает на то, что функция принимает позиционные аргументы). В случае флага METH_O аргумент *args* является одиночным. В случае флага METH_VARARGS аргумент *args* является кортежем всех аргументов, для разбора которых используется функция PyArg_ParseTuple из C API. Однако поле ml_flags также может иметь значение METH_NOARGS, указывающее на то, что функция не принимает аргументов, или значение METH_KEYWORDS, указывающее на то, что функция принимает как позиционные, так и именованные аргументы. Флагу METH_NOARGS соответствует сигнатура `PyObject* function(PyObject* self)` без дополнительных аргументов. Флагу METH_KEYWORDS соответствует следующая сигнатура:

```
PyObject* function(PyObject* self, PyObject* args, PyObject* kwds)
```

Здесь *args* — кортеж позиционных аргументов, а *kwds* — словарь именованных аргументов. Для разбора обоих этих аргументов используется функция PyArg_ParseTupleAndKeywords из C API. В этих случаях требуется явное приведение второго поля к типу PyCFunction. Если функция на языке C реализует функцию модуля, то значением параметра self C-функции является NULL, независимо от значения поля ml_flags. Если C-функция реализует неспециальный метод типа расширения, то параметр self указывает на экземпляр, для которого вызывается данный метод.

Подсчет ссылок

Объекты Python хранятся в куче, и код на языке C обращается к ним посредством указателей типа PyObject*. Каждый объект PyObject подсчитывает количество ссылок на себя и уничтожает себя, если оно становится равным 0. Чтобы этот механизм работал, ваш код должен использовать макросы, которые предоставляет Python: Py_INCREF, увеличивающий счетчик ссылок на объект Python, и Py_DECREF, уменьшающий счетчик ссылок. Макросы Py_XINCREF и Py_XDECREF аналогичны макросам Py_INCREF и Py_DECREF, но их можно безболезненно применять к нулевому указателю. Макросы Py_XINCREF и Py_XDECREF выполняют неявную проверку того, является ли указатель нулевым, избавляя вас от дополнительной работы, связанной с написанием кода для явного выполнения такой проверки в тех случаях, когда достоверно не известно, является ли указатель ненулевым.

Указатель PyObject* *p*, который получает ваш код, вызывая другие функции или будучи вызванным другими функциями, является *новой ссылкой*, если код, предоставивший *p*, уже вызвал макрос Py_INCREF вместо вас. В противном случае указатель является *заимствованной ссылкой*. Говорят, что код *владеет* ссылкой, если она является новой, а не заимствованной. Вызов макроса Py_INCREF для заимствованной ссылки превращает ее в ссылку, которой владеет ваш код. Вы должны выполнять этот вызов в тех случаях, когда данную ссылку нужно использовать в вызовах кода, который может уменьшить счетчик ссылки, заимствованной вашим кодом. Вы всегда должны вызывать макрос Py_DECREF, прежде чем удалять или переопределять ссылки, которыми владеет ваш код, но никогда не должны поступать так в отношении ссылок, которыми ваш код не владеет. Следовательно, очень важно, чтобы вы хорошо понимали, какие взаимодействия передают право владения ссылками, а какие основаны только на заимствовании ссылок. Для большинства функций C API и для всех написанных вами функций, вызываемых в Python, действуют следующие правила:

- аргументы PyObject* — заимствованные ссылки;
- указатели PyObject*, возвращаемые функциями, передают право владения ссылками.

Каждое из этих двух правил допускает исключения для некоторых функций C API. Функции PyList_SetItem и PyTuple_SetItem *захватывают* ссылку на устанавливаемый ими элемент (но не на объект списка или кортежа, в котором они его

устанавливают) в том смысле, что они завладевают ею, даже если в соответствии с общими правилами соответствующий элемент должен был бы быть *заимствованной* ссылкой. Макросы `PyList_SET_ITEM` и `PyTuple_SET_ITEM` препроцессоров C, реализующие более быстрые версии одноименных функций, также захватывают ссылки. Точно так же ведет себя и функция `PyModule_AddObject`, описанная в табл. 24.1. Других исключений из первого правила нет. Рациональным зерном перечисленных исключений, понимание которого поможет вам запомнить их, является то, что созданным вами объектом будет владеть список, кортеж или модуль, поэтому семантика захвата ссылок избавляет от необходимости использования макроса `Py_DECREF` сразу же после этого.

Второе правило имеет больше исключений, чем первое. Существует несколько случаев, когда возвращенный указатель `PyObject*` является заимствованной, а не новой ссылкой. Абстрактные функции, имена которых начинаются с префиксов `PyObject_`, `PySequence_`, `PyMapping_` и `PyNumber_`, возвращают новые ссылки. Это объясняется тем, что их можно вызывать для объектов многих типов, и на возвращаемый ими результирующий объект может не быть никаких других ссылок (т.е. возвращаемый объект может быть создан на лету). Конкретные функции, имена которых начинаются с префиксов `PyList_`, `PyTuple_`, `PyDict_` и т.п., возвращают заимствованную ссылку, если семантика возвращаемого ими объекта гарантирует, что на возвращаемый объект где-то в другом месте кода должны существовать другие ссылки.

В этой главе мы проиллюстрируем все случаи исключений из указанных правил (т.е. возврат заимствованных ссылок и редкие случаи захвата ссылок из аргументов), касающиеся всех рассматриваемых функций. Если явно не указано, что функция является исключением из правил, то подразумевается, что она подчиняется правилам: ее аргументы `PyObject*`, если таковые имеются, являются заимствованными ссылками, а результат типа `PyObject*`, если таковой возвращается, является новой ссылкой.

Доступ к аргументам

Функции, в структуре `PyMethodDef` которых в поле `ml_flags` установлен флаг `METH_NOARGS`, вызываются из Python без аргументов. Соответствующая функция на языке C имеет сигнатуру с единственным аргументом: *`self`*. Если в поле `ml_flags` установлен флаг `METH_O`, то код Python вызывает функцию ровно с одним аргументом. Вторым аргументом функции на языке C является заимствованная ссылка на объект, передаваемый вызывающим кодом на языке Python в качестве значения аргумента.

Если в поле `ml_flags` установлен флаг `METH_VARARGS`, то код Python вызывает функцию с любым количеством позиционных аргументов, которые интерпретатор Python неявно собирает в кортеж. Вторым аргументом функции на языке C является заимствованная ссылка на кортеж. Затем ваш C-код вызывает функцию `PyArg_ParseTuple`, которая описана ниже.

PyArg_ParseTuple

`int PyArg_ParseTuple(PyObject* tuple, char* format,...)`

Возвращает значение `0`, если возникли ошибки, и значение, не равное `0`, в случае успешного выполнения. Аргумент *`tuple`* — указатель `PyObject*`, который был вторым аргументом С-функции. Аргумент *`format`* — это строка С, которая описывает обязательные и необязательные аргументы. Последующие аргументы функции `PyArg_ParseTuple` — это адреса переменных С, в которые должны быть помещены значения, извлеченные из кортежа. Любые переменные `PyObject*` среди переменных С являются заимствованными ссылками. В табл. 24.2 приведены строковые значения наиболее часто используемых спецификаторов формата, объединением которых в количестве от нуля и более формируется строка *`format`*.

Таблица 24.2. Спецификаторы формата для функции `PyArg_ParseTuple`

Спецификатор	Тип С	Описание
`c`	`int`	Типы Python `bytes`, `bytearray` и `str` длины `1` преобразуются в тип С `int` (`char` в версии v2)
`d`	`double`	Тип Python `float` преобразуется в тип С `double`
`D`	`Py_Complex`	Тип Python `complex` преобразуется в тип С `Py_Complex`
`f`	`float`	Тип Python `float` преобразуется в тип С `float`
`i`	`int`	Тип Python `int` преобразуется в тип С `int`
`l`	`long`	Тип Python `int` преобразуется в тип С `long`
`L`	`long long`	Тип Python `int` преобразуется в тип С `long long` (`__int64` на платформах Windows)
`O`	`PyObject*`	Получает ненулевую заимствованную ссылку на аргумент Python
`O!`	`type+PyObject*`	Аналогичен спецификатору `O`, но выполняет дополнительную проверку типа (см. ниже)
`O&`	`convert+void*`	Произвольное преобразование (см. ниже)
`s`	`cha` `*`	Строка Python без встроенных нулевых символов преобразуется в тип С `char*` (используется кодировка UTF-8)
`s#`	`char*+int`	Любая строка Python преобразуется в адрес С и длину
`t#`	`char*+int`	Доступный только для чтения односегментный буфер преобразуется в адрес С и длину

Спецификатор	Тип C	Описание
`u`	`Py_UNICODE*`	Строка Unicode без встроенных нулевых символов преобразуется в адрес C
`u#`	`Py_UNICODE*+int`	Любая строка Unicode преобразуется в адрес C и длину
`w#`	`char*+int`	Доступный для чтения/записи односегментный буфер преобразуется в адрес C и длину
`z`	`char*`	Аналогичен `s`, но принимает также значение `None` (устанавливает тип C `char*` в `NULL`)
`z#`	`char*+int`	Аналогичен `s#`, но принимает также значение `None` (устанавливает тип C `char*` в `NULL` и тип `int` в `0`)
`(...)`	`Согласно ...`	Последовательность Python поэлементно обрабатывается как один аргумент
`|`	-	Все последующие аргументы необязательны
`:`	-	Конец формата, за которым следует имя функции для вывода сообщений об ошибках
`;`	-	Конец формата, за которым следует полный текст сообщения об ошибке

Спецификаторы формата от `d` до `n` (и другие редко используемые спецификаторы для типов `unsigned char` и `short int`) принимают числовые аргументы от Python. Python выполняет приведение типов для соответствующих значений. Например, спецификатор `i` может соответствовать типу Python `float`. В этом случае дробная часть числа отбрасывается, как если бы была вызвана встроенная функция `int`. `Py_Complex` — это структура C с двумя полями — `real` и `imag`, оба типа `double`.

`O` — наиболее общий спецификатор формата, допускающий аргумент любого типа, который вы впоследствии можете проверить и/или преобразовать в случае необходимости. Спецификатор `O!` соответствует двум аргументам среди задаваемых в переменном количестве: первый — адрес объекта Python, второй — адрес `PyObject*`. Спецификатор `O!` выполняет проверку того, что соответствующее значение принадлежит к заданному типу (или любому подтипу этого типа), прежде чем устанавливать указатель `PyObject*` на это значение. В противном случае возбуждается исключение `TypeError` (вызов в целом завершается аварийно, и ошибка устанавливается в подходящий экземпляр `TypeError`; см. главу 5). Вариант `O&` также соответствует двум аргументам: первый из них — это адрес предоставляемой вами функции преобразования, второй — указатель `void*` (т.е. любой адрес). Функция преобразования должна иметь сигнатуру `int` *`convert`*`(PyObject*,`

void*). Python вызывает вашу функцию преобразования со значением, переданным из Python в качестве первого аргумента, и указателем void* из аргументов, передаваемых в переменном количестве в качестве второго аргумента. Функция преобразования должна либо вернуть значение 0 и возбудить исключение (см. главу 5) для индикации ошибки, либо вернуть значение 1 и сохранить результат посредством полученного указателя void*.

Спецификатор формата s принимает строку из Python и адрес char* (т.е. char**) среди аргументов, передаваемых в переменном количестве. Он изменяет char* таким образом, чтобы он указывал на буфер, который ваш код на языке C должен обрабатывать как доступный только для чтения символьный массив с завершающим нулевым символом (т.е. как типичную строку C; однако ваш код не должен изменять ее). Строка Python не должна содержать встроенных нулевых символов. В версии v3 результирующей кодировкой является UTF-8. Спецификатор s# ведет себя аналогичным образом, но соответствует двум аргументам: первый из них — адрес char*, второй — адрес целого числа, для которого в качестве значения устанавливается длина строки. Строка Python может содержать встроенные нулевые символы, а значит, это возможно и для буфера, на который указывает char*. Спецификаторы формата u и u# аналогичны спецификаторам s и s#, но принимают конкретно строку Unicode (в обеих версиях, v2 и v3), а указателями C должны быть Py_UNICODE*, а не char*. Макрос Py_UNICODE, определенный в заголовочном файле *Python.h*, соответствует типу символов Unicode в реализации (часто, но не всегда, это тип C wchar_t).

Спецификаторы формата t# и w# аналогичны спецификатору s#, но соответствующим аргументом Python может быть объект любого типа, следующий протоколу буфера, доступному только для чтения или для чтения/записи соответственно. Строки являются типичным примером буфера, доступного только для чтения. Экземпляры mmap и array являются типичными примерами буфера, доступного для чтения/записи, и, как и любой такой буфер, они допустимы также в тех случаях, когда требуется буфер, доступный только для чтения (т.е. допустимы также для спецификатора формата t#).

Если одним из аргументов является последовательность Python известной фиксированной длины, то разрешается использовать спецификаторы формата для каждого из ее элементов и соответствующих C-адресов из числа аргументов, передаваемых в переменном количестве, группируя спецификаторы формата с помощью круглых скобок. Например, спецификатор (ii) соответствует последовательности Python из двух чисел и двум адресам переменных типа int среди оставшихся аргументов.

Строка формата может включать вертикальную черту (|), указывающую на то, что все последующие аргументы являются необязательными. В этом случае вы должны инициализировать переменные C, адреса которых передаете последующим аргументам из числа передаваемых в переменном количестве, подходящими

значениями по умолчанию до вызова функции PyArg_ParseTuple. Функция PyArg_ParseTuple не изменяет переменные C, соответствующие необязательным аргументам, которые не были переданы из Python вашей функции на языке C в данном вызове.

В качестве примера ниже приведено начало функции, вызываемой с одним обязательным целочисленным аргументом, за которым может следовать другой целочисленный аргумент, принимающий значение по умолчанию 23, если он не указан (очень похоже на инструкцию Python `def f(x, y=23):`, за исключением того, что данная функция должна вызываться только с позиционными аргументами, и эти аргументы должны быть числовыми).

```
PyObject* f(PyObject* self, PyObject* args) {
  int x, y=23;
  if(!PyArg_ParseTuple(args, "i|i", &x, &y)
    return NULL;
  /* остальная часть функции опущена */
}
```

Строка формата может заканчиваться необязательным спецификатором *:name*, указывающим имя, которое должно использоваться в качестве имени функции при выводе любых сообщений об ошибках. Возможен и другой вариант, когда строка формата заканчивается спецификатором *;text*, указывающим текст, который должен использоваться в качестве полного сообщения об ошибке в случае обнаружения функцией PyArg_ParseTuple ошибок (эта форма используется редко).

Функция, в структуре PyMethodDef которой в поле ml_flags установлен флаг METH_KEYWORDS, принимает позиционные и именованные аргументы. Код Python вызывает эту функцию с любым количеством позиционных аргументов, которые интерпретатор Python собирает в кортеж, и именованных аргументов, которые интерпретатор Python собирает в словарь. Вторым аргументом функции на языке C является заимствованная ссылка на этот кортеж, а третьим — заимствованная ссылка на этот словарь. Затем ваш код вызывает описанную ниже функцию PyArg_ParseTupleAndKeywords.

PyArg_ParseTupleAndKeywords

`int PyArg_ParseTupleAndKeywords(PyObject* tuple, PyObject* dict, char* format, char** kwlist,...)`

Возвращает значение 0, если возникла ошибка, и значение, не равное 0, в случае успешного выполнения. Аргумент *tuple* — указатель PyObject*, который был вторым аргументом функции на языке C. Аргумент *dict* — указатель PyObject*, который был третьим аргументом функции на языке C. Аргумент *format* — то же, что и для функции PyArg_ParseTuple, за исключением того, что он не может включать спецификатор формата (...) для разбора вложенных последовательностей. Аргумент *kwlist* — массив char*,

заканчивающийся сигнальной меткой NULL, с именами параметров, следующими одно за другим. Например, приведенный ниже код на языке C

```
static PyObject*
func_c(PyObject* self, PyObject* args, PyObject* kwds)
{
    static char* argnames[] = {"x", "y", "z", NULL};
    double x, y=0.0, z=0.0;
    if(!PyArg_ParseTupleAndKeywords(
        args,kwds,"d|dd",argnames,&x,&y,&z))
        return NULL;
    /* остальная часть функции опущена */
```

эквивалентен следующему коду на языке Python:

```
def func_py(x, y=0.0, z=0.0):
    x, y, z = map(float, (x,y,z))
    # остальная часть функции опущена
```

Создание значений Python

Функции C, которые обмениваются данными с Python, часто должны создавать значения Python, как для возврата результата в виде объекта PyObject*, так и для собственных целей, таких как установка значений элементов и атрибутов. Чаще всего самый простой и удобный способ создания значений Python обеспечивает функция Py_BuildValue.

Py_BuildValue `PyObject* Py_BuildValue(char* format,...)`

Аргумент *format* — это строка C, описывающая создаваемый объект Python. Последующими аргументами функции Py_BuildValue являются значения C, на основе которых создается результат. Результатом типа PyObject* является новая ссылка. В табл. 24.3 приведены строковые значения наиболее часто используемых спецификаторов формата, объединением которых в количестве от нуля и более формируется строка *format*. Если строка *format* содержит два или более спецификатора формата или начинается с символа (и заканчивается символом), то функция Py_BuildValue создает и возвращает кортеж. В противном случае результат не является кортежем. Если вы передаете буферы, как, например, в случае спецификатора формата s#, то функция Py_BuildValue копирует данные. Поэтому после возврата из

функции Py_BuildValue вы можете изменить, удалить или освободить с помощью функции free() первоначальную копию данных. Функция Py_BuildValue всегда возвращает новую ссылку (за исключением случаев, когда используется спецификатор формата N). Вызов Py_BuildValue("") с пустой строкой *format* возвращает новую ссылку на значение None.

Таблица 24.3. Спецификаторы формата для функции `Py_BuildValue`

Спецификатор	Тип C	Описание
`B`	`unsigned char`	Тип C `unsigned char` преобразуется в тип Python `int`
`b`	`char A`	Тип C `char` преобразуется в тип Python `int`
`C`	`char`	Тип C `char` преобразуется в строку Unicode Python длиной `1` (только в версии v3)
`c`	`char`	Тип C `char` преобразуется в байтовую строку Python длиной `1` (тип `bytes` в версии v3, тип `str` в версии v2)
	`double`	Тип C `double` преобразуется в тип Python `float`
`d`	`Py_Complex`	Тип C `Py_Complex` преобразуется в тип Python `complex`
`H`	`unsigned short`	Тип C `unsigned short` преобразуется в тип Python `int`
`h`	`short A`	Тип C `short` преобразуется в тип Python `int`
`I`	`unsigned int`	Тип C `unsigned int` преобразуется в тип Python `int`
`i`	`int`	Тип C `int` преобразуется в тип Python `int`
`K`	`unsigned long long`	Тип C `unsigned long long` преобразуется в тип Python `int` (если платформа поддерживает это)
`k`	`unsigned long`	Тип C `unsigned long` преобразуется в тип Python `int`
`L`	`long long`	Тип C `long long` преобразуется в тип Python `int` (если платформа поддерживает это)
`l`	`long`	Тип C `long` преобразуется в тип Python `int`
`N`	`PyObject*`	Передает объект Python и захватывает ссылку

Спецификатор	Тип C	Описание
`O`	`PyObject*`	Передает объект Python и увеличивает на единицу его счетчик ссылок
`O&`	`convert+void*`	Произвольное преобразование (см. ниже)
`s`	`char*`	Тип C `char*` с завершающим нулевым символом преобразуется в строку Python (тип `bytes` в версии v2, декодирование в строку Unicode с использованием кодировки utf8 в версии v3), или преобразование `NULL` в `None`
`s#`	`char*+int`	Тип C `char*` и длина преобразуются в строку Python (подобно спецификатору `s`), или преобразование `NULL` в `None`
`u`	`Py_UNICODE*`	Строка C `Py_UNICODE*` (UCS-2 или UCS-4) с завершающим нулевым символом преобразуется в строку Unicode Python, или преобразование `NULL` в `None`
`u#`	`Py_UNICODE*+int`	Строка C `Py_UNICODE*` (UCS-2 или UCS-4) и длина преобразуются в строку Unicode Python, или преобразование `NULL` в `None`
`y`	`char*+int`	Строка C `char*` с завершающим нулевым символом преобразуется в байты, или преобразование `NULL` в `None` (только в версии v3)
`y#`	`char*+int`	Строка C `char*` и длина преобразуются в тип Python `bytes`, или преобразование `NULL` в `None` (только в версии v3)
`(...)`	Согласно `...`	Создает кортеж Python на основе значений C
`[...]`	Согласно `...`	Создает список Python на основе значений C
`{...}`	Согласно `...`	Создает словарь Python на основе значений C, чередуя ключи и значения (количество значений C должно быть четным)

Спецификатор `O&` соответствует двум аргументам из числа передаваемых в переменном количестве: первый — адрес предоставляемой вами функции преобразования, второй — `void*` (т.е. любой адрес). Функция преобразования должна иметь сигнатуру `PyObject* convert(void*)`. Python вызывает функцию преобразования с единственным аргументом типа `void*` из числа аргументов, передаваемых в переменном количестве. Функция преобразования должна либо возвращать значение `NULL` и возбуждать исключение (см. главу 5) для индикации ошибки, либо возвращать новую ссылку `PyObject*`, созданную на основе данных, полученных посредством указателя `void*`.

Спецификатор `{...}` создает словари на основе четного количества значений C, чередуя ключи и значения. Например, вызов `Py_BuildValue("{issi}",23,"zig","zag",42)` возвращает новую ссылку `PyObject*` для словаря `{23:'zig','zag':42}`.

Обратите внимание на важное различие между спецификаторами `N` и `O`. Спецификатор `N` *захватывает* ссылку из соответствующего значения `PyObject*`, взятого из аргументов, задаваемых в переменном количестве, поэтому его удобно использовать для создания объекта, включающего ссылку, которой вы владеете и счетчик которой должны были бы так или иначе уменьшить на единицу. Спецификатор `O` не захватывает ссылку, поэтому он подходит для создания объекта, включающего ссылку, которой вы не владеете, или ссылку, которую вы также должны хранить в другом месте.

Исключения

Для распространения исключений, возбуждаемых из других вызываемых вами функций, достаточно вернуть из вашей функции на языке C значение `NULL` в качестве результата типа `PyObject*`. Для возбуждения собственных исключений нужно задать индикатор текущего исключения и вернуть значение `NULL`. Встроенные классы исключений Python (см. раздел "Классы стандартных исключений" в главе 5) доступны глобально с именами, начинающимися с префикса `PyExc_`, такими как `PyExc_AttributeError`, `PyExc_KeyError` и т.п. Кроме того, ваш модуль расширения может предоставить и использовать собственные классы исключений. Наиболее часто используемые функции C API, связанные с возбуждением исключений, описаны ниже.

`PyErr_Format`

```
PyObject* PyErr_Format(PyObject* type,
char* format,...)
```

Возбуждает исключение класса *`type`*, которое должно быть либо встроенным, таким как `PyExc_IndexError`, либо принадлежать классу исключений, созданному с помощью функции `PyErr_NewException`. Создает соответствующее значение на основе строки *`format`*, синтаксис которой аналогичен синтаксису функции `printf` языка C, и последующих значений C, которые указаны с помощью аргументов, задаваемых в переменном количестве. Возвращает значение `NULL`, поэтому ваш код может просто использовать следующую инструкцию:

```
return PyErr_Format(PyExc_KeyError,
    "Unknown key name (%s)", thekeystring);
```

PyErr_NewException

```
PyObject* PyErr_NewException(char* name,
PyObject* base, PyObject* dict)
```

Расширяет класс исключений `base` путем добавления в него дополнительных атрибутов и методов из словаря *dict* (обычно этот аргумент имеет значение `NULL`, означающее отсутствие дополнительных атрибутов и методов класса), создавая новый класс исключений *name* (строка *name* должна задаваться в виде "*modulename.classname*"), и возвращает новую ссылку на новый объект класса. Если аргумент *base* имеет значение *NULL*, то в качестве базового используется класс `PyExc_Exception`. Обычно эту функцию вызывают в процессе инициализации объекта модуля.

```
PyModule_AddObject(module, "error",
    PyErr_NewException(
    "mymod.error", NULL, NULL));
```

PyErr_NoMemory

```
PyObject* PyErr_NoMemory()
```

Возбуждает ошибку нехватки памяти и возвращает значение `NULL`, поэтому ваш код может использовать следующую инструкцию:

```
return PyErr_NoMemory();
```

PyErr_SetObject

```
void PyErr_SetObject(PyObject* type,
PyObject* value)
```

Возбуждает исключение класса *type*, которое должно быть либо встроенным, таким как `PyExc_KeyError`, либо принадлежать классу исключений, созданному с помощью функции `PyErr_NewException`, со значением *value* в качестве связанного значения (заимствованная ссылка). Функция `PyErr_SetObject` имеет тип `void` (т.е. не возвращает значение).

PyErr_SetFromErrno

```
PyObject* PyErr_SetFromErrno(PyObject* type)
```

Возбуждает исключение класса *type*, которое должно быть либо встроенным, таким как `PyExc_OSError`, либо принадлежать классу исключений, созданному с помощью функции `PyErr_NewException`. Получает подробную информацию из кода ошибки, который устанавливается многими функциями стандартной библиотеки C и системными вызовами, а также с помощью функции `strerror` стандартной библиотеки C, которая транслирует коды ошибок в соответствующие строки. Возвращает значение `NULL`, поэтому ваш код может использовать следующую инструкцию:

```
return PyErr_SetFromErrno(PyExc_IOError);
```

PyErr_SetFromErrnoWithFilename

```
PyObject* PyErr_SetFromErrnoWithFilename(PyObject* type, char* filename)
```

Аналогична функции `PyErr_SetFromErrno`, но также предоставляет строку *filename* как часть значения исключения. Если аргумент *filename* имеет значение `NULL`, то эта функция работает аналогично функции `PyErr_SetFromErrno`.

Иногда вашему коду на языке C может потребоваться обработать исключение и продолжить выполнение аналогично тому, как это делается в коде на языке Python с помощью инструкции `try/except`. Наиболее распространенными функциями C API, имеющими отношение к перехвату исключений, являются следующие.

PyErr_Clear

```
void PyErr_Clear()
```

Сбрасывает индикатор ошибки. В отсутствие ошибок никакие действия не выполняются.

PyErr_ExceptionMatches

```
int PyErr_ExceptionMatches(PyObject* type)
```

Вызывайте эту функцию только при возникновении ошибки, иначе вся программа может завершиться аварийно. Возвращает значение, не равное `0`, если текущим исключением является экземпляр типа *type* или любой подкласс *type*, или значение `0`, если текущее исключение не принадлежит данному классу.

PyErr_Occurred

```
PyObject* PyErr_Occurred()
```

Возвращает значение `NULL` в отсутствие текущих ошибок. В противном случае возвращает заимствованную ссылку на тип текущего исключения. (*Не* используйте конкретное возвращаемое значение. Вместо этого вызывайте функцию `PyErr_ExceptionMatches`, чтобы перехватывались также исключения подклассов, что является нормальным и ожидаемым порядком действий.)

PyErr_Print

```
void PyErr_Print()
```

Вызывайте эту функцию только при возникновении ошибки, иначе вся программа может завершиться аварийно. Выводит стандартную трассировочную информацию в стандартный поток вывода ошибок `sys.stderr`, после чего сбрасывает индикатор ошибки.

Если вы нуждаетесь в обработке ошибок более сложными способами, изучите другие функции C API, связанные с обработкой ошибок, такие как `PyErr_Fetch`, `PyErr_Normalize`, `PyErr_GivenExceptionMatches` и `PyErr_Restore`. Эти дополнительные и редко используемые возможности в данной книге не рассматриваются.

Функции абстрактного слоя

В типичных случаях код C-расширения нуждается в использовании функциональности Python. Например, вашему коду может понадобиться проверить или установить атрибуты и элементы объектов Python, вызвать встроенные или написанные на языке Python функции и методы и т.п. В большинстве случаев наилучшим подходом для вашего кода является вызов функций из *абстрактного слоя* C API. Это функции, которые вы можете вызывать для любого объекта Python (имена данных функций начинаются с префикса `PyObject_`) или для любого из объектов широкой категории, таких как отображения, числа или последовательности (имена таких функций начинаются с префиксов `PyMapping_`, `PyNumber_` и `PySequence_` соответственно).

Многие из функций, вызываемых для объектов конкретных типов в пределах этих категорий, дублируют функциональность, обеспечиваемую функциями `PyObject_`. В подобных случаях вы практически всегда должны отдавать предпочтение более общим функциям `PyObject_`. Аналогичные им функции конкретных категорий, которые оказываются чуть ли не лишними, в данной книге не рассматриваются.

Функции абстрактного слоя возбуждают исключения Python, если вы вызываете их для объектов, к которым они не применимы. Все эти функции принимают заимствованные ссылки для аргументов `PyObject*` и возвращают новую ссылку (значение `NULL` для исключений), если они возвращают результат типа `PyObject*`.

Наиболее часто используемые функции абстрактного слоя приведены в табл. 24.4.

Таблица 24.4. Функции абстрактного слоя

Функция	Описание
`PyCallable_Check`	`int PyCallable_Check(PyObject* x)` Возвращает истинное значение, если `x` — вызываемый объект; аналогична вызову `callable(x)`
`PyIter_Check`	`int PyIter_Check(PyObject* x)` Возвращает истинное значение, если `x` — итератор
`PyIter_Next`	`PyObject* PyIter_Next(PyObject* x)` Возвращает следующий элемент из итератора `x`. Возвращает значение `NULL` без возбуждения исключения, если итерирование с помощью `x` завершено (т.е. если вызов `next(x)` приведет к возбуждению исключения `StopIteration`)
`PyNumber_Check`	`int PyNumber_Check(PyObject* x)` Возвращает истинное значение, если `x` — число
`PyObject_Call`	`PyObject* PyObject_Call(PyObject* f, PyObject* args, PyObject* kwds)` Вызывает вызываемый объект `f` Python с позиционными аргументами в кортеже `args` (может быть пустым, но

Функция	Описание
	никогда не может иметь значение NULL) и именованными аргументами в словаре *kwds*. Возвращает результат этого вызова. Аналогична вызову *f*(**args*, ***kwds*)
PyObject_CallObject	PyObject* PyObject_CallObject(PyObject* *f*, PyObject* *args*) Вызывает вызываемый объект *f* Python с позиционными аргументами в кортеже *args* (может иметь значение NULL). Возвращает результат вызова. Аналогична вызову *f*(**args*)
PyObject_CallFunction	PyObject* PyObject_CallFunction(PyObject* *f*, char* *format*,...) Вызывает вызываемый объект *f* Python с позиционными аргументами, описанными строкой формата *format* с использованием тех же спецификаций формата, что и для функции Py_BuildValue, рассмотренной в разделе "Создание значений Python". Если аргумент *format* имеет значение NULL, вызывает *f* без аргументов. Возвращает результат этого вызова
PyObject_CallFunctionObjArgs	PyObject* PyObject_CallFunctionObjArgs(PyObject* *f*,..., NULL) Вызывает вызываемый объект *f* Python с позиционными аргументами PyObject*, переданными в количестве от нуля и более аргументов. Возвращает результат этого вызова
PyObject_CallMethod	PyObject* PyObject_CallMethod(PyObject* *x*, char* *method*,char* *format*,...) Вызывает метод *method* объекта *x* Python с позиционными аргументами, описанными строкой формата *format*, используя те же спецификаторы формата, что и для функции Py_BuildValue. Если аргумент *format* имеет значение NULL, вызывает метод *method* без аргументов. Возвращает результат этого вызова
PyObject_CallMethodObjArgs	PyObject* PyObject_CallMethodObjArgs(PyObject* *x*, char* *method*,..., NULL) Вызывает метод *method* объекта *x* Python с позиционными аргументами PyObject*, переданными в количестве от нуля и более аргументов. Возвращает результат этого вызова
PyObject_DelAttrString	int PyObject_DelAttrString(PyObject* *x*, char* *name*) Удаляет атрибут *name* объекта *x* аналогично инструкции del *x.name*

Функция	Описание
PyObject_DelItem	`int PyObject_DelItem(PyObject* x, PyObject* key)` Удаляет элемент объекта *x* с ключом (или индексом) *key* аналогично инструкции `del x[key]`
PyObject_DelItemString	`int PyObject_DelItemString(PyObject* x, char* key)` Удаляет элемент объекта *x* с ключом *key* аналогично инструкции `del x[key]`
PyObject_GetAttrString	`PyObject* PyObject_GetAttrString(PyObject* x, char* name)` Возвращает атрибут *name* объекта *x* аналогично обращению *x.name*
PyObject_GetItem	`PyObject* PyObject_GetItem(PyObject* x, PyObject* key)` Возвращает элемент объекта *x* с ключом (или индексом) *key* аналогично обращению *x*[*key*]
PyObject_GetItemString	`int PyObject_GetItemString(PyObject* x, char* key)` Возвращает элемент объекта *x* с ключом *key* аналогично обращению *x*[*key*]
PyObject_GetIter	`PyObject* PyObject_GetIter(PyObject* x)` Возвращает итератор по *x* аналогично вызову `iter(x)`
PyObject_HasAttrString	`int PyObject_HasAttrString(PyObject* x, char* name)` Возвращает значение `True`, если объект *x* имеет атрибут *name*, аналогично вызову `hasattr(x, name)`
PyObject_IsTrue	`int PyObject_IsTrue(PyObject* x)` Возвращает значение `True`, если *x* является истинным значением в Python, аналогично вызову `bool(x)`
PyObject_Length	`int PyObject_Length(PyObject* x)` Возвращает длину объекта *x* аналогично вызову `len(x)`
PyObject_Repr	`PyObject* PyObject_Repr(PyObject* x)` Возвращает подробное строковое представление объекта *x* аналогично вызову `repr(x)`
PyObject_RichCompare	`PyObject* PyObject_RichCompare(PyObject* x, PyObject* y, int op)` Применяет к объектам *x* и *y* операцию сравнения, заданную аргументом *op*, и возвращает результат в виде

Функция	Описание
	объекта Python. Аргумент *op* может принимать значения `Py_EQ`, `Py_NE`, `Py_LT`, `Py_LE`, `Py_GT` или `Py_GE`, соответствующие операциям сравнения Python *x*==*y*, *x*!=*y*, *x*<*y*, *x*<=*y*, *x*>*y* или *x*>=*y*
PyObject_RichCompareBool	`int PyObject_RichCompareBool(PyObject* x, PyObject* y,int op)` Аналогична функции `PyObject_RichCompare`, но возвращает значение `0` для ложных значений и `1` для истинных значений
PyObject_SetAttrString	`int PyObject_SetAttrString(PyObject* x, char* name, PyObject* v)` Устанавливает для атрибута *name* объекта *x* значение *v* аналогично операции присваивания *x*.*name*=*v*
PyObject_SetItem	`int PyObject_SetItem(PyObject* x, PyObject* k, PyObject *v)` Устанавливает для элемента *x* с ключом (или индексом) *key* значение *v* аналогично операции присваивания *x*[*key*]=*v*
PyObject_SetItemString	`int PyObject_SetItemString(PyObject* x, char* key, PyObject *v)` Устанавливает для элемента *x* с ключом *key* значение *v* аналогично операции присваивания *x*[*key*]=*v*
PyObject_Str	`PyObject* PyObject_Str(PyObject* x)` Возвращает удобочитаемую строковую форму объекта *x* (строка Unicode в версии v3, тип `bytes` в версии v2) аналогично вызову `str(x)`. Для получения результата в виде байтов в версии v3 используйте функцию `PyObject_Bytes`. Для получения результата в виде строки `unicode` в версии v2 используйте функцию `PyObject_Unicode`
PyObject_Type	`PyObject* PyObject_Type(PyObject* x)` Возвращает тип объекта *x* аналогично вызову `type(x)`
PySequence_Contains	`int PySequence_Contains(PyObject* x, PyObject* v)` Возвращает значение `True`, если *v* является элементом *x*, аналогично инструкции *v* `in` *x*
PySequence_DelSlice	`int PySequence_DelSlice(PyObject* x, int start, int stop)` Удаляет срез *x* в пределах индексов от *start* до *stop* аналогично инструкции `del` *x*[*start*:*stop*]

Функция	Описание
PySequence_Fast	`PyObject* PySequence_Fast(PyObject* x)` Возвращает новую ссылку на кортеж с теми же элементами, что и *x*, но если объект *x* — список, то возвращает новую ссылку на *x*. Если нужно получить большое количество элементов произвольной последовательности *x*, то наиболее быстрый способ для этого — однократный вызов *t*=`PySequence_Fast(x)` с последующими многократными вызовами `PySequence_Fast_GET_ITEM(t, i)` нужное количество раз и завершающим вызовом `Py_DECREF(t)`
PySequence_Fast_GET_ITEM	`PyObject* PySequence_Fast_GET_ITEM(PyObject* x, int i)` Возвращает элемент *i* объекта *x*, где *x* должен быть возвращаемым результатом вызова функции `PySequence_Fast`, не равным `NULL`, и 0<=*i*<`PySequence_Fast_GET_SIZE(t)`. Нарушение этих условий может привести к аварийному завершению программы. Этот подход оптимизирован для повышения производительности, но не безопасности
PySequence_Fast_GET_SIZE	`int PySequence_Fast_GET_SIZE(PyObject* x)` Возвращает длину объекта *x*. Объект *x* должен быть возвращаемым результатом вызова функции `PySequence_Fast`, не равным `NULL`
PySequence_GetSlice	`PyObject* PySequence_GetSlice(PyObject* x, int start, int stop)` Возвращает срез объекта *x* в пределах индексов от *start* до *stop* аналогично операции взятия среза *x*[*start*:*stop*]
PySequence_List	`PyObject* PySequence_List(PyObject* x)` Возвращает новый объект списка с теми же элементами, что и в *x*, аналогично вызову `list(x)`
PySequence_SetSlice	`int PySequence_SetSlice(PyObject* x, int start, int stop, PyObject* v)` Устанавливает для среза объекта *x* в пределах индексов от *start* до *stop* значение *v* аналогично операции присваивания *x*[*start*:*stop*]=*v*. Как и в эквивалентной инструкции Python, объект *v* также должен быть последовательностью
PySequence_Tuple	`PyObject* PySequence_Tuple(PyObject* x)` Возвращает новую ссылку на кортеж с теми же элементами, что и в *x*, аналогично вызову `tuple(x)`

Другие функции, имена которых начинаются с префикса PyNumber_, позволяют выполнять операции над числами. Унарные функции PyNumber, имеющие единственный аргумент *x* типа PyObject* и возвращающие ссылку на объект PyObject*, приведены в табл. 24.5 вместе с их эквивалентами в Python.

Таблица 24.5. Унарные функции PyNumber

Функция	Эквивалент в Python
PyNumber_Absolute	abs(x)
PyNumber_Float	float(x)
PyNumber_Int	int(x) (только в версии v2)
PyNumber_Invert	~x
PyNumber_Long	long(x) (int(x) в версии v3)
PyNumber_Negative	-x
PyNumber_Positive	+x

Бинарные функции PyNumber, имеющие два аргумента *x* и *y* типа PyObject* и возвращающие объект PyObject*, приведены в табл. 24.6.

Таблица 24.6. Бинарные функции PyNumber

Функция	Эквивалент Python
PyNumber_Add	x + y
PyNumber_And	x & y
PyNumber_Divide	x // y
PyNumber_Divmod	divmod(x, y)
PyNumber_FloorDivide	x // y
PyNumber_Lshift	x << y
PyNumber_Multiply	x * y
PyNumber_Or	x | y
PyNumber_Remainder	x % y
PyNumber_Rshift	x >> y
PyNumber_Subtract	x - y
PyNumber_TrueDivide	x / y (без усечения)
PyNumber_Xor	x ^ y

У каждой бинарной функции PyNumber имеется эквивалент, выполняемый на месте, имя которого начинается с префикса PyNumber_InPlace, например PyNumber_InPlaceAdd и др. Версия, выполняемая на месте, пытается изменить на месте первый аргумент, если это возможно, и в любом случае возвращает новую ссылку на результат, будь то первый аргумент (измененный) или новый объект. Встроенные числовые типы Python неизменяемы, поэтому, если первый аргумент является

числом встроенного типа, версии, выполняемые на месте, работают точно так же, как и обычные версии. Функция PyNumber_Divmod возвращает кортеж с двумя элементами (частное от деления и остаток) и не имеет эквивалента, выполняемого на месте.

Существует единственная тернарная функция PyNumber: PyNumber_Power.

PyNumber_Power `PyObject* PyNumber_Power(PyObject* x, PyObject* y, PyObject* z)`

Если *z* — значение Py_None, то данная функция возвращает аргумент *x*, возведенный в степень *y*, аналогично выражению *x****y* или, что эквивалентно, вызову pow(*x*, *y*). В противном случае функция возвращает *x****y*%*z* аналогично вызову pow(*x*, *y*, *z*). Выполняемая на месте версия — функция PyNumber_InPlacePower.

Функции конкретного слоя

Каждый конкретный встроенный тип объектов Python предоставляет встроенные функции, воздействующие на экземпляры данного типа, с именами, начинающимися с префикса Pytype_ (например, PyInt_ для функций, связанных с объектами Python типа int). В большинстве случаев такие функции дублируют функциональность функций абстрактного слоя или вспомогательных функций, рассмотренных ранее в этой главе, таких как Py_BuildValue, которые могут генерировать объекты многих типов. В данном разделе мы обсудим лишь некоторые часто используемые функции конкретного слоя, которые предоставляют уникальную функциональность, обеспечивают существенное повышение быстродействия или большие удобства в работе. Большинство типов допускает проверку принадлежности объекта данному типу путем вызова функции Pytype_Check, которой передаются также экземпляры подтипов, или функции Pytype_CheckExact, которой передаются лишь экземпляры определенного типа, но не его подтипов. Сигнатуры этих функций совпадают с сигнатурой функции PyIter_Check, описанной в табл. 24.4.

Все указанные ниже функции, имена которых начинаются с префикса PyString, в версии v3 имеют фактические имена, начинающиеся с префикса PyBytes (и работают с объектами bytes в Python, а не с объектами str), но имена, начинающиеся с префикса PyString, остаются доступными (в качестве синонимов, реализованных с помощью макроопределений препроцессора C) из соображений удобства. Имена функций конкретного слоя, предназначенных для работы с экземплярами str в версии v3 и экземплярами unicode в версии v2, начинаются с префикса PyUnicode, но обычно вместо них лучше использовать соответствующие функции абстрактного слоя, в связи с чем подобные эквиваленты, относящиеся к конкретному слою, в данной книге не рассматриваются.

PyDict_GetItem `PyObject* PyDict_GetItem(PyObject* x, PyObject* key)`

Возвращает заимствованную ссылку на элемент с ключом *key* в словаре *x*.

PyDict_GetItemString `int PyDict_GetItemString(PyObject* x, char* key)`

Возвращает заимствованную ссылку на элемент с ключом *key*, заданным в виде строки с завершающим нулевым символом, в словаре *x*.

PyDict_Next `int PyDict_Next(PyObject* x, int* pos, PyObject**k, PyObject** v)`

Выполняет итерации по элементам словаря *x*. Вы должны инициализировать **pos* значением 0 в начале итерирования: функция `PyDict_Next` использует и обновляет **pos* для отслеживания указываемой им позиции. Для каждой успешной итерации функция возвращает значение 1. Если элементов больше нет, возвращается значение 0. Значения **k* и **v* обновляются таким образом, чтобы они указывали на следующие ключ и значение соответственно (заимствованные ссылки) на каждом шаге итерации, который возвращает значение 1. Если вас не интересует ключ или значение, можете передать значение `NULL` в качестве *k* или *v*. В процессе итерирования нельзя никоим образом изменять множество ключей словаря *x*, но допускается изменение значений словаря *x*, если множество его ключей остается тем же.

PyDict_Merge `int PyDict_Merge(PyObject* x, PyObject* y, int override)`

Обновляет словарь *x* посредством слияния с ним элементов словаря *y*. Аргумент *override* определяет, что произойдет, если ключ *k* имеется как в словаре *x*, так и в словаре *y*: если *override* имеет значение *0*, то *x*[*k*] сохраняет прежнее значение, в противном случае *x*[*k*] заменяется значением *y*[*k*].

PyDict_MergeFromSeq2 `int PyDict_MergeFromSeq2(PyObject* x, PyObject* y, int override)`

Подобна функции `PyDict_Merge`, за исключением того, что *y* является не словарем, а последовательностью последовательностей, в которой каждая подпоследовательность имеет длину 2 и используется в качестве пары (*key*, *value*).

PyFloat_AS_DOUBLE `double PyFloat_AS_DOUBLE(PyObject* x)`

Очень быстро возвращает значение с двойной точностью C, эквивалентное значению `float` *x* в Python, без проверки ошибок.

PyList_New `PyObject* PyList_New(int length)`

Возвращает новый, неинициализированный список заданной длины. Вы должны самостоятельно инициализировать этот список, обычно посредством вызова функции `PyList_SET_ITEM` *length* раз.

PyList_GET_ITEM

`PyObject* PyList_GET_ITEM(PyObject* x, int pos)`

Возвращает элемент списка *x*, заданный своей позицией *pos*, без проверки ошибок.

PyList_SET_ITEM

`int PyList_SET_ITEM(PyObject* x, int pos, PyObject* v)`

Устанавливает для элемента списка *x*, заданного своей позицией *pos*, значение *v* без проверки ошибок. Захватывает ссылку на *v*. Используйте данную функцию только сразу же после создания списка *x* с помощью функции `PyList_New`.

PyString_AS_STRING

`char* PyString_AS_STRING(PyObject* x)`

Очень быстро возвращает указатель на внутренний буфер строки *x* без проверки ошибок. Вы не должны никоим образом изменять этот буфер, за исключением первоначального выделения для него области памяти посредством вызова функции `PyString_FromStringAndSize(NULL, size)`.

PyString_AsStringAndSize

`int PyString_AsStringAndSize(PyObject* x, char** buffer, int* length)`

Помещает указатель на внутренний буфер строки *x* в `*buffer`, а длину *x* — в `*length`. Вы не должны никоим образом изменять этот буфер, за исключением первоначального выделения для него области памяти посредством вызова функции `PyString_FromStringAndSize(NULL, size)`.

PyString_FromFormat

`PyObject* PyString_FromFormat(char* format,...)`

Возвращает строку Python, создаваемую на основе строки формата *format*, синтаксис которой аналогичен синтаксису функции `printf`, а последующие значения C указываются в виде аргументов, задаваемых в переменном количестве (`...`).

PyString_FromStringAndSize

`PyObject* PyString_FromFormat(char* data, int size)`

Возвращает строку Python длины *size*, копируя *size* байтов из *data*. Если аргумент *data* имеет значение `NULL`, то строка Python не инициализируется, и вы должны самостоятельно инициализировать ее. Вы можете получить указатель на внутренний буфер строки посредством вызова `PyString_AS_STRING`.

PyTuple_New

`PyObject* PyTuple_New(int length)`

Возвращает новый, неинициализированный кортеж заданной длины *length*. Вы должны самостоятельно инициализировать этот кортеж, обычно посредством вызова функции `PyTuple_SET_ITEM` *length* раз.

PyTuple_GET_ITEM `PyObject* PyTuple_GET_ITEM(PyObject* x, int pos)`

Возвращает элемент кортежа *x*, заданный своей позицией *pos*, без проверки ошибок.

PyTuple_SET_ITEM `int PyTuple_SET_ITEM(PyObject* x, int pos, PyObject* v)`

Устанавливает для элемента кортежа *x*, заданного своей позицией *pos*, значение *v* без проверки ошибок. Захватывает ссылку на *v*. Используйте эту функцию только сразу же после создания нового кортежа *x* с помощью функции `PyTuple_New`.

Пример простого расширения

Листинг 24.1 иллюстрирует использование в Python функций C API `PyDict_Merge` и `PyDict_MergeFromSeq2`. Метод `update` словарей работает подобно функции `PyDict_Merge` с аргументом *override*=1, но листинг 24.1 имеет несколько более общий характер.

Листинг 24.1. Простое расширение модуля merge.c

```
#include <Python.h>

static PyObject*
merge(PyObject* self, PyObject* args, PyObject* kwds)
{
    static char* argnames[] = {"x","y","override",NULL};
    PyObject *x, *y;
    int override = 0;
    if(!PyArg_ParseTupleAndKeywords(args, kwds, "O!O|i", argnames,
        &PyDict_Type, &x, &y, &override))
            return NULL;
    if(-1 == PyDict_Merge(x, y, override)) {
        if(!PyErr_ExceptionMatches(PyExc_TypeError))
            return NULL;
        PyErr_Clear();
        if(-1 == PyDict_MergeFromSeq2(x, y, override))
            return NULL;
    }
    return Py_BuildValue("");
}

static char merge_docs[] = "\
merge(x,y,override=False): merge into dict x the items of dict y (or\n\
  the pairs that are the items of y, if y is a sequence), with\n\
  optional override. Alters dict x directly, returns None.\n\
";

static PyObject*
```

```
mergenew(PyObject* self, PyObject* args, PyObject* kwds)
{
    static char* argnames[] = {"x","y","override",NULL};
    PyObject *x, *y, *result;
    int override = 0;
    if(!PyArg_ParseTupleAndKeywords(args, kwds, "O!O|i", argnames,
        &PyDict_Type, &x, &y, &override))
            return NULL;
    result = PyObject_CallMethod(x, "copy", "");
    if(!result)
        return NULL;
    if(-1 == PyDict_Merge(result, y, override)) {
        if(!PyErr_ExceptionMatches(PyExc_TypeError))
            return NULL;
        PyErr_Clear( );
        if(-1 == PyDict_MergeFromSeq2(result, y, override))
            return NULL;
    }
    return result;
}

static char mergenew_docs[] = "\
mergenew(x,y,override=False): merge into dict x the items of dict y\n\
  (or the pairs that are the items of y, if y is a sequence), with\n\
  optional override. Does NOT alter x, but rather returns the\n\
  modified copy as the function's result.\n\
";

static PyMethodDef merge_funcs[] = {
    {"merge", (PyCFunction)merge, METH_KEYWORDS, merge_docs},
    {"mergenew", (PyCFunction)mergenew, METH_KEYWORDS, mergenew_docs},
    {NULL}
};

static char merge_module_docs[] = "Example extension module";

static struct PyModuleDef merge_module = {
   PyModuleDef_HEAD_INIT,
   "merge",
   merge_module_docs,
   -1,
   merge_funcs
};

PyMODINIT_FUNC
PyInit_merge(void)
{
    return PyModule_Create(&merge_module);
}
```

В этом примере объявляются как static все функции и глобальные переменные в файле исходного кода на языке C, за исключением функции PyInit_merge (в версии v3; в версии v2 она называлась бы initmerge), которая должна быть видимой снаружи, чтобы Python мог вызывать ее. Поскольку для Python доступ к функциям и переменным открывается через структуры PyMethodDef, их имена не обязательно должны быть непосредственно видимы Python. Поэтому лучше всего объявить их статическими: это является гарантией того, что их имена не окажутся случайно в глобальном пространстве имен всей программы, как это могло бы произойти на некоторых платформах с риском возникновения конфликтов и ошибок.

Строка формата "O!O|i", передаваемая функции PyArg_ParseTupleAndKeywords, указывает на то, что функция merge получает от Python три аргумента: объект с ограничением по типу, обобщенный объект и необязательный целочисленный аргумент. В то же время данный формат указывает на то, что переменная часть аргументов функции PyArg_ParseTupleAndKeywords должна содержать четыре адреса в следующем порядке: адрес объекта типа Python, два адреса переменных PyObject* и адрес переменной типа int. Целочисленная переменная должна быть предварительно инициализирована требуемым значением по умолчанию, поскольку соответствующий аргумент Python является необязательным.

И действительно, вслед за аргументом argnames код передает аргумент &PyDict_Type (т.е. адрес объекта типа словаря). Затем передаются адреса двух переменных типа PyObject*. Наконец, код передает адрес переменной override — целого числа, ранее инициализированного значением 0, поскольку по умолчанию, если аргумент override не передается явным образом из Python, он имеет смысл "без замены значений". Если функция PyArg_ParseTupleAndKeywords возвращает значение 0, то код немедленно возвращает значение NULL для распространения исключения. Это автоматически диагностирует большинство случаев передачи неверных аргументов кодом Python в нашу новую функцию merge.

Если с аргументами все нормально, то проверяется значение, возвращаемое функцией PyDict_Merge, которая выполняется успешно, если y — словарь. Если функция PyDict_Merge возбуждает исключение TypeError, указывающее на то, что y не является словарем, то код сбрасывает ошибку и приступает к следующей проверке, на этот раз возвращаемого значения функции PyDict_MergeFromSeq2, которая выполняется успешно, если y — последовательность пар. Если не проходит и эта проверка, то код возвращает значение NULL для распространения исключения. В противном случае код возвращает значение None простейшим способом (т.е. посредством инструкции return Py_BuildValue("")) для индикации успешного выполнения.

Функция mergenew в целом дублирует функциональность функции merge. Однако функция mergenew не изменяет свои аргументы, а создает и возвращает в качестве результата новый словарь. Функция C API PyObject_CallMethod позволяет функции mergenew вызвать метод copy для первого из переданных ей аргументов Python — объекта словаря, и получить новый объект словаря, который она затем изменяет (с использованием той же самой логики, что и в функции merge). После этого

в качестве результата работы функции возвращается измененный словарь (таким образом, в данном случае нет необходимости в вызове функции Py_BuildValue).

Код листинга 24.1 должен находиться в файле *merge.c*. Создайте в том же каталоге файл *setup.py*, который должен содержать следующий сценарий.

```
from setuptools import setup, Extension
setup(name='merge',
      ext_modules=[Extension('merge',sources=['merge.c'])])
```

Выполните в командной строке в этом каталоге команду **python setup.py install** (от имени пользователя, имеющего соответствующие полномочия для записи в место установки используемого вами интерпретатора Python, или используя команду sudo в Unix-подобных системах, если это потребуется, или же, что еще лучше, воспользуйтесь виртуальным окружением). В результате выполнения этой команды создается динамически загружаемая библиотека для модуля расширения merge, которая копируется в соответствующий каталог вашей установки Python. Теперь код на языке Python (выполняемый в соответствующем виртуальном окружении, если вы, как это рекомендуется, используете модуль venv) может использовать данный модуль.

```
import merge
x = {'a':1,'b':2 }
merge.merge(x,[['b',3],['c',4]])
print(x)                          # вывод: {'a':1, 'b':2, 'c':4 }
print(merge.mergenew(x,{'a':5,'d':6},override=1))
# вывод: {'a':5, 'b':2, 'c':4, 'd':6 }
print(x)                          # вывод: {'a':1, 'b':2, 'c':4 }
```

Этот пример демонстрирует различие между функциями merge (изменяет свой первый аргумент) и mergenew (возвращает новый объект и не изменяет свой первый аргумент). В нем также показано, что второй аргумент может быть либо словарем, либо последовательностью подпоследовательностей, состоящих из двух элементов. Кроме того, продемонстрировано выполнение программы в режиме по умолчанию (когда ключи, существующие в первом аргументе, остаются неизменными) и с опцией override (когда приоритет имеют ключи из второго аргумента, как в методе update словарей Python).

Определение новых типов

У вас часто будет возникать желание определить в своем модуле расширения новые типы и сделать их доступными для Python. Определение типа хранится в структуре PyTypeObject. Большинство полей этой структуры — указатели на функции. Некоторые поля указывают на другие структуры, которые, в свою очередь, являются блоками указателей на функции. Кроме того, структура PyTypeObject включает несколько полей, которые определяют имя, размер и детали поведения (флаги опций) типа. Почти для всех полей структуры PyTypeObject можно оставить установленные

для них значения NULL, если вы не предоставляете соответствующую функциональность. Вы можете предоставить объекту базовую функциональность стандартными способами, задав в некоторых полях указатели на функции Python/C API.

Наилучшим способом реализации типа является копирование из исходных файлов Python одного из трех файлов, находящихся в каталоге *Modules*, которые Python специально предлагает для облегчения этой задачи, и внести в него соответствующие изменения. Речь идет о файлах *xxlimited.c* (только в версии v3), *xxmodule.c* и *xxsubtype.c* (последний сфокусирован на создании подклассов встроенных типов и предоставляет два типа, предназначенных для создания подклассов типов `list` и `dict`).

Для получения более подробной информации относительно структуры `PyTypeObject` и других родственных ей структур обратитесь к онлайн-документации (`https://docs.python.org/3/c-api/typeobj.html`). Заголовочный файл *Include/object.h*, находящийся среди исходных файлов Python, содержит объявления этих типов, а также несколько важных комментариев, которые вам будет полезно изучить.

Данные экземпляров

Чтобы предоставить экземпляр собственного типа, объявите структуру C, начинающуюся сразу же за открывающей фигурной скобкой, с макроопределением `PyObject_HEAD`. Этот макрос раскрывается в поля данных, которыми должна начинаться ваша структура, чтобы быть объектом Python. В число этих полей входит счетчик ссылок и указатель на тип экземпляра. Любой указатель на вашу структуру может быть корректно приведен к типу `PyObject*`. Такой подход можно рассматривать как своеобразную реализацию механизма (одиночного) наследования на уровне языка C.

Структура `PyTypeObject`, определяющая характеристики и поведение вашего типа, должна содержать размер вашей структуры экземпляра, а также указатели на функции C, оперирующие вашей структурой. Таким образом, обычно вы помещаете структуру `PyTypeObject` ближе к концу исходного кода модуля на языке C, после определений структуры экземпляра и всех функций, оперирующих экземплярами таких структур. Каждая переменная *x*, указывающая на структуру, начинающуюся с макроса `PyObject_HEAD`, и, в частности, каждая переменная `PyObject*` *x*, имеет поле *x*`->ob_type`, которое является адресом структуры `PyTypeObject`, описывающей тип Python переменной *x*.

Определение структуры `PyTypeObject`

Для структуры экземпляра вида

```
typedef struct {
    PyObject_HEAD
    /* другие данные, необходимые экземпляру данного типа, опущены */
} mytype;
```

соответствующая структура PyTypeObject почти всегда начинается примерно так, как показано ниже.

```
static PyTypeObject t_mytype = {
/* tp_head */      PyObject_HEAD_INIT(NULL) /* используйте NULL
                                               для MSVC++ */
/* tp_internal */  0,                       /* должен быть 0 */
/* tp_name */      "mymodule.mytype", /* имя типа, включающее
                                         имя модуля */
/* tp_basicsize */ sizeof(mytype),
/* tp_itemsize */  0,     /* 0, кроме типов переменного размера */
/* tp_dealloc */   (destructor)mytype_dealloc,
/* tp_print */     0,     /* обычно 0, вместо него используйте str */
/* tp_getattr */   0,     /* обычно 0 (см. getattro) */
/* tp_setattr */   0,     /* обычно 0 (см. setattro) */
/* tp_compare*/    0,     /* см. также richcompare */
/* tp_repr */      (reprfunc)mytype_str,  /* как __repr__ в Python */
/* остальная часть структуры опущена */
```

Чтобы обеспечить переносимость в Microsoft Visual C++, аргумент макроса PyObject_HEAD_INIT в начале структуры PyTypeObject должен иметь значение NULL. В процессе инициализации модуля вы должны вызвать функцию PyType_Ready(&t_mytype), которая, среди всего прочего, вставляет в t_mytype адрес своего типа (типа типа, также называемый *метатипом*), обычно &PyType_Type. Другим полем в PyTypeObject, указывающим на другой тип объектов, является поле tp_base, появляющееся далее в структуре. В определении самой структуры значением tp_base должно быть NULL, опять-таки в интересах совместимости с Microsoft Visual C++. Однако, прежде чем вызвать функцию PyType_Ready(&t_mytype), вы можете установить в качестве значения t_mytype.tp_base адрес объекта другого типа. Если вы сделаете это, то ваш тип будет наследоваться от другого типа в полной аналогии с тем, как класс в Python может наследоваться от встроенного типа. Применительно к типу Python, запрограммированному на языке C, "наследование" означает, что в большинство полей структуры PyTypeObject, если вы установили в них значение NULL, функция PyType_Ready скопирует значения из соответствующих полей базового типа. Тип должен явно подтвердить в своем поле tp_flags, что его можно использовать в качестве базового, в противном случае ни один тип не сможет наследоваться от него.

Поле tp_itemsize представляет интерес только в случае типов, которые могут иметь экземпляры различного размера, как кортежи, и определять размер экземпляра раз и навсегда во время создания. Большинство типов просто устанавливает для поля tp_itemsize значение 0. Поля tp_getattr и tp_setattr обычно устанавливаются в NULL, поскольку они нужны лишь для обеспечения обратной совместимости. Современные типы используют вместо них поля tp_getattro и tp_setattro. Большинство оставшихся полей содержит адреса функций, непосредственно соответствующих специальным методам Python. Их типичным представителем может

служить поле tp_repr, в котором хранится адрес функции, соответствующей специальному методу __repr__ в Python. Вы можете установить для такого поля значение NULL, указывающее на то, что ваш тип не предоставляет данный специальный метод, или задать в нем указатель на функцию, обеспечивающую требуемую функциональность. Если вы установите это поле в NULL, но при этом также укажете базовый тип с помощью поля tp_base, то унаследуете специальный метод, если таковой в нем имеется, от базового типа. Вам часто потребуется приводить свои функции к конкретному типу typedef, требуемому для поля (в данном случае тип reprfunc для поля tp_repr), поскольку первым аргументом typedef является PyObject* self, в то время как ваша функция, специфическая для вашего типа, обычно использует более специализированные указатели.

```
static PyObject* mytype_str(mytype* self) { ...
/* остальная часть опущена */
```

Другой возможный вариант заключается в том, чтобы объявить функцию mytype_str с аргументом PyObject* self, а затем использовать приведение (mytype*)self в теле функции. Допустимы оба варианта, однако чаще всего приведения помещают в объявление PyTypeObject.

Инициализация и финализация экземпляра

Задача финализации экземпляров распределяется между двумя функциями. Поле tp_dealloc никогда не должно иметь значение NULL, за исключением бессрочно существующих типов (т.е. типов, не требующих освобождения памяти, занимаемой их экземплярами). Python вызывает функцию *x*->ob_type->tp_dealloc(*x*) для каждого экземпляра *x*, счетчик ссылок которого уменьшается до 0, и вызванная таким образом функция должна освобождать любой ресурс, удерживаемый объектом *x*, включая занимаемую объектом *x* память. Если экземпляр mytype не удерживает никаких других ресурсов, которые должны быть освобождены (и в частности, не является владельцем ссылок на другие объекты Python, счетчик ссылок которых вы должны были бы уменьшить с помощью макроса DECREF), то деструктор экземпляра mytype может выглядеть чрезвычайно просто.

```
static void mytype_dealloc(PyObject *x)
{
    x->ob_type->tp_free((PyObject*)x);
}
```

Функция, указанная в поле tp_free, решает частную задачу освобождения памяти, занимаемой объектом *x*. Часто для этого достаточно указать в поле tp_free адрес функции C API _PyObject_Del.

Задача инициализации ваших экземпляров распределяется между тремя функциями. Чтобы выделить память для новых экземпляров вашего типа, укажите в поле tp_alloc функцию C API PyType_GenericAlloc, которая выполняет абсолютно минимальные действия по инициализации, заполняя нулями вновь выделяемые байты

памяти, за исключением указателя типа и счетчика ссылок. Аналогичным образом в поле tp_new часто можно задать функцию C API PyType_GenericNew. В этом случае вы можете выполнить всю инициализацию экземпляра в функции, указанной в поле tp_init, которая имеет следующую сигнатуру:

```
int init_name(PyObject *self, PyObject *args, PyObject *kwds)
```

Позиционными и именованными аргументами функции, указанной в поле tp_init, являются аргументы, переданные при вызове типа для создания нового экземпляра, точно так же, как в Python позиционными и именованными аргументами метода __init__ являются аргументы, переданные при вызове класса. Опять-таки, как и для типов (классов), определяемых в Python, общее правило заключается в том, чтобы выполнить как можно меньший объем операций по инициализации экземпляра в функции tp_new и как можно больший — в функции tp_init. Использование функции PyType_GenericNew для поля tp_new решает эту задачу. Однако вы можете определить в поле tp_new собственную функцию для специальных типов, например типов, имеющих неизменяемые экземпляры, инициализация которых должна выполняться раньше. Такие функции имеют следующую сигнатуру:

```
PyObject* new_name(PyObject *subtype, PyObject *args,
                   PyObject *kwds)
```

Функция, указанная в поле tp_new, возвращает вновь созданный экземпляр, обычно экземпляр подтипа (который может быть вашим подтипом). С другой стороны, функция, указанная в поле tp_init, должна возвращать 0 в случае успешного выполнения или -1 для индикации исключения.

Если ваш тип допускает создание подклассов, то очень важно, чтобы любые инварианты экземпляра устанавливались до возврата из функции, указанной в поле tp_new. Например, если необходимо гарантировать, что определенное поле экземпляра никогда не будет иметь значение NULL, то значение, не равное NULL, должно быть установлено для этого поля в функции tp_new. Вызов вашей функции tp_init подтипами вашего типа может оказаться невозможным. Поэтому для типов, допускающих создание подклассов, обязательные операции по инициализации, необходимые для установки значений инвариантов типа, всегда должны выполняться в функции tp_new.

Доступ к атрибутам

Доступ к атрибутам ваших экземпляров, включая методы (см. раздел "Основные сведения о ссылках на атрибуты" в главе 4), осуществляется через функции, указанные в полях tp_getattro и tp_setattro вашей структуры PyTypeObject. Обычно вы будете использовать функции C API PyObject_GenericGetAttr и PyObject_GenericSetAttr, реализующие стандартную семантику. В частности, эти функции получают доступ к методам вашего типа посредством поля tp_methods, указывающего на заканчивающийся сигнальной меткой массив структур PyMethodDef, и к

элементам ваших экземпляров посредством поля tp_members, указывающего на заканчивающийся сигнальной меткой массив структур PyMemberDef.

```
typedef struct {
    char* name;   /* имя элемента, видимое для Python */
    int type;     /* код, определяющий тип данных элемента */
    int offset;   /* смещение элемента в структуре экземпляра */
    int flags;    /* READONLY для члена, доступного только
                     для чтения */
    char* doc;    /* строка документирования для элемента */
} PyMemberDef;
```

В виде исключения из общего правила, в соответствии с которым включение заголовочного файла *Python.h* доставляет вам все необходимые объявления, вы должны явно включить заголовочный файл *structmember.h*, чтобы ваш код на языке C видел объявление структуры PyMemberDef.

В общем случае type — это тип T_OBJECT для элементов, являющихся объектами PyObject*, но в файле *Include/structmember.h* определены многие другие коды типов для элементов, которые ваши экземпляры хранят как собственные типы данных C (например, T_DOUBLE для чисел двойной точности или T_STRING для типа char*). Пусть, например, ваша структура экземпляра имеет следующий вид.

```
typedef struct {
    PyObject_HEAD
    double datum;
    char* name;
} mytype;
```

Предоставьте Python возможность доступа к атрибутам экземпляра datum (чтение/запись) и name (только чтение), определив следующий массив и указав его в поле tp_members вашей структуры PyTypeObject.

```
static PyMemberDef[] mytype_members = {
    {"datum", T_DOUBLE, offsetof(mytype, datum), 0,
     "Current datum"},
    {"name", T_STRING, offsetof(mytype, name), READONLY,
     "Datum name"}, {NULL}
};
```

Кроме того, использование функций PyObject_GenericGetAttr и PyObject_GenericSetAttr для полей tp_getattro и tp_setattro предоставляет дополнительные возможности, не рассматриваемые подробно в данной книге. Поле tp_getset указывает на заканчивающийся сигнальной меткой массив PyGetSetDefstructs, эквивалентный свойствам экземпляра в классе на языке Python. Если значение поля tp_dictoffset вашей структуры PyTypeObjectis не равно 0, то в нем должно быть указано смещение переменной PyObject*, указывающей на словарь Python, в пределах структуры экземпляра. В этом случае обобщенные API-функции для доступа

к атрибутам используют этот словарь для того, чтобы позволить коду на языке Python устанавливать значения произвольных атрибутов экземпляров вашего типа, точно так же, как и для экземпляров классов, код которых написан на языке Python.

Еще один словарь определен на уровне типа, а не на уровне экземпляра: указатель `PyObject*` для него задается в поле `tp_dict` вашей структуры `PyTypeObject`. Вы можете задать для поля `tp_dict` значение `NULL`, и тогда функция `PyType_Ready` инициализирует словарь соответствующим образом. Вы также можете действовать иначе, задав в поле `tp_dict` словарь атрибутов типа, а затем добавить с помощью функции `PyType_Ready` другие записи в тот же словарь в дополнение к установленным вами атрибутам типа. Как правило, проще начать с установки для поля `tp_dict` значения `NULL`, вызвать функцию `PyType_Ready` для создания и инициализации словаря на уровне типа, а затем, если потребуется, явным образом добавить дополнительные записи в словарь с помощью кода на языке C.

Поле `tp_flags` имеет тип `long`, и его биты определяют точную компоновку структуры вашего типа, главным образом в интересах обратной совместимости. Установите для этого поля значение `Py_TPFLAGS_DEFAULT`, если хотите указать, что определяете нормальный, современный тип. Установите для поля `tp_flags` значение `Py_TPFLAGS_DEFAULT|Py_TPFLAGS_HAVE_GC`, если ваш тип поддерживает сбор циклического мусора. Ваш код должен поддерживать сбор циклического мусора, если экземпляры типа содержат поля `PyObject*`, которые могут указывать на произвольные объекты и быть частью циклических ссылок. Для поддержки сборки циклического мусора недостаточно добавить флаг `Py_TPFLAGS_HAVE_GC` в поле `tp_flags`. Вы дополнительно должны предоставить соответствующие функции, указав их в полях `tp_traverse` и `tp_clear`, а также соответствующим образом организовать регистрацию и отмену регистрации ваших экземпляров с помощью сборщика циклического мусора. Поддержка сбора циклического мусора относится к категории тем повышенной сложности и далее не рассматривается. Для получения дополнительной информации по этой теме обратитесь к онлайн-документации (`https://docs.python.org/3/c-api/gcsupport.html`). По той же причине в книге не рассматриваются и слабые ссылки. В онлайн-документации вы найдете их полное описание (`https://docs.python.org/3/extending/newtypes.html#weakref-support`).

Поле `tp_doc` типа `char*` содержит символьную строку с завершающим нулевым символом, используемую в качестве строки документирования вашего типа. Другие поля указывают на структуры (поля которых указывают на функции). Для каждого из этих полей вы можете установить значение `NULL`, указывающее на то, что вы не поддерживаете ни одну из этих функций. Для указания этих блоков функций используются следующие поля: `tp_as_number` — специальные методы, обычно предоставляемые числовыми типами; `tp_as_sequence` — специальные методы, обычно предоставляемые последовательностями; `tp_as_mapping` — специальные методы, обычно предоставляемые отображениями; и `tp_as_buffer` — специальные методы протокола буферов.

Например, объекты, не являющиеся последовательностями, все еще могут поддерживать один или несколько методов, входящих в блок, на который указывает поле tp_as_sequence, и в этом случае поле tp_as_sequence структуры PyTypeObject не должно иметь значение NULL, даже если блок указателей на функции, на который оно указывает, в основном заполнен значениями NULL. Например, словари предоставляют специальный метод __contains__, позволяющий проверить, принадлежит ли *x* объекту *d*, если *d* — словарь. На уровне C-кода этот метод является функцией, на которую указывает поле sq_contains, являющееся частью структуры PySequenceMethods, на которую указывает поле tp_as_sequence. Поэтому значение поля tp_as_sequence структуры PyTypeObject для типа dict, называемой PyDict_Type, не равно NULL, несмотря на то что словарь не предоставляет никаких других полей в структуре PySequenceMethods, кроме поля sq_contains, и поэтому все остальные поля в структуре *(PyDict_Type.tp_as_sequence) содержат значения NULL.

Пример определения типа

В листинге 24.2 представлен завершенный модуль расширения Python, определяющий очень простой тип intpair, каждый экземпляр которого хранит два целых числа с именами first и second.

Листинг 24.2. Определение нового типа intpair

```
#include "Python.h"
#include "structmember.h"

/* Структура данных для каждого экземпляра */
typedef struct {
    PyObject_HEAD
    int first, second;
} intpair;

static int
intpair_init(PyObject *self, PyObject *args, PyObject *kwds)
{
    static char* nams[] = {"first","second",NULL};
    int first, second;
    if(!PyArg_ParseTupleAndKeywords(args, kwds, "ii", nams,
                                    &first, &second))
        return -1;
    ((intpair*)self)->first = first;
    ((intpair*)self)->second = second;
    return 0;
}

static void
intpair_dealloc(PyObject *self)
```

```
{
    self->ob_type->tp_free(self);
}

static PyObject*
intpair_str(PyObject* self)
{
    return PyString_FromFormat("intpair(%d,%d)",
        ((intpair*)self)->first, ((intpair*)self)->second);
}

static PyMemberDef intpair_members[] = {
    {"first", T_INT, offsetof(intpair, first), 0, "first item" },
    {"second", T_INT, offsetof(intpair, second), 0, "second item" },
    {NULL}
};

static PyTypeObject t_intpair = {
    PyObject_HEAD_INIT(0)      /* tp_head */
    0,                         /* tp_internal */
    "intpair.intpair",         /* tp_name */
    sizeof(intpair),           /* tp_basicsize */
    0,                         /* tp_itemsize */
    intpair_dealloc,           /* tp_dealloc */
    0,                         /* tp_print */
    0,                         /* tp_getattr */
    0,                         /* tp_setattr */
    0,                         /* tp_compare */
    intpair_str,               /* tp_repr */
    0,                         /* tp_as_number */
    0,                         /* tp_as_sequence */
    0,                         /* tp_as_mapping */
    0,                         /* tp_hash */
    0,                         /* tp_call */
    0,                         /* tp_str */
    PyObject_GenericGetAttr,   /* tp_getattro */
    PyObject_GenericSetAttr,   /* tp_setattro */
    0,                         /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT,
    "two ints (first,second)",
    0,                         /* tp_traverse */
    0,                         /* tp_clear */
    0,                         /* tp_richcompare */
    0,                         /* tp_weaklistoffset */
    0,                         /* tp_iter */
    0,                         /* tp_iternext */
    0,                         /* tp_methods */
    intpair_members,           /* tp_members */
```

```
    0,                          /* tp_getset */
    0,                          /* tp_base */
    0,                          /* tp_dict */
    0,                          /* tp_descr_get */
    0,                          /* tp_descr_set */
    0,                          /* tp_dictoffset */
    intpair_init,               /* tp_init */
    PyType_GenericAlloc,        /* tp_alloc */
    PyType_GenericNew,          /* tp_new */
    _PyObject_Del,              /* tp_free */
};

static PyMethodDef no_methods[] = { {NULL} };

static char intpair_docs[] =
    "intpair: data type with int members .first, .second\n";

static struct PyModuleDef intpair_module = {
    PyModuleDef_HEAD_INIT,
    "intpair",
    intpair_docs,
    -1,
    no_methods
};

PyMODINIT_FUNC
PyInit_intpair(void)
{
    PyObject* this_module = PyModule_Create(&intpair_module);
    PyType_Ready(&t_intpair);
    PyObject_SetAttrString(this_module, "intpair",
                           (PyObject*)&t_intpair);
    return this_module;
}
```

Тип `intpair`, определенный в листинге 24.2, не предоставляет никаких ощутимых преимуществ по сравнению, например, со следующим определением его эквивалента в Python.

```
class intpair(object):
    __slots__ = 'first', 'second'
    def __init__(self, first, second):
        self.first = first
        self.second = second
    def __repr__(self):
        return 'intpair(%s,%s)' % (self.first, self.second)
```

Однако версия, запрограммированная на языке C, гарантирует, что оба атрибута являются целыми числами, усекая аргументы в виде чисел с плавающей точкой или комплексных чисел в случае необходимости.

```
import intpair
x=intpair.intpair(1.2,3.4)                    # x: intpair(1,3)
```

(В Python вы могли бы аппроксимировать эту функциональность, пропуская аргументы через функцию `int`, но это все еще было бы не то же самое, поскольку Python допускал бы в подобном случае также строковые значения аргументов, такие как строка `'23'`, в то время как версия на языке C не пропустит такие значения.)

Версия `intpair`, написанная на языке C, занимает немного меньше памяти, чем версия на языке Python. Однако листинг 24.2 преследует исключительно учебные цели: представить версию расширения Python на языке C, которая определяет простой новый тип.

Расширение Python без использования C API

Для написания расширений Python можно использовать другие классические компилируемые языки программирования, а не только C. Если вы остановили свой выбор на языке Fortran, рекомендуем воспользоваться инструментом F2PY (`https://pypi.python.org/pypi/F2PY/2.45.241_1926`). Теперь он является частью пакета NumPy (`www.numpy.org`), поскольку Fortran часто используется для численных расчетов. Если, как это рекомендуется, вы установили модуль `numpy`, то в отдельной установке модуля `f2py` нет необходимости.

Разумеется, для C++ можно использовать тот же подход, что и для C (добавляя в необходимых случаях описание `extern "C"`). Существует много альтернатив, специфических для C++, но, по-видимому, из всех подобных инструментов активно поддерживается и сопровождается только SIP (`https://riverbankcomputing.com/software/sip/intro`).

Популярной альтернативой является Cython (`cython.org`) — диалект Python, генерирующий C-код на основе синтаксиса, близкого к синтаксису Python, с некоторыми добавлениями, которые ориентированы на снижение накладных расходов, связанных с управляющими структурами и вызовами подпрограмм. Мы настоятельно рекомендуем использовать этот инструмент, рассмотренный в разделе "Cython".

К числу других альтернативных средств относится инструмент CFFI (C Foreign Function Interface — интерфейс доступа к внешним функциям C; `https://pypi.python.org/pypi/cffi`), первоначально разработанный в рамках проекта PyPy и идеально подходящий для него, но полностью поддерживающий CPython. Режим работы "ABI online" инструмента CFFI соответствует модулю `ctypes` стандартной библиотеки Python, кратко охарактеризованному в следующем разделе, однако инструмент CFFI обладает рядом дополнительных преимуществ и, в частности, возможностью генерировать и компилировать код на языке C (режим "offline" согласно

терминологии CFFI), а также предоставлением интерфейса на более стабильном уровне API, чем ABI.

Модуль ctypes

Модуль `ctypes` стандартной библиотеки Python позволяет загружать существующие динамические библиотеки, написанные на языке C, и вызывать в коде на языке Python функции, определенные в этих библиотеках. Данный модуль пользуется популярностью, поскольку позволяет писать соответствующий программный код на языке Python, что избавляет от необходимости использовать компилятор C или сторонние расширения Python. Однако результирующая программа может оказаться довольно хрупкой, платформозависимой, а ее портирование в другие версии тех же библиотек может быть сопряжено с определенными трудностями, поскольку она зависит от деталей реализации их двоичных интерфейсов.

Мы рекомендуем избегать использования инструментария `ctypes` в производственном коде и использовать вместо него одну из превосходных альтернатив, рассмотренных или упомянутых в этой главе, таких, в частности, как CFFI или, что еще лучше, Cython.

Cython

Язык Cython (`cython.org`), представляющий собой дальнейшее развитие системы Pyrex (`http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`), часто оказывается наиболее удобным средством для написания расширений Python. Cython — почти полное подмножество Python (которое, в соответствии с заявленными намерениями, в конечном счете должно стать его полным супермножеством[1], включающим дополнительные типы переменных в стиле C). Вы можете автоматически компилировать программы на языке Cython (исходные файлы с расширением *.pyx*) в машинный код (посредством промежуточной стадии генерирования C-кода), получая на выходе расширения, которые можно импортировать в Python. На сайте `cython.org` вы найдете прекрасно подготовленную документацию, содержащую подробное описание языка программирования Cython. В данном разделе мы приведем лишь некоторые минимальные сведения, которые помогут вам приступить к работе с Cython.

Cython реализует почти всю функциональность Python, если не считать четырех несущественных ограничений (`http://docs.cython.org/en/latest/src/userguide/limitations.html`), которые авторы Cython не намерены устранять, и нескольких незначительных отличий, рассматриваемых как ошибки в текущей версии Cython, исправление которых в ближайшем будущем запланировано авторами (перечислять здесь эти ошибки не имеет смысла, поскольку вполне вероятно, что к тому времени, когда вы будете читать эти строки, указанные ошибки уже будут исправлены).

[1] По умолчанию в основном поддерживается версия v2, но вы можете запустить Cython с переключателем `-3` или включить директиву `# cython: language_level=3` в начале исходного кода, чтобы обеспечить поддержку версии v3 (строки — Unicode, `print` — функция и т.п.).

Итак, Cython — широкое подмножество Python. Что более важно, Cython добавляет в Python ряд инструкций, которые позволяют использовать объявления в стиле C, обеспечивающие возможность генерирования эффективного машинного кода (через стадию генерирования C-кода). Ниже приведен простой пример. Сохраните этот код в файле *hello.pyx* в новом пустом каталоге.

```
def hello(char *name):
    return 'Hello, ' + name + '!'
```

Этот код почти в точности является кодом на языке Python, за исключением того, что имени параметра предшествует объявление типа `char*`, указывающее на то, что данный параметр всегда должен быть строкой C с завершающим нулевым символом (но, как вы можете убедиться, в Cython всегда можно использовать этот тип в качестве обычной строки Python).

Установив Cython (посредством его загрузки и последующего выполнения команды **`python setup.py install`** или выполнения команды **`pip install cython`**), вы сможете осуществлять сборку файлов с исходным кодом на языке Cython в динамические библиотеки расширений, используя подход, основанный на использовании пакета `distutils`. Сохраните в новом каталоге файл setup.*py*, имеющий следующее содержимое.

```
from setuptools import setup
from Cython.Build import cythonize

setup(name='hello',ext_modules=cythonize('hello.pyx'))
```

После этого выполните команду **`python setup.py install`** в новом каталоге. (Игнорируйте предупреждения, выводимые в процессе компиляции кода. Они вполне ожидаемы и ни к чему не обязывают.) Теперь вы можете импортировать и использовать новый модуль, в том числе в интерактивном сеансе Python.

```
>>> import hello
>>> hello.hello('Alex')
'Hello, Alex!'
```

С учетом того, как был написан данный код на языке Cython, мы должны передать функции `hello.hello` в качестве аргумента одну строку. Если вызвать эту функцию без аргументов или же передать ей большее количество аргументов или аргумент, не являющийся строкой, то будет возбуждено исключение.

```
>>> hello.hello()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function takes exactly 1 argument (0 given)
>>> hello.hello(23)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: argument 1 must be string, not int
```

Инструкции cdef и cpdef и параметры функции

Вы можете использовать ключевое слово cdef в основном так же, как использовали бы ключевое слово def, но cdef определяет функции, которые являются внутренними функциями модуля расширения, невидимыми снаружи, в то время как def-функции могут вызываться также кодом Python, который импортирует данный модуль. Функции, определяемые с использованием ключевого слова cpdef, могут вызываться как из модуля (со скоростью, близкой к скорости вызова cdef-функций), так и внешним кодом на языке Python (точно так же, как def-функции), но во всем остальном они идентичны cdef-функциям.

В функциях любого рода параметры и возвращаемые значения, для которых тип не указан или, что еще лучше, указан тип object, становятся указателями PyObject* в сгенерированном C-коде (с неявной стандартной обработкой увеличения и уменьшения счетчиков ссылок). Функции, определенные с использованием ключевого слова cdef, также могут иметь параметры и возвращаемые значения любого другого типа C. Функции, определенные с использованием ключевого слова def, в дополнение к не типизированным (или, что эквивалентно, типизированным как object) аргументам могут принимать только аргументы, имеющие тип int, float или char*. Ниже приведен пример cdef-функции, предназначенной для суммирования двух целых чисел.

```
cdef int sum2i(int a, int b):
    return a + b
```

Кроме того, ключевое слово cdef можно использовать для объявления переменных C — скаляров, массивов и указателей, как в языке C:

```
cdef int x, y[23], *z
```

а также переменных, относящихся к типам struct, union и enum, как в Python (двоеточие в заголовке и выделение отступом последующих строк):

```
cdef struct Ure:
    int x, y
    float z
```

(Впоследствии вы должны обращаться к новому типу только по имени, например Ure. Никогда не используйте ключевые слова struct, union и enum, кроме как в определении типа с помощью ключевого слова cdef.)

Внешние объявления

Для организации интерфейса с внешним кодом на языке C вы можете объявлять переменные как cdef extern с тем же результатом, который обеспечивает спецификатор extern в языке C. Однако чаще всего объявления библиотеки на языке C, которую вы хотите использовать, лучше помещать в заголовочный файл C с расширением .h. Для гарантии того, что сгенерированный с помощью Cython код на языке C будет включать этот заголовочный файл, используйте следующее объявление

cdef и поместите вслед за ним блок объявлений в стиле cdef (не повторяя ключевое слово cdef в блоке), выделенный отступами:

```
cdef extern from "someheader.h":
```

Объявляйте только функции и переменные, которые хотите использовать в своем Cython-коде. Cython не читает заголовочный файл C — он доверяет вашим Cython-декларациям в блоке, не генерируя для них никакого кода на языке C. Cython неявно использует Python/C API, рассмотренный в начале этой главы, но вы можете явно обращаться к любой из его функций. Например, если ваш файл Cython содержит приведенный ниже код, то последующий код Cython может использовать функцию PyString_FromStringAndSize.

```
cdef extern from "Python.h":
    object PyString_FromStringAndSize(char *, int)
```

Это оказывается очень кстати, поскольку по умолчанию считается, что "строки" C заканчиваются нулевым символом, но с помощью этой функции можно явно указывать длину строки C, а также получать любой нулевой символ (символы), который она может содержать.

Удобно то, что Cython позволяет группировать подобные объявления в *.pxd*- файлах (примерно аналогичных *.h*-файлам C, в то время как *.pyx*-файлы Cython примерно аналогичны *.c*-файлам C). Файлы с расширением *.pxd* также могут включать встроенные объявления cdef inline, которые будут встраиваться в код во время компиляции. Файл с расширением *.pyx* может импортировать объявления, содержащиеся в *.pxd*-файле, используя ключевое слово cimport, аналогичное ключевому слову import в Python.

Кроме того, Cython поставляется вместе с несколькими готовыми *.pxd*-файлами в своем каталоге *Cython/includes*. В частности, *.pxd*-файл *cpython* уже содержит все объявления cdef extern from "Python.h", которые могут вам понадобиться: для доступа к ним достаточно выполнить команду cimport cpython.

cdef-классы

Инструкция cdef class позволяет определить новый тип Python в Cython. Этот тип может включать cdef-объявления атрибутов (применяемые к каждому экземпляру, а не к типу в целом), которые обычно остаются невидимыми для кода на языке Python. Однако вы можете объявить конкретные атрибуты как cdef public, чтобы сделать их обычными атрибутами с точки зрения Python, или как cdef readonly, чтобы они были видимыми, но доступными только для чтения из Python (видимые из Python атрибуты должны быть числами, строками или объектами).

Класс, определенный с помощью инструкции cdef class (для краткости — cdef-класс), поддерживает специальные методы (с некоторыми оговорками), свойства (подчиняющиеся специальному синтаксису) и наследование (только одиночное). Для объявления свойства в теле cdef-класса используйте следующую инструкцию:

```
property name:
```

Вслед за ней записываются с отступом инструкции `def` для метода `__get__(self)`, а также для необязательных методов `__set__(self, value)` и `__del__(self)`.

Метод `__new__` `cdef`-класса отличается от одноименного метода обычного класса Python: его первым аргументом является `self` — новый экземпляр, для которого память уже выделена и заполнена нулями. Cython всегда вызывает специальный метод `__cinit__(self)` сразу же после размещения экземпляра в памяти, чтобы сделать возможной его последующую инициализацию. Метод `__init__`, если он определен, вызывается следующим. Во время уничтожения объекта Cython вызывает специальный метод `__dealloc__(self)`, позволяющий освободить память, выделенную ранее с помощью методов `__new__` и/или `__cinit__` (`cdef`-классы не имеют специального метода `__del__`).

Правосторонних версий специальных арифметических методов, таких как метод `__radd__`, идущий в паре с методом `__add__` в Python, не существует. Вместо этого, если, скажем, выражение `a+b` не может найти или использовать метод `type(a).__add__`, то далее вызывается метод `type(b).__add__(a, b)`, — обратите внимание на порядок следования аргументов (никакого обмена местами!). Чтобы убедиться в корректности выполнения операций во всех случаях, вам может потребоваться проверка типов.

Чтобы превратить экземпляры класса, объявленного с помощью инструкции `cdef class`, в итераторы, определите специальный метод `__next__(self)`, как в версии v3 (даже если используете Cython с версией v2).

Ниже приведен эквивалент примера 24.2 на языке Cython.

```
cdef class intpair:
    cdef public int first, second
    def __init__(self, first, second):
        self.first = first
        self.second = second
    def __repr__(self):
        return 'intpair(%s,%s)' % (self.first, self.second)
```

Как и C-расширение из примера 24.2, данное расширение на языке Cython также не обеспечивает никаких преимуществ по сравнению с их эквивалентом на языке Python. Однако простота и лаконичность кода на Cython делают его гораздо более близким к коду на Python, чем соответствующий код на языке C, требующий большего объема ввода и использования шаблонных блоков кода, но при этом машинный код, сгенерированный из этого файла Cython, очень близок к машинному коду, сгенерированному на основе C-кода из примера 24.2.

Инструкция `ctypedef`

Разрешается использовать ключевое слово `ctypedef` для объявления имени (синонима) типа:

```
ctypedef char* string
```

Инструкция for... from

Кроме обычных инструкций for из Python, Cython предлагает другую форму for:

for *variable* from *lower_expression*<=*variable*<*upper_expression*:

Это наиболее общая форма, но вы можете использовать любой из операторов < или <= по обе стороны от переменной variable после ключевого слова from. Альтернативный вариант заключается в том, чтобы использовать оператор > и/или >= для выполнения цикла в обратном направлении (смешивание оператора < или <= с одной стороны и оператора > или >= с другой не допускается).

Инструкция for...from работает намного быстрее, чем обычная инструкция Python for *variable* in range(...):, если переменная *variable* и границы цикла являются целыми числами C. Однако в современном Cython инструкция for *variable* in range(...): оптимизирована почти до уровня инструкции for... from, поэтому обычному стилю for *variable* in range(...): в духе классического Python можно отдать предпочтение в силу его простоты и удобочитаемости.

Выражения Cython

Кроме предусмотренного в Python синтаксиса выражений, Cython может использовать некоторые, но не все возможности C в этом отношении. Чтобы принять адрес переменной *var*, используйте выражение &*var*, как в C. Однако для того, чтобы разыменовать указатель *p*, используйте выражение p[0]. Эквивалентный синтаксис C **p* в Cython недопустим. Так, где в C обычно используется выражение *p->q*, в Cython используйте выражение *p.q*. Для нулевого указателя в Cython используется ключевое слово NULL. Для символьных констант используйте синтаксис c'x'. Для приведений типов используйте угловые скобки, например <int>*somefloat* вместо используемого в C эквивалента (int)*somefloat*. Кроме того, используйте приведения только в отношении значений и типов C, но не значений и типов Python (если речь идет о значениях и типах Python, позвольте Cython выполнять автоматическое преобразование типов).

Пример на языке Cython: наибольший общий делитель

Алгоритм Евклида для нахождения наибольшего общего делителя двух чисел несложно реализовать исключительно на языке Python.

```
def gcd(dividend, divisor):
    remainder = dividend % divisor
    while remainder:
        dividend = divisor
        divisor = remainder
        remainder = dividend % divisor
     divisor
```

Его версия на языке Cython очень похожа на предыдущую.

```
def gcd(int dividend, int divisor):
    cdef int remainder
    remainder = dividend % divisor
    while remainder:
        dividend = divisor
        divisor = remainder
        remainder = dividend % divisor
    return divisor
```

Выполнение вызова `gcd(454803,278255)` на ноутбуке Macbook Air длится примерно 1 мкс для версии на языке Python, тогда как версия на языке Cython выполняется в течение примерно 0,22 мкс. Ускорение примерно в четыре раза ценой столь малых усилий стоило затраченных трудов (разумеется, в предположении, что на эту функцию приходится значительная доля всего времени выполнения программы!), даже несмотря на то, что версия на языке Python обладает рядом практических преимуществ (она может выполняться в Jython, IronPython или PyPy так же, как и в CPython; она работает с типами `long` и `int`, безболезненно переносится на другую платформу и т.п.).

Внедрение Python

Если у вас имеется готовое приложение, написанное на C или C++ (или на любом другом классическом компилируемом языке программирования), то может возникнуть желание использовать встроенный Python в качестве языка сценариев для вашего приложения. Чтобы можно было встроить Python в другой язык, отличный от C, этот язык должен допускать возможность вызова функций на языке C (конкретный способ того, как это делается, зависит не только от самого языка, но и от его конкретной реализации: компилятора, компоновщика и т.п). В последующем изложении мы ориентируемся на язык C. Процедура работы с другими языками, как уже отмечалось, будет меняться в зависимости от способа вызова из них функций на языке C.

Установка резидентных модулей расширения

Чтобы сценарии на языке Python могли обмениваться данными с вашим приложением, оно должно предоставить модули расширения с доступными для Python функциями и классами, реализующими функциональность вашего приложения. Если, как это обычно бывает, указанные модули компонуются с вашим приложением (а не находятся в динамических библиотеках, которые Python может загружать по мере необходимости), зарегистрируйте их в Python в качестве дополнительных встроенных модулей посредством вызова функции `PyImport_AppendInittab` из C API.

PyImport_AppendInittab `int PyImport_AppendInittab(char* name, void (*initfunc)(void))`

Аргумент *name* — это имя модуля, которое сценарий на языке Python использует для его импорта. Аргумент *initfunc* — это функция инициализации модуля, которая не принимает аргументов и не возвращает результат, о чем говорилось в разделе "Модуль инициализации" (т.е. *initfunc* — это функция модуля, которая в случае обычного модуля расширения в динамической библиотеке в версии v2 называлась бы init*name*). Вызывайте функцию `PyImport_AppendInittab` перед вызовом функции `Py_Initialize`.

Задание аргументов

Вы можете задать имя программы и аргументы так, чтобы сценарии на языке Python могли получать к ним доступ через переменную `sys.argv`, вызвав одну или обе функции C API, которые описаны ниже.

Py_SetProgramName `void Py_SetProgramName(char* name)`

Задает имя программы, к которому сценарии на языке Python могут обращаться как к `sys.argv[0]`. Должна вызываться *до* вызова функции `Py_Initialize`.

PySys_SetArgv `void PySys_SetArgv(int argc, char** argv)`

Задает *argc* аргументов программы в массиве *argv* в виде строк с завершающим нулевым символом, к которым сценарии на языке Python могут обращаться как к `sys.argv[1:]`. Должна вызываться *после* вызова функции `Py_Initialize`.

Инициализация и финализация Python

После установки дополнительных встроенных модулей и необязательного задания имени программы ваше приложение инициализирует Python. Когда необходимость в использовании Python отпадает, приложение финализирует Python. Соответствующие функции C API описаны ниже.

Py_Finalize `void Py_Finalize(void)`

Освобождает всю память и другие ресурсы, которые способен освободить Python. После вызова этой функции не следует вызывать никакие другие функции C API.

Py_Initialize `void Py_Initialize(void)`

Инициализирует среду Python. До вызова этой функции не следует вызывать никакие другие функции C API, за исключением функций `PyImport_AppendInittab` и `Py_SetProgramName`.

Выполнение кода Python

Ваше приложение может выполнить исходный код на языке Python из символьной строки или из файла. Для компиляции или выполнения исходного кода на языке Python выберите режим выполнения, который устанавливается с помощью одной из следующих трех констант, определенных в заголовочном файле *Python.h*.

`Py_eval_input`
: Кодом является вычисляемое выражение (аналогично передаче `'eval'` встроенной функции `compile` в Python).

`Py_file_input`
: Кодом является блок, который состоит из одной или нескольких инструкций, подлежащих выполнению. (Аналогично передаче `'exec'` функции `compile`. Точно так же, как и в этом случае, составные инструкции должны заканчиваться замыкающим символом `'\n'`.)

`Py_single_input`
: Кодом является одиночная инструкция, предназначенная для интерактивного выполнения (аналогично передаче `'single'` функции `compile`; неявно выводит результаты инструкций-выражений).

Выполнение исходного кода на языке Python аналогично передаче строки исходного кода функции `exec` или `eval` Python или, в версии v2, передаче файла с исходным кодом встроенной функции `execfile`. Для этого используются две общие функции, описание которых приведено ниже.

`PyRun_File` `PyObject* PyRun_File(FILE* fp, char* filename, int start, PyObject* globals, PyObject* locals)`

Аргумент *`fp`* — это файл исходного кода на языке Python, открытый для чтения. Аргумент *`filename`* — это имя файла, используемое в сообщениях об ошибке. Аргумент *`start`* — это одна из констант `Py_..._input`, которые определяют режим выполнения. Аргументы *`globals`* и *`locals`* — это словари (могут быть одним и тем же словарем, указанным дважды), используемые в качестве глобального и локального пространств имен для выполнения кода. Возвращает результат вычисления выражения, если аргумент *`start`* имеет значение `Py_eval_input`, в противном случае — новую ссылку на объект `Py_None` или значение `NULL` для индикации возбуждения исключения в процессе выполнения.

`PyRun_String` `PyObject* PyRun_String(char* astring, int start, PyObject* globals, PyObject* locals)`

Аналогична функции `PyRun_File`, но исходный код находится в строке *`astring`*, завершающейся нулевым символом.

Словарями *`locals`* и *`globals`* часто являются новые, пустые словари (которые удобно создавать с помощью функции `Py_BuildValue("{}")`) или словарь модуля.

Функция PyImport_Import обеспечивает удобный способ получения существующего объекта модуля. Функция PyModule_GetDict получает словарь модуля.

Если необходимо создать новый объект модуля на лету, нередко для его заполнения вызовами функции PyRun_, то для этого можно использовать функцию PyModule_New из C API, которая описана ниже.

PyModule_New `PyObject* PyModule_New(char& name)`

Возвращает новый, пустой объект модуля для модуля *name*. Прежде чем можно будет использовать новый объект, вы должны добавить в него строку атрибута с именем `__file__`.

```
PyObject* newmod = PyModule_New("mymodule");
PyModule_AddStringConstant(newmod,"__file__","<synth>");
```

После выполнения этого кода объект модуля *newmod* готов к использованию. Вы можете получить словарь модуля с помощью вызова `PyModule_GetDict(newmod)` и передать его такой функции, как `PyRun_String`, в качестве аргумента *globals* и, возможно, аргумента *locals*.

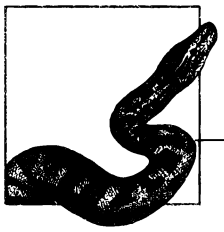
Для многократного выполнения кода на языке Python, а также для отделения диагностики синтаксических ошибок от исключений времени выполнения, возбуждаемых выполняющимся кодом, можно скомпилировать исходный код в объект кода и многократно выполнять его. Это относится как к случаям использования C API, так и к случаям динамического выполнения в Python, о чем шла речь в разделе "Динамическое выполнение и инструкция exec" главы 13. Для этих задач можно использовать две функции C API, описание которых приведено ниже.

Py_CompileString `PyObject* Py_CompileString(char* code, char* filename, int start)`

Аргумент *code* — это строка исходного кода, завершающаяся нулевым символом. Аргумент *filename* — это имя файла, используемое в сообщениях об ошибке. Аргумент *start* — это одна из констант, определяющих режим выполнения. Возвращает объект кода Python, содержащий байт-код, или значение NULL для индикации синтаксических ошибок.

PyEval_EvalCode `PyObject* PyEval_EvalCode(PyObject* co, PyObject* globals, PyObject* locals)`

Аргумент *co* — это объект кода Python, возвращенный, например, функцией Py_CompileString. Аргументы *globals* и *locals* — это словари (могут быть одним и тем же словарем, указанным дважды), используемые в качестве глобального и локального пространств имен для выполнения кода. Возвращает результат вычисления выражения, если объект *co* компилировался с использованием режима выполнения Py_eval_input, в противном случае — новую ссылку на объект Py_None или значение NULL для индикации возбуждения исключения в процессе выполнения.



25

Распространение расширений и программ

Пакет `distutils` в Python является частью стандартной библиотеки, которая лежит в основе инструментов, используемых для пакетирования и распространения программ и расширений Python. Однако мы не рекомендуем непосредственно использовать этот пакет: вместо него следует применять более новые сторонние инструменты. В большинстве случаев вы будете отдавать предпочтение модулям `setuptools` и `wheels` для создания *wheel*-файлов, а затем использовать `twine` для выгрузки этих файлов в предпочитаемый вами репозиторий (обычно PyPI, также известный как CheeseShop; `https://wiki.python.org/moin/CheeseShop`) и последующей их загрузки и установки с помощью команды `pip`. Если вы не установили ранее утилиту `pip` (она поставляется с версиями v2 и v3), посетите веб-страницу с описанием процедуры установки `pip` (`https://pip.pypa.io/en/stable/installing/`). Проследите за тем, чтобы использовать самую последнюю версию `pip`, выполнив команду **`pip install --upgrade pip`**. На компьютере Windows выполните команду **`py -m pip install -U pip setuptools`**, а затем убедитесь в том, что путь к `pip` включен в вашу системную переменную среды `PATH`, аналогично тому, как проверяется включение в эту переменную пути к вашей установке Python (`https://docs.python.org/3/using/windows.html#finding-the-python-executable`).

В этой главе рассмотрены простейшие способы использования `setuptools` и `twine` для создания и распространения пакетов. Более подробную информацию по этой теме можно найти в руководстве пользователя по созданию пакетов в Python (`https://docs.python.org/3/using/windows.html#finding-the-python-executable`). Во время написания данной книги был выпущен новый документ, специфицирующий протоколы распространения расширений и программ с помощью других инструментов сборки, но эти средства в настоящее время еще не поддерживаются и потому нами не рассматриваются.

Если вы заинтересованы в создании и распространении более сложных кросс-платформенных или облачных приложений, то вам имеет смысл ознакомиться с контейнерами *Docker* (`www.docker.com`). Если вам нужен полнофункциональный менеджер пакетов, особенно если вы работаете в области науки о данных или занимаетесь инженерными разработками, то вас должен заинтересовать пакет `conda`, лежащий в основе программ установки *Miniconda* (`https://conda.io/miniconda.html`) и *Anaconda* (`https://docs.anaconda.com/`). Если захотите использовать сценарный установщик с возможностями управления виртуальными окружениями, обратите внимание на установщик *pipsi* (`https://github.com/mitsuhiko/pipsi/blob/master/README.rst`).

Пакет `setuptools`

Пакет `setuptools` предоставляет набор инструментов с богатыми и гибкими возможностями, предназначенный для пакетирования программ и расширений Python в целях распространения. Мы рекомендуем использовать именно этот пакет, отдавая ему предпочтение по сравнению с пакетом `distutils` стандартной библиотеки.

Дистрибутив и его корневой каталог

Дистрибутив — это набор файлов, упаковываемый в единый архивный файл в целях его распространения. Дистрибутив может включать пакеты Python и/или другие модули Python (см. главу 6), а также, по выбору, сценарии на языке Python, расширения на языке C (и других языках), файлы данных и вспомогательные файлы с метаданными. Дистрибутив называют *чистым*, если весь включенный в него код написан на языке Python, и *смешанным*, если он включает также код, написанный на других языках программирования (как правило, C-расширения). Дистрибутив называют *универсальным*, если он чистый и может использоваться как с версией v2, так и с версией v3 без использования транслятора `2to3`.

Обычно все файлы дистрибутива помещают в каталог, который называется *корневым каталогом дистрибутива*, и его подкаталоги. В большинстве случаев вы можете организовать поддерево файлов и каталогов, удовлетворяющее вашим потребностям, используя в качестве корневого каталога корневой каталог дистрибутива. Однако, о чем было сказано в разделе "Пакеты" главы 6, пакет Python должен располагаться в собственном каталоге (если только вы не создаете пакеты пространств имен, о которых шла речь в разделе "Пакеты пространств имен (только в версии v3)" главы 6), а каталог пакета должен содержать файл *__init__.py* (равно как и подкаталоги должны содержать файлы *__init__.py* для подпакетов, если таковые имеются), а также другие модули, которые принадлежат данному пакету.

Дистрибутив должен включать сценарий *setup.py*, а также файл *README* (предпочтительно в формате *.rst* — reStructuredText; `http://docutils.sourceforge.net/rst.html`). Он также может содержать файлы *requirements.txt*, *MANIFEST.in* и *setup.cfg*, которые рассмотрены в следующих разделах.

Тестирование пакета в процессе разработки

Если вы хотите тестировать свой пакет в то время, когда он только разрабатывается, установите его локально с помощью команды **`pip install -e`**. Флаг **`-e`** означает "editable" (редактируемый), а объяснение остальных деталей вы найдете в онлайн-документации (`https://packaging.python.org/tutorials/distributing-packages/#working-in-development-mode`).

Сценарий *setup.py*

Корневой каталог дистрибутива должен содержать сценарий Python с красноречивым названием *setup.py*. С теоретической точки зрения сценарий *setup.py* может содержать произвольный код на языке Python. Однако на практике он сводится к некоторым вариантам следующего кода.

```
from setuptools import setup, find_packages

setup( здесь задаются многочисленные аргументы )
```

Вы также должны импортировать класс `Extension`, если файл *setup.py* предназначен для обработки смешанного дистрибутива. Все действия определяются параметрами, предоставляемыми при вызове `setup`. Разумеется, ничто не запрещает вам поместить другие инструкции перед вызовом `setup`, чтобы организовать аргументы более понятным и удобочитаемым способом, чем если бы вы использовали встроенные инструкции как часть вызова `setup`. Например, строка *long_description* может быть предоставлена из отдельного файла.

```
with open('./README.rst') as f:
    long_desc=f.read()
```

Функция `setup` поддерживает лишь именованные аргументы, и общее количество аргументов, которые допускается предоставлять, очень велико. Именованные аргументы, передаваемые функции `setup`, можно условно разделить на три основные группы: метаданные, описывающие дистрибутив, информация о том, какие файлы входят в дистрибутив, и информация о зависимостях. В качестве простого примера можно привести сценарий *setup.py* для фреймворка Flask (см. раздел "Flask" в главе 20), отображающий некоторые из этих метаданных.

```
""" Здесь определяется __doc__ для long_description; не показано. """

import re
import ast
from setuptools import setup

_version_re = re.compile(r'__version__\s+=\s+(.*)')
with open('flask/__init__.py', 'rb') as f:
    version = str(ast.literal_eval(_version_re.search(
```

```
        f.read().decode('utf-8')).group(1)))

setup(
    name='Flask',
    version=version,
    url='http://github.com/pallets/flask/',
    license='BSD',
    author='Armin Ronacher',
    author_email='armin.ronacher@active-4.com',
    description='A microframework based on Werkzeug, Jinja2 '
                'and good intentions',
    long_description=__doc__,
    packages=['flask', 'flask.ext'],
    include_package_data=True,
    zip_safe=False,
    platforms='any',
    install_requires=[
        'Werkzeug>=0.7',
        'Jinja2>=2.4',
        'itsdangerous>=0.21',
        'click>=2.0',
    ],
    classifiers=[
        'Development Status :: 4 - Beta',
        'Environment :: Web Environment',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License',
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        'Programming Language :: Python :: 2',
        'Programming Language :: Python :: 2.6',
        'Programming Language :: Python :: 2.7',
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.3',
        'Programming Language :: Python :: 3.4',
        'Programming Language :: Python :: 3.5',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
        'Topic :: Software Development :: Libraries :: Python
         Modules'
    ],
    entry_points='''
        [console_scripts]
        flask=flask.cli:main
    '''

)
```

Метаданные, описывающие дистрибутив

Метаданные о дистрибутиве предоставляются посредством передачи описанных ниже именованных аргументов функции `setup` при ее вызове. Значения, связываемые с именем каждого аргумента, задаются в виде строк, главным образом предназначенных для чтения человеком. Спецификации формата этих строк в основном носят рекомендательный характер. Приведенные ниже объяснения и рекомендации, касающиеся полей метаданных, также не являются нормативными и соответствуют лишь общепринятым, но не универсальным соглашениям. К числу обязательных аргументов относятся следующие.

`author`
: Имя (имена) автора (авторов) материала, включенного (включенных) в данный дистрибутив. Вы всегда должны предоставлять эту информацию: авторы заслуживают подобной формы выражения признательности за проделанную ими работу.

`author_email`
: Адрес (адреса) электронной почты автора (авторов), указанного (указанных) в аргументе `author`. Как правило, следует предоставлять такую информацию. Однако вместо этого вы можете переадресовать людей к лицу, ответственному за сопровождение данного дистрибутива (`maintainer`), указав адрес его электронной почты в соответствующем поле (`maintainer_email`).

`classifiers`
: Список Trove-строк, классифицирующих ваш пакет. Каждая строка должна быть одной из тех, которые включены в список классификаторов, доступный на сайте PyPI (`https://pypi.python.org/pypi?%3Aaction=list_classifiers`).

`description`
: Краткое описание дистрибутива. Желательно, чтобы оно умещалось в одной строке, содержащей не более 80 символов.

`long_description`
: Развернутое описание дистрибутива, желательно в формате *.rst*, обычно предоставляемое в том виде, в каком оно содержится в файле *README*.

`license`
: Условия лицензионного соглашения в сжатой форме, обычно включающие ссылку на полный текст соглашения, который включен в дистрибутив в качестве отдельного файла или задан с помощью URL-адреса.

`name`
: Имя дистрибутива, заданное в виде допустимого идентификатора Python; см. описание критериев в документе **PEP 426** (`https://www.python.org/dev/peps/pep-0426/#name`). Если вы планируете выгружать свой проект на сайт

PyPI, то это имя не должно вступать в конфликт с именем любого другого проекта, уже находящегося в базе данных PyPI.

`url`
: URL-адрес, по которому можно найти дополнительную информацию о данном дистрибутиве, или значение `None`, если такого URL-адреса не существует.

`version`
: Номер версии данного дистрибутива, обычно структурированный в виде *`major.minor`* или имеющий еще более детализированную структуру. Рекомендованные схемы нумерации версий приведены в документе **PEP 440** (`https://www.python.org/dev/peps/pep-0440/`).

Следующие аргументы являются необязательными и могут предоставляться по мере необходимости.

`keywords`
: Список строк, которые, как ожидается, будут использоваться пользователями в качестве ключевых слов при выполнении поиска функциональности, реализуемой данным дистрибутивом. Вы должны предоставлять эту информацию, чтобы заинтересованные лица могли найти ваш пакет на сайте PyPI или с помощью других поисковых систем.

`maintainer`
: Имя лица, которое в настоящее время обеспечивает сопровождение данного дистрибутива. Эта информация должна предоставляться в том случае, если автор и лицо, осуществляющее сопровождение дистрибутива, — разные люди.

`maintainer_email`
: Адрес (адреса) электронной почты лица (лиц), указанного (указанных) в аргументе `maintainer`. Эта информация должна предоставляться лишь в том случае, если вы задаете аргумент `maintainer` и соответствующий человек хочет получать электронные сообщения, касающиеся сопровождения дистрибутива.

`platforms`
: Список известных платформ, на которых может работать данный дистрибутив. Эта информация предоставляется в том случае, если у вас есть причины полагать, что данный дистрибутив может работать не на любой платформе. Информация должна быть в разумной мере краткой, поэтому данное поле часто содержит ссылки на информацию, которую можно найти по заданным URL-адресам или в другом файле дистрибутива.

Содержимое дистрибутива

Дистрибутив может иметь смешанный состав и включать файл с исходным кодом на языке Python, C-расширения и файлы данных. Функция `setup` поддерживает необязательные именованные аргументы, детализирующие, какие файлы должны

помещаться в дистрибутив. Всякий раз, указывая пути к файлам, вы должны задавать их относительно корневого каталога дистрибутива и использовать символ / в качестве разделителя. Модуль `setuptools` соответствующим образом адаптирует расположение и разделитель в процессе установки дистрибутива. В частности, файлы *wheel* не поддерживают абсолютные пути: все пути задаются относительно каталога верхнего уровня вашего пакета.

В именованных аргументах `packages` и `py_modules` указываются не пути к файлам, а пакеты и модули Python соответственно. Поэтому в значениях этих именованных аргументов не должны указываться разделители путей или расширения имен файлов. Если вы указываете имена подпакетов в аргументе `packages`, используйте вместо этого точечный синтаксис Python (например, *`top_package.sub_package`*).

Исходные файлы на языке Python

По умолчанию функция `setup` ищет модули Python (перечисленные в значении именованного аргумента `py_modules`) в корневом каталоге дистрибутива, а пакеты (перечисленные в значении именованного аргумента `packages`) — в его подкаталогах.

Ниже описаны именованные аргументы функции `setup`, которые вы будете чаще всего использовать для детализации информации о том, какие файлы с исходным кодом на языке Python являются частью дистрибутива.

`entry_points`

`entry_points={'group':['name=P.m:obj',],}`

`entry_points` — это словарь, в котором хранится одна или несколько *групп*. Каждая группа *`group`* имеет список строк *`name=value`*, где *`name`* — идентификатор, а *`value`* — модуль *`P.m`* Python (часть *`P.`*, указывающая пакет, — необязательная), за которым следует указание функции, класса или другого объекта *`obj`* модуля *`m`* с использованием символа двоеточия (`:`) в качестве разделителя между именами модуля и объекта.

Группа *`group`* может представлять плагин, парсер или другую услугу. Более подробную информацию по этой теме вы найдете в разделе *Dynamic Discovery of Services and Plugins* онлайн-документации инструмента `setuptools` (`http://setuptools.readthedocs.io/en/latest/setuptools.html#dynamic-discovery-of-services-and-plugins`). Однако в большинстве случаев словарь `entry_points` используется для создания исполняемых сценариев: чаще всего в качестве аргумента *`group`* задаются сценарии `console_scripts` и `gui_scripts` (раздел "Что такое `entry_points`?").

`packages`

`packages=find_packages()|[`*`список строк с именами пакетов`*`]`

Вы можете импортировать и использовать функцию *`find_packages`* из модуля `setuptools` для автоматического обнаружения местоположения и включения пакетов и подпакетов в корневой каталог своего дистрибутива. Альтернативный вариант заключается в предоставлении списка пакетов. Для каждой строки с именем пакета *`p`*, содержащейся в этом списке, функция `setup` рассчитывает найти подкаталог *`p`* корневого каталога дистрибутива

и включить в дистрибутив файл *p/__init__.py*, наличие которого, как и других файлов *p/*.py* (т.е. всех модулей пакета *p*), является обязательным. Функция `setup` не выполняет поиск подпакетов *p*: если только вы не используете значение `find_packages`, то должны явно перечислить все подпакеты, а также пакеты верхнего уровня в значении именованного аргумента `packages`. Мы рекомендуем использовать значение `find_packages`, описание которого приведено ниже, чтобы избежать необходимости обновления пакетов (с риском пропустить какой-либо пакет) по мере наращивания дистрибутива.

`find_packages` `find_packages(where='.', exclude=())`

Здесь *`where`* указывает каталог, который обходит функция `find_packages` (включая его подкаталоги) для поиска и включения всех пакетов (и содержащихся в них модулях). По умолчанию `where` имеет значение `'.'`, соответствующее обычной нотации "текущего каталога", под которым в данном случае подразумевается корневой каталог дистрибутива. В аргументе `exclude` указываются имена, включая имена с метасимволами (например, `'tests'` или `'*.test'`), которые должны быть исключены из возвращаемого списка пакетов (обратите внимание на то, что функция `exclude` выполняется последней).

`py_modules` `py_modules=[` *`список строк с именами модулей`* `]`

Для каждой строки с именем пакета *m*, содержащейся в этом списке, функция `setup` рассчитывает найти файл *m.py* в корневом каталоге дистрибутива и включить его в дистрибутив. Используйте аргумент `py_modules` вместо `find_packages` в случае очень простого пакета, содержащего лишь несколько модулей и не имеющего подкаталогов.

Что такое `entry_points`?

Аргумент `entry_points` (точки входа) позволяет сообщить установщику (обычно программе `pip`) о необходимости регистрации в ОС плагинов, служб или сценариев и, если в этом есть потребность, создания специфического для платформы исполняемого объекта. В качестве элементов *`group`* словаря `entry_points` в основном используются `console_scripts` (заменяет именованный аргумент `scripts`, признанный устаревшим) и `gui_scripts`. Поддерживаются также другие плагины и службы (например, парсеры), но в данной книге они не рассматриваются. Для получения более подробной информации по этой теме обратитесь к руководству пользователя по созданию пакетов в Python (`https://packaging.python.org/`).

Когда программа `pip` устанавливает пакет, она регистрирует имя *`name`* каждой точки входа в ОС и создает соответствующий исполняемый объект (включая программу запуска *.exe*-файлов в Windows), для выполнения которого впоследствии будет достаточно ввести это имя в ответ на приглашение к вводу в окне терминала вместо того, чтобы, например, вводить команду **`python -m mymodule`**.

Сценарии — это файлы с исходным кодом на языке Python, которые предназначены для выполнения в качестве основной программы (см. раздел "Основная программа" в главе 6), обычно запускаемой из командной строки. Первой строкой каждого файла сценария должна быть "магическая" строка, т.е. строка, начинающаяся с символов `#!` и содержащая подстроку `python`. Кроме того, каждый сценарий должен заканчиваться следующим блоком кода.

```
if __name__ == '__main__':
    mainfunc()
```

Чтобы программа `pip` установила ваш сценарий в качестве исполняемого объекта, включите его в список `console_scripts` (или `gui_scripts` в случае необходимости) словаря `entry_points`. В дополнение к основной функции вашего сценария (или вместо нее) можете включить в словарь `entry_points` любую другую функцию для ее регистрации в качестве интерфейса сценария. Вот как может выглядеть определение словаря `entry_points`, содержащего как ключ `console_scripts`, так и ключ `gui_scripts`.

```
entry_points={
    'console_scripts': ['example=example:mainfunc',
                        'otherfunc=example:anotherfunc',
                       ],
    'gui_scripts': ['mygui=mygui.gui_main:run',
                   ],
    },
```

Выполнив установку, введите **`example`** в ответ на выведенное в окне терминала приглашение к вводу, чтобы выполнить сценарий `mainfunc`, содержащийся в модуле `example`. Если вместо этого вы введете **`otherfunc`**, то система выполнит сценарий `anotherfunc`, также содержащийся в модуле `example`.

Файлы данных и другие файлы

Чтобы поместить файлы любого рода в дистрибутив, предоставьте описанные ниже именованные аргументы. В большинстве случаев вы будете указывать список своих файлов данных с помощью аргумента `package_data`. Именованный аргумент `data_files` используется для указания файлов, которые вы хотите установить в каталогах, внешних по отношению к вашему пакету. Однако мы не рекомендуем так поступать ввиду сложного и непоследовательного поведения процесса установки в этом случае, о чем говорится ниже.

`data_files` `data_files=[`*список пар* `(`*target_directory*`, `*list_of_files*`)]`

Значением именованного аргумента `data_files` является список пар. Первый элемент каждой пары — строка с именем *целевого каталога* (т.е. каталога, в который модуль `setuptools` помещает файлы данных при установке дистрибутива); второй элемент — список строк, которые содержат пути к файлам, помещаемым в целевой каталог.

В процессе установки из файлов *wheel* каждый целевой каталог помещается в качестве подкаталога в каталог, указанный в переменной Python `sys.prefix` для чистого дистрибутива или в переменной `sys.exec_prefix` — для смешанного. Установка из *sdist* с помощью программы `pip` использует `setuptools` для размещения целевых каталогов относительно каталога `site_packages`, но установка без использования `pip` и с помощью `distutils` характеризуется тем же поведением, что и в случае файлов *wheel*. С учетом описанной непоследовательности поведения мы не рекомендуем использовать аргумент `data_files`.

package_data `package_data={`*k*`:`*список шаблонов поиска*`, ...}`

Значением именованного аргумента `package_data` является словарь. Каждый ключ — это строка с именем пакета, в котором следует искать файлы данных. Соответствующим значением является список шаблонов поиска файлов данных, включаемых в дистрибутив. Шаблоны могут включать подкаталоги (указанные с помощью относительных путей, в которых в качестве разделителя используется символ косой черты, `/`, причем даже на компьютерах Windows). Пустая строка пакета, `''`, рекурсивно включает файлы, находящиеся в любом подкаталоге, соответствующем шаблону. Например, элемент словаря `'': ['*.txt']` включает все текстовые файлы, расположенные в каталоге верхнего уровня или его подкаталогах. В процессе установки пакет `setuptools` помещает каждый файл в соответствующие каталоги, располагая их относительно каталога `site_packages`.

C-расширения

Чтобы поместить в дистрибутив расширения на языке C, задайте следующий именованный аргумент.

ext_modules `ext_modules=[` *список экземпляров класса Extension* `]`

Подробная информация о каждом расширении предоставляется в виде аргументов во время инстанциализации класса `setuptools.Extension`. Конструктор `Extension` принимает два обязательных аргумента и большое количество необязательных именованных аргументов. В качестве простейшего примера можно привести следующий код:

```
ext_modules=[Extension('x',sources=['x.c'])]
```

Ознакомьтесь с описанием конструктора класса `Extension`.

Extension `class Extension(`*name*`, `*sources*`, **`*kwds*`)`

Аргумент *name* — это строка с именем модуля для C-расширения. Строка *name* может включать точки, указывающие на то, что модуль расширения находится в пакете. Аргумент *sources* — это список файлов с исходным кодом на языке C, которые должны компилироваться для создания расширения. Каждый элемент *sources* представляет собой строку, которая содержит путь к файлу с исходным

кодом, имеющему расширение имени файла *.c* (путь задается относительно корневого каталога дистрибутива). Аргумент `kwds` позволяет передать другие, необязательные именованные аргументы конструктору `Extension`, о чем пойдет речь далее.

Класс `Extension` поддерживает и другие расширения имен файлов, кроме *.c*, указывающие на другие языки, которые можно использовать для программирования расширений для Python. На платформах, имеющих компилятор C++, расширение имени файла *.cpp* указывает на файлы с исходным кодом на языке C++. К числу других расширений имен файлов, поддержка которых зависит от платформы и наличия различных плагинов для `setuptools`, относятся *.f* для файлов Fortran, *.i* для файлов SWIG и *.pyx* для файлов Cython. Более подробная информация об использовании различных языков программирования для создания расширений Python содержится в разделе "Расширение Python без использования C API" главы 24.

В большинстве случаев ваше расширение не нуждается ни в какой дополнительной информации, кроме аргументов `name` и `sources`. Обратите внимание на то, что вы должны перечислить любые заголовочные *.h*-файлы в своем файле *MANIFEST.in*. Пакет `setuptools` выполнит все, что необходимо для того, чтобы сделать каталог заголовков Python и библиотеку Python доступными для компиляции и компоновки ваших расширений, и предоставит флаги или опции, необходимые компилятору или компоновщику для создания расширений на данной платформе.

Если для успешной компиляции и компоновки вашего расширения требуется дополнительная информация, вы можете предоставить ее посредством именованных аргументов класса `Extension`. Такие аргументы могут мешать кросс-платформенной переносимости вашего дистрибутива. В частности, всякий раз, когда вы указываете пути к файлам или каталогам в качестве значений подобных аргументов, их следует задавать относительно корневого каталога дистрибутива. Но если вы планируете распространять свои расширения на другие платформы, то должны проверять, действительно ли необходимо предоставлять информацию о сборке посредством передачи аргументов конструктору класса `Extension`. Иногда от этого можно избавиться, тщательно продумав код на уровне языка C.

Ниже описаны некоторые именованные аргументы, которые можно передавать конструктору `Extension`.

`define_macros = [ (macro_name, macro_value) ... ]`

Каждый из элементов `macro_name` и `macro_value` является строкой, представляющей соответственно имя и значение макроопределения для препроцессора C, которое по своему действию эквивалентно следующей директиве препроцессора C: `#define macro_name macro_value`.

Элемент `macro_value` может также иметь значение `None`, обеспечивающее тот же эффект, что и следующая директива препроцессора C: `#define macro_name`.

`extra_compile_args = [ список строк compile_arg ]`
Каждая из строк, указанных в качестве значения в аргументе `extra_compile_args`, включается в состав других аргументов командной строки при каждом вызове компилятора C.

`extra_link_args = [ список строк link_arg ]`
Каждая из строк, указанных в качестве значения в аргументе `extra_link_args`, включается в состав других аргументов командной строки для компоновщика.

`extra_objects = [ список строк object_name ]`
Каждая из строк, указанных в качестве значения в аргументе `extra_objects`, определяет имя объектного файла для связывания, причем расширение имени объектного файла не должно указываться: пакет `distutils` помогает вам в обеспечении кросс-платформенной переносимости, добавляя расширение имени файла, соответствующее платформе (*.o* на Unix-подобных платформах и *.obj* на платформах Windows).

`include_dirs = [ список строк directory_path ]`
Каждая из строк, указанных в качестве значения в аргументе `include_dirs`, идентифицирует каталог, предоставляемый компилятору в числе других каталогов, в которых следует искать заголовочные файлы.

`libraries = [ список строк library_name ]`
Каждая из строк, указанных в качестве значения в аргументе `libraries`, указывает имя библиотеки для связывания. Указывать расширение имени файла или какой-либо префикс в качестве части имени библиотеки не следует. Пакет `distutils`, действуя совместно с компоновщиком, помогает вам в обеспечении кросс-платформенной переносимости, добавляя расширение имени файла и префикс, соответствующие платформе (расширение имени файла *.a* и префикс *lib* на Unix-подобных платформах и расширение имени файла *.lib* на платформах Windows).

`library_dirs = [ список строк directory_path ]`
Каждая из строк, указанных в качестве значения в аргументе `library_dirs`, идентифицирует каталог, предоставляемый компилятору в числе других каталогов, в которых следует искать библиотечные файлы.

`runtime_library_dirs = [ список строк directory_path ]`
Каждая из строк, указанных в качестве значения в аргументе `runtime_library_dirs`, идентифицирует каталог, в котором следует искать динамически загружаемые библиотеки во время выполнения.

`undef_macros = [ список строк macro_name ]`
Каждая из строк *macro_name*, указанных в качестве значения в аргументе `undef_macros`, является строкой, представляющей имя макроопределения для препроцессора C, которое по своему действию эквивалентно следующей директиве препроцессора C: `#undef` *macro_name*.

Зависимости и требования

У вас имеется возможность указать список зависимостей с помощью именованных аргументов в файле *setup.py* или в файле требований (раздел "Файл *requirements.txt*").

install_requires `install_requires=['pkg',['pkg2>=n.n']]`

Аргумент `install_requires` принимает список имен пакетов в виде строк с возможностью указания версионных требований `n.n`. Для получения более подробной информации о том, как обрабатываются спецификаторы версий, обратитесь к документу **PEP 440** (`https://www.python.org/dev/peps/pep-0440/#version-specifiers`). Старайтесь предоставить как можно более широкие версионные требования для вашей программы. Используйте этот именованный аргумент для задания зависимостей, минимально необходимых для выполнения вашей программы. Программа `pip` автоматически вызывает команду `pip install pkg` для каждого аргумента `pkg` с целью установки зависимостей.

extras_require `extras_require={'rec':['pkgname']}`

Аргумент `extras_require` принимает словарь, содержащий рекомендованные дополнительные зависимости: `'rec'` — строка ключа, описывающая рекомендацию (например, `'PDF'`), `'pkgname'` — значение. Программа `pip` не устанавливает автоматически эти зависимости, но пользователь может выбрать устанавливаемые зависимости *rec* во время установки основного пакета *mpk* с помощью команды **`pip install mpk[rec]`**.

Файл *requirements.txt*

Вы также можете использовать файл *requirements.txt*. Этот файл, если вы его предоставляете, должен содержать вызовы `pip install` — по одному в строке. В частности, файл *requirements.txt* целесообразно применять для воссоздания конкретного окружения в процессе установки или для принудительного включения определенных версий зависимостей.

Если пользователь введет следующую команду в командной строке, то программа `pip` установит все элементы, перечисленные в файле *requirements.txt*, без каких-либо гарантий соблюдения определенного порядка их установки:

```
pip install -r requirements.txt
```

Списки `install_requires`, `extras_require` и файл *requirements.txt*

Программа `pip` автоматически обнаруживает и устанавливает лишь зависимости, указанные в списке `install_requires`. Записи словаря `extras_require` должны вручную выбираться во время установки или использоваться в списке `install_requires` сценария *setup.py* другого пакета (например, ваш пакет используется в качестве библиотеки другими пакетами). Файл *requirements.txt* должен вручную

выполняться пользователем. Более подробные сведения относительно использования файлов *requirements.txt* приведены в документации программы `pip` (`https://pip.pypa.io/en/latest/user_guide/#requirements-files`).

Файл *MANIFEST.in*

В процессе пакетирования исходного дистрибутива библиотека `setuptools` по умолчанию вставляет в него следующие файлы:

- все файлы с исходным кодом на языках Python (*.py*) и C, явно указанные в аргументе `packages` или найденные с помощью функции `find_packages` в файле *setup.py*;
- файлы, перечисленные в аргументах `package_data` и `data_files` в файле *setup.py*;
- сценарии или плагины, определенные в аргументе `entry_points` в файле *setup.py*;
- тесты, которые содержатся в файлах *test/test*.py*, расположенных в корневом каталоге дистрибутива, если они не исключены функцией `find_packages`;
- файлы *README.rst* или *README.txt* (если таковые имеются), *setup.cfg* (если таковые имеются) и *setup.py*.

Чтобы добавить другие файлы в дистрибутив, поместите в корневой каталог дистрибутива *шаблон манифеста* — файл *MANIFEST.in*, в строках которого содержатся последовательно применяемые правила, определяющие добавление (`include`) или исключение (`prune`) файлов из списка файлов, включаемых в дистрибутив. Более подробную информацию по этой теме вы найдете в документации Python (`https://docs.python.org/3/distutils/sourcedist.html#specifying-the-files-to-distribute`). Если в вашем проекте имеются C-расширения (перечисленные в именованном аргументе `ext_modules` в файле *setup.py*), то для гарантии включения заголовков путь к любому из нужных заголовочных *.h*-файлов должен быть указан в файле *MANIFEST.in* в строке вида `graft /dir/*.h`, где *dir* — относительный путь к заголовкам.

Файл *setup.cfg*

Файл *setup.cfg* предоставляет подходящие значения по умолчанию для параметров команд времени сборки. Например, чтобы в процессе сборки всегда создавались универсальные файлы *wheel* (раздел "Создание файлов *wheel*"), добавьте в файл *setup.cfg* следующие строки:

```
[bdist_wheel]
universal=1
```

Распространение пакета

Подготовив файл *setup.py* (и другие файлы), вы сможете приступить к распространению своего пакета, которое включает следующие этапы.

1. Создайте (упакуйте) дистрибутив в файле формата *wheel* или другого формата архивирования.
2. Зарегистрируйте свой пакет, если это необходимо, в репозитории.
3. Выгрузите свой пакет в репозиторий.

Создание дистрибутива

В прошлом упакованный дистрибутив исходного кода, созданный с помощью команды `python setup.py sdist`, был наиболее полезным файлом, который можно было получить с помощью пакета `distutils`. Если вы распространяете пакеты с C-расширениями для ответвлений Linux, то по-прежнему будете использовать команду `sdist`. А если вам безусловно требуются абсолютные пути (а не относительные) для установки определенных файлов, перечисленных в аргументе `data_files` в файле *setup.py*, то вы должны будете использовать `sdist`. (См. обсуждение аргумента `data_files` в руководстве пользователя по созданию пактов Python; `https://packaging.python.org/tutorials/distributing-packages/`.) Но если вы упаковываете файлы, содержащие чистый Python-код или платформозависимые C-расширения для macOS или Windows, то сможете облегчить жизнь большинству пользователей, архивируя дистрибутивы в файлы *wheel*.

Файлы *wheel*

Файлы формата *wheel* ("колесо") — это новый способ упаковки модулей и пакетов Python для распространения, заменяющий бывший ранее предпочтительным формат файлов *egg*. Файлы *wheel* рассматриваются как "готовые" дистрибутивы в том смысле, что их установка не требует от пользователя выполнения их сборки. Файлы *wheel* обновляются быстрее, чем файлы *egg*, и могут работать на многих платформах, поскольку они не включают *.pyc*-файлов. Кроме того, вашим пользователям не понадобится компилятор для C-расширений на компьютерах Mac и Windows. Наконец, для *wheel*-файлов предоставляется хорошая поддержка на сайте PyPI, и они являются предпочтительными для программы `pip`.

Для создания *wheel*-файлов вам понадобится установить пакет `wheel`, выполнив следующую команду в окне терминала: `pip install wheel`.

Чистые файлы *wheel*

Файлы *wheel* считаются *чистыми*, если они содержат только код на языке Python, и *смешанными*, если содержат расширения на языке C (или других языках программирования). Чистые *wheel*-файлы могут быть *универсальными*, если код может

выполняться как в версии v2, так и в версии v3 без его трансляции с помощью утилиты `2to3` (раздел "Исходный код v2 с преобразованием в v3" в главе 26). Универсальные *wheel*-файлы — кросс-платформенные (однако будьте осторожны: не все встроенные объекты Python работают одинаково на всех платформах). Версионно-зависимые чистые файлы *wheel* также являются кросс-платформенными, но требуют использования конкретной версии Python. Если ваш чистый дистрибутив нуждается в трансляции с помощью утилиты `2to3` (или выдвигает различные требования в версиях v2 и v3, связанные, например, с аргументами `install_requires`, указываемыми в файле *setup.py*), то вам нужно создать два файла *wheel*: один для версии v2, другой — для версии v3.

Смешанные файлы *wheel*

В случае чистых дистрибутивов предоставление *wheel*-файлов является лишь вопросом удобства для пользователей. В случае же смешанных дистрибутивов создание готовых, не требующих сборки, форм обеспечивает уже не просто удобства. Смешанный дистрибутив по определению включает код, написанный не только на языке Python, но и, как правило, код на языке C. Если вы не предоставляете дистрибутив в готовом виде, то у пользователей должен быть установлен подходящий C-компилятор, чтобы они смогли выполнить сборку и установку дистрибутива. Кроме того, установка дистрибутива исходного кода может оказаться довольно сложной для конечных пользователей, недостаточно искушенных в программировании. Поэтому в случае смешанных пакетов рекомендуется предоставлять как дистрибутив исходного кода в формате *.tar.gz*, так и файл *wheel* (или несколько таких файлов). Для этого у вас должен быть установлен соответствующий компилятор C. Смешанные *wheel*-файлы работают только на компьютерах с той же платформой (например, macOS, Windows) и архитектурой (например, 32- и 64-разрядная), что и компьютер, на котором выполнялась их компиляция и сборка.

Создание файлов *wheel*

Во многих случаях для создания *wheel*-файла вам потребуется выполнить всего лишь одну строку. Для чистого (содержащего только код на языке Python) и универсального (работающего без трансляции как в версии v2, так и в версии v3) дистрибутива вам достаточно выполнить в каталоге верхнего уровня дистрибутива следующую команду:

```
python setup.py bdist_wheel --universal
```

В результате ее выполнения будет создан *wheel*-файл, который можно будет установить на любой платформе. Если вы создаете чистый пакет, который работает в обеих версиях, v2 и v3, но требует трансляции с помощью утилиты `2to3` (или, например, требует указания в файле *setup.py* различных значений аргументов `install_requires` для версий v2 и v3), то вам понадобится создать два файла *wheel*: один для версии v2, другой — для версии v3:

```
python2 setup.py bdist_wheel
python3 setup.py bdist_wheel
```

Для чистого пакета, который может работать только в версии v2 или только в версии v3, используйте только подходящую версию Python для создания одного *wheel*-файла. Версионно-зависимые чистые *wheel*-файлы работают на любой платформе, на которой установлена подходящая версия Python.

Не используйте флаг `--universal` со смешанными версионно-зависимыми пакетами

Команда `bdist_wheel --universal` *не определяет*, является ли ваш пакет смешанным или версионно-зависимым. В частности, если пакет содержит C-расширения, то вы *не должны* создавать универсальный *wheel*-файл. Вместо этого программа `pip` переопределит *wheel*-файл вашей платформы или дистрибутив исходного кода, чтобы установить универсальный файл *wheel*.

Смешанные *wheel*-файлы можно создавать для macOS или Windows, но только на тех платформах, которые использовались для их создания. Команда `python setup.py bdist_wheel` (без флага `--universal`) автоматически обнаруживает расширения и создает подходящий *wheel*-файл. Одно из преимуществ создания *wheel*-файлов заключается в том, что ваши пользователи получают возможность устанавливать пакеты с помощью команды `pip install`, независимо от того, установлен ли у них C-компилятор, если они выполняются на той же платформе, которая использовалась для создания *wheel*-файла.

Смешанные *wheel*-файлы Linux

К сожалению, из-за различий в дистрибутивах Linux (см. документ **PEP 513**; `https://www.python.org/dev/peps/pep-0513/#id35`) распространение смешанных *wheel*-файлов Linux сопряжено с некоторыми трудностями. Каталог PyPI не принимает смешанные *wheel*-файлы Linux, если они не помечены как *manylinux*. В настоящее время сложный процесс создания кросс-дистрибутивных файлов *wheels* включает использование образов Docker и инструмента `auditwheel`. Для получения более подробной информации по этой теме обратитесь к документации проекта *manylinux* (`https://github.com/pypa/manylinux`).

Выполнив команду `python setup.py bdist_wheel`, вы получите *wheel*-файл с именем наподобие *mypkg-0.1-py2.py3-none-any.whl*, расположенный в (новом, если файл *setup.py* выполнялся в первый раз) каталоге *dist/*. Для получения более подробной информации относительно правил именования и маркирования *wheel*-файлов обратитесь к документу **PEP 425** (`https://www.python.org/dev/peps/pep-0425/`). Чтобы проверить, какие файлы были включены в *wheel*-файл в результате его

сборки, не устанавливая и не распаковывая его, можно воспользоваться командой `unzip -l mypkg`, поскольку *wheel*-файл — это всего лишь zip-архив с подходящими метаданными.

Создание дистрибутива исходного кода

Чтобы создать дистрибутив исходного кода для своего проекта, выполните в каталоге верхнего уровня своего пакета следующую команду:

```
python setup.py sdist
```

В результате выполнения этой команды будет создан *.tar.gz*-файл (если она создает *.zip*-архив на платформе Windows, вам может понадобиться использовать флаг `--formats` для указания формата архива).

Файлы *.tar.gz* предпочтительны для каталога PyPI

Не пытайтесь выгружать в каталог PyPI одновременно *.zip*- и *.tar.gz*-файлы, иначе вы получите сообщение об ошибке. В большинстве случаев придерживайте файлов *.tar.gz*.

Впоследствии вы сможете выгрузить свой *.tar.gz*-файл в каталог PyPI или распространить иным образом. Ваши пользователи должны будут распаковать и установить его, используя, как правило, команду `python setup.py install`. Для получения более подробной информации относительно дистрибутивов исходного кода обратитесь к онлайн-документации (`https://docs.python.org/3.5/distutils/sourcedist.html#`).

Регистрация и выгрузка пакетов в репозиторий

Как только вы создадите *wheel*-файл или пакет дистрибутива исходного кода, вы сможете выгрузить его в какой-либо репозиторий, чтобы упростить его распространение среди пользователей. Для этого можно использовать локальное хранилище, например частный репозиторий компании, или общедоступный репозиторий наподобие PyPI. В прошлом сценарий *setup.py* использовался для сборки и немедленной выгрузки дистрибутива, однако в связи с проблемами безопасности такая практика в настоящее время не рекомендуется. Вместо этого вы должны использовать один из сторонних модулей, таких как `twine` или `Flit` для предельно простых пакетов, о чем можно прочитать в документации `Flit` (`https://pypi.python.org/pypi/flit`). Существуют планы относительно слияния `twine` и `pip`: следите за обновлением информации по этому вопросу в руководстве пользователя по созданию пакетов Python (`https://packaging.python.org/`).

С использованием пакета `twine` не связаны какие-либо трудности. Чтобы загрузить его, выполните команду `pip install twine`. Вам понадобится создать файл *~/.pypirc* (предоставляющий информацию о репозитории), а затем зарегистрировать или выгрузить пакет с помощью простой команды.

Ваш файл *~/.pypirc*

В документации `Twine` рекомендуется, чтобы вы создали файл *~/.pypirc* (т.е. файл *.pypirc*, расположенный в вашем домашнем каталоге), содержащий перечень репозиториев, в которые вы хотите выгрузить свой пакет, в том числе ваше имя пользователя и пароль. Поскольку файлы *.pypirc* обычно сохраняются в незашифрованном виде, вы должны установить для битов разрешений доступа значение `600` (на Unix-подобных платформах) или не сохранять пароль и вводить его в ответ на соответствующее приглашение всякий раз, когда выполняете `twine`. Ниже приведен пример содержимого файла `.pypirc`.

```
[distutils]
index-servers=
    testpypi
    pypi
    warehouse
    myrepo

[testpypi]
repository=https://testpypi.python.org/pypi
username=yourusername
password=yourpassword

[pypi]
repository=https://pypi.python.org/pypi
username=yourusername
password=yourpassword

[warehouse]
repository=https://upload.pypi.org/legacy/
username=yourusername
password=yourpassword

[myrepo]
repository=https://otherurl/myrepo
username=yourusername
password=yourpassword
```

Регистрация и выгрузка пакетов в каталог PyPI

Если вы ранее не использовали каталог PyPI, создайте учетную запись, указав имя пользователя и пароль. Это можно сделать на стадии регистрации дистрибутива, но все же лучше сделать это в режиме онлайн на сайте PyPI (`https://pypi.python.org/pypi`). Вам также стоит создать учетную запись на тестовом сайте (`https://testpypi.python.org/pypi`), чтобы попрактиковаться в выгрузке пакетов во временное хранилище, прежде чем выгружать их в общедоступный репозиторий.

В процессе выгрузки своих пакетов в каталог PyPI от вас может потребоваться регистрация пакета, если вы делаете это в первый раз. Чтобы указать, какой именно пакет вы регистрируете, *wheel*-файл или дистрибутив исходного кода, используйте соответствующую команду (простое использование `dist/*` не работает):

```
twine register -r repo dist/mypkg.whl # для wheel
```

или

```
twine register -r repo dist/mydist.tar.gz # для sdist
```

Здесь `repo` — это конкретный репозиторий, в котором вы регистрируете пакет, из числа тех, которые указаны в файле *~/.pypirc*. Вы также можете поступить иначе, предоставив URL-адрес репозитория в командной строке с флагом `--repository-url`.

Warehouse — это новый сервер PyPI. Регистрация больше не требуется. Теперь вы можете добавлять его в свои файлы *~/.pypirc*, как показано в предыдущем примере. Вызов `twine` без параметра `-r` должен выполнить выгрузку в корректную версию PyPI по умолчанию.

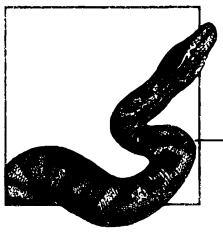
Зарегистрировав пакет, вы можете выгрузить его, выполнив следующую команду в каталоге верхнего уровня своего пакета:

```
twine upload -r repo dist/*
```

При этом `twine` найдет последнюю версию вашего дистрибутива и выгрузит ее в репозиторий, указанный в вашем файле *.pypirc* (или же вы можете предоставить URL-адрес репозитория в командной строке с помощью флага `--repository-url`).

С этого момента ваши пользователи смогут установить ваш пакет с помощью команды `pip install` *пакет* `--user`. (Или просто команды `pip install` в виртуальном окружении. Пользователи поступят благоразумно, если будут избегать использования команды `pip install` в невиртуальном окружении, поскольку это повлияет на всю установку Python для всех пользователей).

Организация Python Packaging Authority (PyPA; `https://www.pypa.io/en/latest/`) продолжает работать над усовершенствованием системы пакетирования и распространения пакетов Python. Вы также можете внести свой вклад в эту работу (`https://www.pypa.io/en/latest/help/`), присылая кодовую базу или документацию, публикуя сообщения об обнаруженных ошибках и участвуя в обсуждении системы пакетирования Python.



26

Переход с версии 2.x на версию 3.x и сосуществование различных версий

Самый ранний выпуск Python 3 (предшественник того, что мы называем версией v3 в данной книге) появился в 2008 году как "Python 3.0": он не был производственного качества, да это и не планировалось. Его основным назначением было дать программистам возможность адаптироваться к изменениям в синтаксисе и семантике. Некоторые элементы этой реализации работали неудовлетворительно (в частности, это касается средств ввода/вывода, в настоящее время намного улучшенных). На момент написания этой главы текущими были выпуски 2.7.12 и 3.5.2, тогда как версия 3.6.0 была выпущена в декабре 2016 года, когда книга была почти написана (в результате чего в ней содержатся только примечания о новинках версии 3.6, к тому же базирующиеся лишь на нашем изучении бета-версии 3.6, поэтому говорить о полном охвате возможностей версии 3.6 мы не можем).

Основная команда разработчиков Python проделала огромную работу по обеспечению ретроподдержки возможностей версии v3 в версии v2, но эта деятельность уже остановлена, и дальнейшая разработка версии v2 прекращена (некоторые возможности версии v3, портированные обратно в версию v2, доступны по команде `pip install`, о чем неоднократно упоминалось в книге). В настоящее время выпускаются лишь исправления версии v2, касающиеся обеспечения безопасности, но и это когда-нибудь должно закончиться, поскольку поддержка версии v2 будет официально прекращена в 2020 году. В своем выступлении на конференции PyCon в 2014 году (`https://www.youtube.com/watch?v=0Ef9GudbxXY`) Гвидо ван Россум публично исключил любую возможность появления выпуска 2.8. Если вы решитесь продолжать

работать с версией v2 и после 2020 года, то будете делать это на собственный страх и риск.

Для тех, кто овладел системой сборки пакетов Python или кто в состоянии нанять независимых разработчиков для поддержки своих программ, ориентированных на версию v2, стратегия "приверженности к версии v2" может оставаться вполне жизнеспособной и после 2020 года. Однако для большинства читателей первоочередной задачей, вероятно, будет преобразование существующего v2-кода в код для версии v3.

Подготовка к переходу на версию Python 3

Если вы не знакомы детально с обеими версиями Python, то, прежде чем браться за преобразование своего проекта, ориентированного на версию v2, вам будет полезно прочитать официальное руководство по портированию кода из Python 2 в Python 3, которое содержится в официальной документации Python (`https://docs.python.org/3/howto/pyporting.html`). Среди всего прочего, одна из содержащихся в этом руководстве рекомендаций перекликается с той, которую давали и мы в отношении версии v2 (речь идет о том, что обеспечивается лишь поддержка версии Python 2.7[1]).

Убедитесь в том, что ваши тесты покрывают как можно бо́льшую часть кода проекта, чтобы тестирование позволило выявить ошибки, вызванные конфликтами между версиями. Стремитесь добиться по крайней мере 80% покрытия кода тестами. Обеспечение покрытия свыше 90% может натолкнуться на трудности, поэтому не тратьте слишком много времени на достижение чрезмерно завышенного стандарта (хотя применение ложных объектов, о которых шла речь в разделе "Тестирование" главы 16, может способствовать увеличению если не глубины, то, по крайней мере, широты охвата кода блочными тестами).

Вы можете обновить свою кодовую базу v2 в полуавтоматическом режиме, используя либо утилиту `2to3` версии Python 2, либо созданные поверх нее пакеты `modernize` (`http://python-modernize.readthedocs.io/en/latest/`) и `futurize` (`http://python-future.org/overview.html#automatic-conversion-to-py2-3-compatible-code`). (В ранее упомянутом руководстве по портированию кода Python содержится неплохой совет относительно выбора между двумя последними инструментами, о чем пойдет речь в разделе "Поддержка v2/v3 с использованием единого дерева исходного кода"). Среда непрерывной интеграции обеспечит вам уверенность в том, что вы придерживаетесь наилучшей практики кодирования, отвечающей принятым стандартам. Кроме того, используйте такие инструменты, как `coverage` (`https://pypi.python.org/pypi/coverage`) и `flake8` (`https://pypi.python.org/pypi/flake8`), для грубой количественной оценки качества вашего кода.

[1] В качестве дополнительного стимула в этом направлении: PyPI и *pip* больше не поддерживают версию Python 2.6.

Даже если вы по-прежнему используете версию v2, вы можете приступить к выявлению некоторых проблем, связанных с переходом к версии v3, используя флаг `-3` командной строки при запуске интерпретатора Python, чтобы получать предупреждения в случае несовместимости, устранение которой с помощью утилиты `2to3` маловероятно. Кроме того, добавьте флаг `-Werror`, чтобы преобразовывать предупреждения в ошибки, тем самым гарантируя, что вы их не пропустите. Однако не думайте, что этот прием позволит вам выяснить все, что нужно: он не позволяет обнаруживать некоторые тонкие проблемы, такие как обработка данных из файлов, открытых в текстовом режиме, как байтов.

Всякий раз, когда вы используете автоматическое преобразование кода, просматривайте результаты процесса преобразования. Идеальная трансляция таких динамических языков, как Python, невозможна. Несмотря на то что тестирование помогает обнаруживать проблемы, оно не позволяет выявить все отклонения от нормы.

Читатели, работающие с большими кодовыми базами или серьезно заинтересованные в разработке мультиверсионного программного обеспечения, также должны прочитать книгу *Supporting Python 3: An in-depth guide* (Lennart Regebro), которая доступна как в печатном, так и в электронном виде (`python3porting.com`). Версия Python 2.7 еще не была выпущена к моменту выхода второго издания книги, однако набор возможностей этой версии хорошо известен, и в упомянутой книге содержится множество полезных и подробных рекомендаций, которые мы не повторяем в своей книге. Вместо этого мы описываем наиболее важные ситуации, которые могут порождать проблемы несовместимости, в разделе "Минимизация синтаксических различий", а также некоторые инструменты, которые могут пригодиться вам для преобразования кода.

Инструкции импорта `from __future__`

`__future__` — это модуль стандартной библиотеки, содержащий ряд средств, описанных в онлайн-документации (`https://docs.python.org/3/library/__future__.html`), которые упрощают миграцию между версиями. Он не похож ни на какой другой модуль, поскольку импортирование функциональности с его помощью может влиять не только на семантику, но и на синтаксис вашей программы. Подобные инструкции импорта должны располагаться в самом начале вашего кода. Каждая "будущая возможность" активизируется посредством такой инструкции:

```
from __future__ import feature
```

Здесь *`feature`* — это функциональная возможность, которую вы хотите использовать. В частности, для обеспечения наилучшей совместимости мы рекомендуем помещать в начале кода ваших модулей версии v2 следующую строку:

```
from __future__ import (print_function, division, absolute_import)
```

В версии v2 эта инструкция устанавливает совместимость с механизмами вывода на печать, деления и импорта версии v3.

Минимизация синтаксических различий

Имеется ряд различий между версией Python 3.5 (на которую мы ссылаемся в данной книге как на версию v3) и предыдущими версиями Python 3. Несмотря на то что существуют возможности справиться с этим, мы решили ради краткости исключить соответствующие методики из рассмотрения. Также подчеркнем, что под версией v2 мы всегда подразумеваем версии 2.7.*x*, а не любую из предыдущих версий Python 2.

Избегайте использования классов старого стиля

Несмотря на то что обновление до объектной модели в версии Python 2.2 в основном обладало обратной совместимостью, существует ряд важных и тонких различий, особенно если используется множественное наследование. Для гарантии того, что эти различия вас не коснутся, убедитесь в том, что в вашем v2-коде используются только классы нового стиля.

Простейший способ добиться этого заключается в том, чтобы ваши классы явно наследовались от объекта `object`. Возможно, вам понадобится скорректировать порядок указания других базовых классов, если таковые имеются, для уверенности в том, что ссылки указывают на нужные методы. В коррекции нуждается бо́льшая часть программного обеспечения, но чем сложнее ваша иерархия множественного наследования, тем более тщательно должна быть проделана эта работа.

Вы можете добавить строку `__metaclass__ = type` в самом начале кода ваших модулей для гарантии того, что все классы в коде, не имеющие базовых классов, будут классами нового стиля (см. раздел "Определение метакласса в версии Python v2" в главе 4; эта инструкция безобидна в версии v3).

`print` как функция

Одно из наиболее бросающихся в глаза различий между версиями v2 и v3 — переход от `print` как инструкции к `print` как к функции. Чтобы минимизировать этот разрыв, всегда используйте функцию `print`, активизировав ее с помощью рекомендованной ранее инструкции `__future__ imports`. (Использование этой инструкции предполагается во всех примерах кода, приведенных в данной книге.) В версии v2 в инструкции `print` для подавления перехода на новую строку использовался замыкающий символ запятой. В версии v3 используйте для этой цели аргумент `end=" "`, переопределяющий заданное по умолчанию значение `end="\n"`.

Строковые литералы

Когда версия v3 только появилась, литералы наподобие `b'bytestring'` для объектов байтовых строк были несовместимы с версией v2, но в настоящее время для этого синтаксиса предоставлена ретроподдержка, и теперь он доступен в обеих версиях. Точно так же формат `u'Unicode string'` строковых литералов Unicode

в версии v2 не был доступен в ранних реализациях v3, но его включение в версию v3 было рекомендовано в документе **PEP 414** (`https://www.python.org/dev/peps/pep-0414/`) и в настоящее время реализовано, что упрощает использование обоих языков в общем дереве исходного кода без внесения радикальных изменений во все строковые литералы.

Если в вашем коде в основном используются строковые литералы Unicode, но они не обозначены как таковые, используйте инструкцию **`from __future__ import unicode_literals`**, которая вынудит интерпретатор v2 обрабатывать все сомнительные литералы так, как если бы они были снабжены префиксом `u` (точно так же, как это всегда делает интерпретатор v3).

Числовые константы

В версии v2 целочисленные литералы, начинающиеся с `0`, интерпретировались как восьмеричные. Это вводило в заблуждение новичков (а иногда и более опытных программистов). Несмотря на то что в версии v2 такой синтаксис все еще допустим, он считается синтаксической ошибкой в версии v3, поэтому полностью исключите использование данной нотации.

Вместо этого используйте для восьмеричных чисел более новую нотацию, лучше согласующуюся с нотацией для двоичных и шестнадцатеричных чисел. В табл. 26.1 основание системы счисления обозначается буквой, следующей за начальным `0` (`b` — двоичная, `o` — восьмеричная, `x` — шестнадцатеричная).

Таблица 26.1. Предпочтительные представления целочисленных литералов

Система счисления	Представление десятичных чисел 0, 61 и 200
Двоичная	`0b0, 0b111101, 0b11001000`
Восьмеричная	`0o0, 0o75, 0o310`
Шестнадцатеричная	`0x0, 0x3d, 0xc8`

Представление длинных целых чисел в более ранних реализациях версии v2 требовало использования замыкающей буквы `L` или `l`. В интересах обеспечения совместимости этот синтаксис все еще допустим в версии v2. Однако он больше не является необходимым (и считается синтаксической ошибкой в версии v3). Убедитесь в том, что в вашем коде он нигде не используется.

Текстовые и двоичные данные

Пожалуй, для большинства проектов больше всего хлопот доставляет настойчивость версии v3 в проведении различия между двоичными и текстовыми данными посредством использования байтовых строк для первых и строк Unicode для последних. В версии v2 доступны типы `bytes` и `bytearray` (тип `bytes` — псевдоним `str`, а не ретроподдержка). Убедитесь в том, что для всех строковых литералов в версии

v2 используется синтаксис, совместимый с версией (см. раздел "Строковые литералы"), и храните оба типа данных по отдельности.

Распутывание строк

В очень многих программах, ориентированных на версию v2, не уделено должного внимания проведению различия между двумя совершенно разными типами строк: байтовыми и строками Unicode. Везде, где подразумевается текст, не забывайте о правиле "декодировать на входе, кодировать на выходе". Существует множество различных способов представления (кодирования) символов Unicode на внешних устройствах: все кодировки должны декодироваться на входе для преобразования в Unicode, ведь именно должен храниться весь текст в вашей программе.

Избегайте использования типа `bytes` как функции с целочисленным аргументом. В версии v2 она возвращает целое число, преобразованное в (байтовую) строку, поскольку `bytes` — это псевдоним `str`, в то время как в версии v3 она возвращает байтовую строку, содержащую заданное количество нулевых символов. Поэтому, например, вместо используемого в версии v3 выражения `bytes(6)` используйте его эквивалент `b'\x00'*6`, который без проблем работает в обеих версиях.

Та же причина порождает еще одно следствие: в версии v2 индексирование байтовой строки возвращает строку, состоящую из одного байта, в то время как в версии v3 эта операция возвращает целочисленное значение индексируемого байта. Это означает, что некоторые операции сравнения предполагают сравнение байтов с байтами в v2 и байтов с целыми в v3, что может привести к несовместимым результатам. По крайней мере, вы должны тестировать свой код с помощью опции `-bb` командной строки Python, которая обеспечит вывод предупреждений при попытках выполнения подобных сравнений.

Никогда не выполняйте сортировку, используя именованный аргумент `cmp`

Именованный аргумент `cmp` функций и методов сортировки исключен в версии v3, так что не используйте его в своем коде. В конце концов, это не очень удачная идея и в версии v2. Аргумент `cmp` должен был быть функцией, принимающей два значения в качестве аргументов и возвращающей одно из значений `-1`, `0` или `+1` в соответствии с тем, меньше ли первый аргумент, чем второй, равен ему или превышает его. Это может замедлять сортировку, поскольку такая функция должна вызываться для каждой пары элементов, сравниваемых в процессе сортировки, приводя к большому количеству вызовов.

В противоположность этому аргумент `key` принимает единственный сортируемый элемент и возвращает соответствующий ключ сортировки: он вызывается

только один раз для каждого элемента сортируемой последовательности. Алгоритм сортировки создает внутренний кортеж (*key*, *value*) для каждого значения входной последовательности, выполняет сортировку, а затем отбрасывает ключи для получения последовательности, включающей только два элемента. В следующем примере продемонстрирована сортировка списка строк *по длине* (в порядке возрастания длины; для строк одинаковой длины сохраняется порядок их следования во входном списке).

```
>>> names = ['Victoria', 'Brett', 'Luciano', 'Anna', 'Alex', 'Steve']
>>> names.sort(key=len)
>>> names
['Anna', 'Alex', 'Brett', 'Steve', 'Luciano', 'Victoria']
```

Поскольку алгоритм сортировки Python является *устойчивым* (т.е. порядок следования равных элементов остается тем же, что и на входе), имена одной и той же длины не обязательно располагаются в алфавитном порядке (вместо этого для строк с равными значениями ключей сохраняется входной порядок следования). Вот почему имя `'Anna'` предшествует имени `'Alex'` в выводе. Это поведение можно изменить, включив значение в качестве вторичного ключа сортировки, используемого для различения элементов, первичные ключи которых указывают на равенство этих элементов.

```
>>> names.sort(key=lambda x: (len(x), x))
>>> names
['Alex', 'Anna', 'Brett', 'Steve', 'Luciano', 'Victoria']
```

Иногда код может работать быстрее, если полагаться исключительно на устойчивость сортировки. Выполните сортировку списка дважды: сначала в алфавитном порядке (т.е. по вспомогательному ключу сортировки), а затем по длине (основной ключ сортировки). Для списка, приведенного в этом примере, нами было эмпирически установлено, что двойная сортировка работает примерно на 25% быстрее, чем однократная сортировка с использованием лямбда-функции в качестве ключа (не забывайте о том, что отказ от использования лямбда-функций часто способствует оптимизации кода).

Предложения except

Если предложение `except` должно работать со значением исключения, используйте современный синтаксис `except ExceptionType as v`. Прежняя форма `except ExceptionType, v` является синтаксической ошибкой в версии v3 и в любом случае менее удобочитаема в версии v2.

Деление

В процессе подготовки к преобразованию кода убедитесь в том, что все целочисленные операции деления выполняются с использованием оператора `//` (деление

с усечением). В версии v2 результатом деления `12 / 5` будет `2`; в версии v3 значением того же выражения будет `2.4`. Из-за этого различия могут возникать проблемы. В обеих версиях `12 // 5` равно `2`, поэтому лучше всего преобразовать код для использования деления с усечением там, где это уместно.

В тех случаях, когда в результате деления вы действительно хотите получить результат в виде числа с плавающей точкой, в версии v2 вы, вероятно, использовали бы целочисленную константу в виде числа с плавающей точкой, как в выражении `j/3.0`, или применяли функцию `float` к одному из операндов, как в выражении `float(j)/3`. Вы можете воспользоваться этой возможностью для упрощения своего v2-кода путем добавления инструкции `from __future__ import division` в самом начале кода своих модулей для гарантии того, что в версии v2 деление будет работать так же, как в версии v3, позволяя просто записать в коде выражение `j/3`. (Приступайте к упрощению кода этим способом лишь после того, как замените все имеющиеся операторы `/` на операторы `//` везде, где требуется деление с усечением.)

Синтаксис, нарушающий совместимость

В процессе сопровождения двухверсионной кодовой базы чаще всего раздражает то, что приходится заставлять себя отказываться от новых возможностей версии v3. Как правило, соответствующие средства были добавлены в язык для того, чтобы упростить его использование, поэтому отказ от них несколько усложняет задачи программиста.

Наряду с описанными до этого различиями в версии v3 существуют другие специфические возможности, включения которых в код следует избегать, а именно: аннотирование функций (см. раздел "Аннотирование функций и подсказки типов (только в версии v3)" в главе 3); ключевое слово `nonlocal` (см. раздел "Вложенные функции и вложенные области видимости" в главе 3); аргументы, которые могут быть только именованными (см. раздел "Параметры, указываемые только как именованные (версия v3)" в главе 3); и аргумент `metaclass` в инструкциях `class` (см. раздел "Инструкция `class`" в главе 4). Кроме того, в версии v3 генераторы списков, в отличие от версии v2, имеют собственную область видимости (см. раздел "Генераторы списков и области видимости переменных" в главе 3). И наконец, что немаловажно, в версии v2 безусловно следует избегать использования всех возможностей, которые отмечены в данной книге как "Новое в версии 3.6".

Выбор стратегии поддержки

Вы должны самостоятельно принять решение, каким образом будете использовать Python, в зависимости от конкретных требований. Выбор стратегии подвержен различным ограничениям, и каждая стратегия имеет как свои плюсы, так и минусы. Вы должны быть прагматиком, выбирая, какую версию (версии) собираетесь поддерживать и как именно.

Одна из стратегий, которая, вероятно, окажется контрпродуктивной, заключается в том, чтобы параллельно поддерживать оба дерева исходного кода (например, как две отдельные ветви в одном и том же репозитории кода) для двух отдельных версий. Основная команда разработчиков Python поддерживала две отдельные ветви исходного кода CPython, 2.*x* и 3.*x*, в течение многих лет, для чего потребовалось выполнить немалый объем излишней во всех остальных отношениях работы.

Если ваша кодовая база в достаточной степени зрелая и редко подвергается изменениям, то параллельная поддержка двух ветвей может оказаться вполне жизнеспособной стратегией. Но если код активно разрабатывается, то вы почти несомненно придете к выводу, что поддержка двух ветвей слишком обременительна и затратна. Кроме того, вы должны поддерживать два отдельных дистрибутива и убеждаться в том, что пользователи получают версию, которая им нужна. Если вы поступите именно так, будьте готовы к тому, что это потребует от вас дополнительных усилий. Однако вместо этого мы рекомендуем вам выбрать один из двух описанных ниже подходов.

Поддержка только версии v2

Если вам необходимо поддерживать большой объем кода, написанного для версии v2, то мы рискнули бы дать вам один еретический совет: продолжайте использовать v2. В этом случае вы должны быть готовы к тому, что останетесь в стороне от мэйнстрима разработок для Python, но это позволит вам использовать библиотеки, которые не были портированы в версию v3 (учтите, что их число постепенно сокращается). Это также избавит вас от необходимости решения задач, связанных с преобразованием кода.

Такая стратегия определенно лучше приспособлена для программ с ограниченным сроком эксплуатации, а также в средах, использующих локально скомпилированные интерпретаторы. Если вы планируете работать с поддерживаемой платформой Python после 2020 года (когда долгосрочная поддержка версии v2 будет прекращена), то должны опередить события и заблаговременно преобразовать код для использования с версией v3.

Одновременная поддержка версий v2/v3 с преобразованием кода

Если вы ведете разработку на основе единого дерева программного кода, то должны решить, будет ли ваш исходный код использовать версию v2 или v3. Работоспособны обе стратегии, и существуют утилиты, позволяющие преобразовывать код в обоих направлениях, так что у вас есть возможность выбора, какую кодовую базу поддерживать: v2 или v3.

За исключением проектов, включающих небольшое количество файлов, для преобразования кода следует использовать сценарий. Если захотите ускорить этот процесс, преобразуя лишь те файлы, которые были изменены со времени последнего

тестового запуска, то сложность этого сценария возрастет. Преобразование кода по-прежнему будет занимать определенное время, что может замедлить прохождение итераций разработки, если вы решите (а в действительности именно так и следует поступать) каждый раз тестировать обе версии.

Для автоматизации тестирования мультиверсионного кода можно использовать пакет `tox` (`https://tox.readthedocs.io/en/latest/`). С его помощью вы сможете тестировать код в нескольких различных виртуальных окружениях и поддерживать не только версии v2 и v3, но и ряд ранних реализаций, а также Jython и PyPy.

Исходный код v2 с преобразованием в v3

Если у вас большая кодовая база v2, то неплохой отправной точкой для миграции проекта будет доработка исходного кода таким образом, чтобы обеспечить его автоматическое преобразование в v3. Основная команда разработчиков Python проявила дальновидность и включила инструменты автоматической трансляции кода в первый выпуск v3, и эта поддержка продолжается вплоть до настоящего времени в виде утилиты `2to3`. Кроме того, утилита `2to3` предоставляет библиотеку `lib2to3`, содержащую *функции-корректоры* (fixers), которые обрабатывают специфические конструкции, требующие преобразования. Другие утилиты преобразования, которые мы будем обсуждать далее, также используют возможности библиотеки `lib2to3`.

Прежде всего, необходимо собрать всю кодовую базу и подготовить ее для преобразования в Python 3. К счастью, сделать это нетрудно, поскольку основная команда разработчиков Python предусмотрительно предоставила средства, осуществляющие такое преобразование. Например, установка версии Anaconda для Python 3.5 (`https://www.anaconda.com/download/`) позволяет (в среде командной строки, допускающей автозавершение ввода с помощью клавиши <Tab>) ввести **`2to3`** и нажать клавишу <Tab> для получения следующего вывода.

```
(s3cmdpy3) airhead:s3cmd sholden$ 2to3
2to3      2to3-2    2to3-2.7  2to3-3.4  2to3-3.5  2to32.6
```

Не весь этот вывод происходит от Anaconda (в используемой системе список путей включал несколько каталогов Python), как показывает следующий вывод каталогов.

```
airhead:s3cmd sholden$ for f in 2to3 2to3- 2to3-2 2to3-2.7
                                2to3-3.4 2to3-3.5 2to32.6
do
 ls -l $(which $f)
done
lrwxr-xr-x 1 sholden staff 8 Mar 14 21:45 /Users/sholden/
Projects/Anaconda/bin/2to3 -> 2to3-3.5
lrwxr-xr-x 1 sholden admin 36 Oct 24 12:30 /usr/local/bin/
2to3-2 -> ../Cellar/python/2.7.10_2/bin/2to3-2
lrwxr-xr-x 1 sholden admin 38 Oct 24 12:30 /usr/local/bin/
2to3-2.7 -> ../Cellar/python/2.7.10_2/bin/2to3-2.7
lrwxr-xr-x 1 sholden admin 38 Sep 12 2015 /usr/local/bin/
```

```
2to3-3.4 -> ../Cellar/python3/3.4.3_2/bin/2to3-3.4
-rwxr-xr-x 1 sholden staff 121 Mar 14 21:45 /Users/sholden/
Projects/Anaconda/bin/2to3-3.5
lrwxr-xr-x 1 root wheel 73 Aug 6 2015 /usr/bin/
2to32.6 -> ../../System/Library/Frameworks/Python.framework/
Versions/2.6/bin/2to32.6
```

Установка Anaconda предоставляет утилиту `2to3.5` и символическую ссылку из утилиты `2to3` в одном и том же каталоге. Остальная часть вывода происходит от других установок. Если каталог с исполняемым файлом Python указан в начале системного списка путей, целевой установкой по умолчанию является `2to3`: код будет преобразовываться в поддерживаемую ею версию v3.

Исходный код v3 с преобразованием в v2

Если вы пишете код в v3, но хотите поддерживать и v2, используйте утилиту `3to2`, также известную как `lib3to2` (`https://pypi.python.org/pypi/3to2`). Утилита `3to2` использует те же функции-корректоры, что и `lib2to3`, обращая их действие для создания версии вашего кода, обеспечивающей обратную совместимость. Утилита `3to2` либо создает работающий v2-код, либо выводит сообщения об ошибках, описывающие, почему это оказалось невозможным. Некоторые возможности v3 недоступны в v2, и поскольку различия могут быть тонкими, следует тестировать код в обеих версиях.

Преобразование `2to3`: учебное упражнение

В настоящее время наиболее часто встречаются ситуации, когда требуется преобразование кода из v2 в v3. Ниже описана процедура такого преобразования, основанная на практическом опыте. В наши дни не так-то просто найти модули, ориентированные только на версию v2, но все-таки это возможно. Мы остановили свой выбор на пакете `voitto` (`https://github.com/japsu/voitto`), который был выпущен в 2012 году и, похоже, остался почти незамеченным. Как это свойственно многим старым пакетам, документация относительно того, как тестировать этот пакет, отсутствует.

Прежде всего мы клонировали репозиторий и создали виртуальное окружение v2 для тестирования загруженного пакета. Затем мы установили фреймворк тестирования `nose`, установили `voitto` в качестве редактируемого пакета и воспользовались утилитой `nosetests` для выполнения тестов пакета. Многие старые пакеты, поддержка которых прекращена, страдают битовой деградацией, поэтому данный шаг является необходимым. В процессе этого выполнялись следующие команды.

```
$ virtualenv --python=$(which python2) ../venv2
Running virtualenv with interpreter /usr/local/bin/python2
New python executable in ¿
/Users/sholden/Projects/Python/voitto/venv2/bin/python2.7
Also creating executable in ¿
```

```
/Users/sholden/Projects/Python/voitto/venv2/bin/python
Installing setuptools, pip, wheel...done.
$ source ../venv2/bin/activate
(venv2) $ pip install nose
Collecting nose
  Using cached nose-1.3.7-py2-none-any.whl
Installing collected packages: nose
Successfully installed nose-1.3.7
(venv2) $ pip install -e .
Obtaining file:///Users/sholden/Projects/Python/voitto
Installing collected packages: voitto
  Running setup.py develop for voitto
Successfully installed voitto-0.0.1
(venv2) $ nosetests
....
----------------------------------------------------------------------
Ran 4 tests in 0.023s
OK
```

Убедившись в том, что все тесты прошли успешно, мы создали виртуальное окружение Python3 и запустили тесты после повторной установки зависимостей. Трассировочная информация усечена, и представлена лишь ее наиболее важная для нас часть.

```
(venv2) $ deactivate
$ pyvenv ../venv3
$ source ../venv3/bin/activate
(venv3) $ pip install nose
Collecting nose
  Using cached nose-1.3.7-py3-none-any.whl
Installing collected packages: nose
Successfully installed nose-1.3.7
(venv3) airhead:voitto sholden$ pip install -e .
Obtaining file:///Users/sholden/Projects/Python/voitto
Installing collected packages: voitto
  Running setup.py develop for voitto
Successfully installed voitto
(venv3) $ nosetests
EEE
======================================================================
ERROR: Failure: NameError (name 'unicode' is not defined)
----------------------------------------------------------------------
Traceback (most recent call last):
 ...
  File "/Users/sholden/Projects/Python/voitto/tappio/lexer.py",
  line 43, in build_set
    assert type(ch) in (str, unicode)
NameError: name 'unicode' is not defined
```

```
======================================================================
ERROR: Failure: ImportError (No module named 'StringIO')
----------------------------------------------------------------------
Traceback (most recent call last):
 ...
  File "/Users/sholden/Projects/Python/voitto/tests/
        script_import_test.py",
  line 23, in <module>
    from StringIO import StringIO
ImportError: No module named 'StringIO'
======================================================================
ERROR: Failure: SyntaxError (invalid syntax (tappio_test.py,
       line 173))
----------------------------------------------------------------------
Traceback (most recent call last):
 ...
  File "/Users/sholden/Projects/Python/voitto/tests/
        tappio_test.py",
  line 173
    print Parser(lex.lex_string(input)).parse_document()
                ^
SyntaxError: invalid syntax
----------------------------------------------------------------------
Ran 3 tests in 0.014s
FAILED (errors=3)
```

Как нетрудно увидеть, программа нуждается в адаптации к виртуальному окружению v3, поскольку в версии v3 тип `unicode` отсутствует, модуль `StringIO` переименован, а инструкция `print` стала функцией.

Начальный запуск утилиты `2to3`

Утилита `2to3` может работать в нескольких режимах. Наиболее полезный способ получения предварительного представления о том, какие трудности могут возникнуть в процессе преобразования, — просто выполнить утилиту в корневом каталоге проекта. Команда

```
2to3 .
```

сканирует весь код и выводит рекомендованные изменения в форме *разностных листингов* (diffs), сообщая о том, какие файлы нуждаются в изменении. Вот эта сводка.

```
RefactoringTool: Files that need to be modified:
RefactoringTool: ./setup.py
RefactoringTool: ./tappio/__init__.py
RefactoringTool: ./tappio/lexer.py
RefactoringTool: ./tappio/models.py
RefactoringTool: ./tappio/parser.py
RefactoringTool: ./tappio/writer.py
```

```
RefactoringTool: ./tappio/scripts/extract.py
RefactoringTool: ./tappio/scripts/graph.py
RefactoringTool: ./tappio/scripts/indent.py
RefactoringTool: ./tappio/scripts/merge.py
RefactoringTool: ./tappio/scripts/missing_accounts.py
RefactoringTool: ./tappio/scripts/move_entries.py
RefactoringTool: ./tappio/scripts/print_accounts.py
RefactoringTool: ./tappio/scripts/print_earnings.py
RefactoringTool: ./tappio/scripts/renumber.py
RefactoringTool: ./tests/helpers.py
RefactoringTool: ./tests/script_import_test.py
RefactoringTool: ./tests/tappio_test.py
RefactoringTool: ./voitto/__init__.py
RefactoringTool: ./voitto/helpers/io.py
RefactoringTool: ./voitto/helpers/tracer.py
RefactoringTool: Warnings/messages while refactoring:
RefactoringTool: ### In file ./tappio/scripts/missing_accounts.py ###
RefactoringTool: Line 36: You should use a for loop here
```

Мы видим, что изменения рекомендованы примерно для 20 файлов исходного кода. Одна старая пословица гласит: "Зачем лаять самому, если держишь собаку?" Мы можем поручить утилите `2to3` фактически внести рекомендованные изменения, добавив в командной строке флаг `-w`. Обычно утилита `2to3` создает резервную копию исходного файла, заменив в его имени расширение *.py* расширением *.py.bak*. Работа в системе управления версиями делает этот шаг излишним, поэтому мы можем добавить также флаг `-n`, чтобы подавить создание резервных копий:

```
(venv3) $ 2to3 -wn .
```

Ниже описаны изменения, внесенные утилитой `2to3`, без указания имен файлов. Многие другие изменения имеют ту же природу, поэтому мы не предоставляем исчерпывающих комментариев по каждому из них. Как подробнее обсуждается далее, многие из этих изменений могут быть не совсем оптимальными. Утилита `2to3` сфокусирована скорее на получении работающего кода, чем на оптимизации его производительности (вспомните "золотое правило программирования", о котором упоминалось в разделе "Оптимизация" главы 16). Оптимизацию лучше всего выполнять отдельно, когда код полностью преобразован и прошел все тесты. Строке, подлежащей удалению, предшествует знак "минус" (-), а заменяющему коду — знак "плюс" (+).

```
- assert type(ch) in (str, unicode)
+ assert type(ch) in (str, str)
```

Конечно же, в v3 тип `unicode` не определен, и поэтому утилита `2to3` преобразует его в тип `str`, но не распознает, что после такого преобразования необходимость в тестировании принадлежности кортежу отпадает. Поскольку эта инструкция все равно будет работать, отложите оптимизацию на некоторое время.

```
- token = self.token_iterator.next()
+ token = next(self.token_iterator)
```

Были рекомендованы также другие изменения аналогичного характера. Во всех случаях метод `next` из версии v3, в настоящее время переименованный в метод `__next__`, мог быть просто транслитерирован в вызов метода. Однако утилита `2to3` выбрала вариант, позволяющий получить лучший код, совместимый с обеими версиями, используя вместо этого функцию `next`.

```
- for account_number, cents in balances.iteritems():
+ for account_number, cents in balances.items():
```

Были рекомендованы также другие изменения, аналогичные этому. В данном случае преобразованная кодовая база работает в версии v2 несколько иначе, поскольку оригинальный код был рассчитан на использование итератора, а не списка, который (в версии v2) возвращает метод `items`. Хотя это и может увеличить потребление памяти, на результат это не должно повлиять.

```
- map(all_accounts.update, flat_accounts.itervalues())
+ list(map(all_accounts.update, iter(flat_accounts.values())))
```

Данное изменение особенно интересно, поскольку оригинальный код был довольно неоднозначным: это строка, в которой утилита `2to3` рекомендовала использовать цикл `for`. В версии v2 `map` возвращает список, тогда как в версии v3 `map` — итератор. В данном случае не только вызов `itervalues` был заменен вызовом `values` (поскольку в версии v3 метод `values` возвращает итератор; вызов функции `iter` является излишним, хотя и безобидным), но и результат был помещен в вызов функции `list`, чтобы гарантировать получение списка. Может показаться, что применение функции `list` не приносит никакой пользы, однако удаление этого вызова означало бы, что функция `all_accounts.update` применялась бы к счетам только по мере их получения от итератора.

Оригинальный код на самом деле страдает некоторой неопределенностью и неправильным использованием функции `map`. В действительности он должен был бы быть чуть более сложным и намного более понятным, чтобы можно было записать так, как рекомендовала утилита `2to3`.

```
for ac_value in iter(flat_accounts.values()):
    all_accounts.update(ac_value)
```

Однако пока что мы не будем заниматься оптимизацией, поскольку, несмотря на свою неоднозначность, полученный код все же работает.

```
- print "{account_num}: ALL {fmt_filenames}".format(**locals())
+ print("{account_num}: ALL {fmt_filenames}".format(**locals()))
```

Утилита `2to3` гарантирует, что все инструкции `print` будут преобразованы в вызовы функции, и неплохо справляется с внесением этих изменений.

```
- from StringIO import StringIO
+ from io import StringIO
```

В версии v3 функциональность `StringIO` перенесена в библиотеку. Обратите внимание на то, что утилита `2to3` не обязательно выдает код, совместимый с обеими версиями. Она преследует свою основную цель — преобразование кода в v3, хотя в данном случае, как и в предыдущих, версия v2 также будет вполне удовлетворена преобразованным кодом (поскольку в версии v2 и, в частности, в версии 2.7 также возможен доступ к `StringIO`).

```
- assert_true(callable(main))
+ assert_true(isinstance(main, collections.Callable))
```

Это изменение — дань прошлому, так как в ранних реализациях v3 функция `callable` была удалена. Поскольку она была вновь восстановлена в версии v3.2, то в этом преобразовании нет необходимости, но оно безобидно как в версии v2, так и в версии v3.

```
- except ParserError, e:
+ except ParserError as e:
```

Это изменение предоставляет совместимый синтаксис, работающий как в версии v2, так и в версии v3.

Однако преобразование, детализированное в этом начальном запуске утилиты `2to3`, не приводит к работающей программе. Выполнение тестов все еще обнаруживает две ошибки: одну в тесте `lexer_single` и еще одну в тесте `writer_single`. Для обеих ошибок выводятся сообщения о различиях в списках лексем. Вот так выглядит информация о первой ошибке (в вывод вставлены разрывы строк, поскольку длина некоторых строк превышает ширину страницы).

```
...
  File "/Users/sholden/Projects/Python/voitto/tests/tappio_test.py",
       line 54, in lexer_single
    assert_equal(expected_tokens, tokens)
AssertionError: Lists differ: [('integer', '101')] !=
                              [<Token: integer '101'>]

First differing element 0:
('integer', '101')
<Token: integer '101'>

- [('integer', '101')]
+ [<Token: integer '101'>]
```

Аналогично выглядит информация о второй ошибке.

```
...
  File "/Users/sholden/Projects/Python/voitto/tests/
         tappio_test.py", line 188, in writer_single
```

```
    assert_equal(good_tokens, potentially_bad_tokens)
AssertionError: Lists differ:
  [<Tok[1169 chars]ce_close ''>, <Token: brace_close ''>,
   <Token: brace_close ''>] != [<Tok[1169 chars]ce_close ''>,
   <Token: brace_close ''>, <Token: brace_close ''>]

First differing element 0:
<Token: brace_open ''>
<Token: brace_open ''>

Diff is 1384 characters long. Set self.maxDiff to None to see it.
```

В обоих случаях трассировка приводит к процедуре сравнения списков лексем, представляемой в оригинальном коде в виде метода `__cmp__` класса `Token`.

```
def __cmp__(self, other):
    if isinstance(other, Token):
        return cmp((self.token_type, self.value),
                   (other.token_type, other.value))
    elif (isinstance(other, list) or
          isinstance(other, tuple)) and len(other) == 2:
        return cmp((self.token_type, self.value), other)
    else:
        raise TypeError('cannot compare Token to {!r}'.format(
                                                    type(other)))
```

Этот метод не используется в версии v3, которая опирается на методы *расширенного сравнения*. Поскольку метод `__eq__` не был обнаружен, проверка равенства перепоручается методу, унаследованному от объекта `object`, который не отвечает необходимым требованиям. Исправление сводится к замене метода `__cmp__` методом `__eq__`. Этого достаточно для того, чтобы реализовать проверки равенства, требуемые тестами. Новый код выглядит так.

```
def __eq__(self, other):
    if isinstance(other, Token):
        return (self.token_type, self.value) ==
               (other.token_type, other.value)
    elif (isinstance(other, list) or
          isinstance(other, tuple)) and len(other) == 2:
        return (self.token_type, self.value) == tuple(other)
    else:
    raise TypeError('cannot compare Token to {!r}'.format(type(other)))
```

Последнее изменение — это все, что требуется для того, чтобы все тесты были успешно пройдены в версии v3.

```
(ve35) airhead:voitto sholden$ nosetests
....
----------------------------------------------------------------------
Ran 4 tests in 0.028s
OK
```

Сохранение совместимости с версией v2

Если сейчас, когда мы работаем с кодом v3, выполнить тесты в v2, то увидим, что нарушили обратную совместимость и тест `writer_test` теперь не проходит.

```
File "/Users/sholden/Projects/Python/voitto/tappio/writer.py",
        line 48, in write
    self.stream.write(str(token))
TypeError: unicode argument expected, got 'str'
```

Было бы несправедливо жаловаться на тот факт, что преобразование с помощью утилиты `2to3` нарушает обратную совместимость, поскольку обеспечение совместимости не входит в круг ее обязанностей. К счастью, этот код, рассчитанный на версию v3, можно легко сделать совместимым с версией v2. Однако этот процесс, впервые в нашем обсуждении, требует написания кое-какого кода, выполняющего интроспекцию с целью определения конкретной версии Python. Мы обсудим совместимость версий в разделе "Поддержка v2/v3 с использованием единого дерева исходного кода".

Проблема заключается в том, что оригинальный код не помечает явно флагами строковые литералы как Unicode, поэтому в версии v2 они обрабатываются как тип `str`, а не как требуемый тип `unicode`. Выполнение этого преобразования вручную было бы слишком трудоемким, но решение предлагает модуль `__future__`: импорт `unicode_literals`. Если такой импорт имеется в программе для v2, то строковые литералы неявно обрабатываются как строки Unicode, а это именно то, что нам нужно. В результате мы добавляем следующую строку в качестве первой выполняемой строки в каждом модуле:

```
from __future__ import unicode_literals
```

Заметьте, что такой подход не всегда является панацеей, как могло бы показаться (и уж точно не обеспечит немедленное исправление ошибки в данном случае, как вы вскоре увидите). Хотя он и полезен в данном случае, но если программа имеет смешанные двоичные и текстовые строковые данные, то не существует какого-то одного представления, которое служило бы обеим целям, и многие разработчики предпочитают решать эту проблему постепенно. Изменения могут вносить регрессию в Python 2, и на это стоит обращать внимание. Но не только на это: все строки документирования неявно становятся строками Unicode, что в версиях, предшествующих версии 2.7.7, привело бы к синтаксическим ошибкам.

Фактически одно только это изменение *увеличивает* количество неудачных тестов до трех с одной и той же ошибкой во всех случаях.

```
File "/Users/sholden/Projects/Python/voitto/tappio/lexer.py",
      line 45, in build_set
  assert type(ch) in (str, str)
```

Как видите, изменение, внесенное утилитой `2to3` для удаления типа `unicode`, обернулось еще большей неудачей, поскольку, хоть этого и достаточно для версии

v3, в версии v2 необходимо явно разрешать строкам иметь тип unicode. Мы не можем просто обратить это изменение, поскольку данный тип недоступен в v3: попытка сослаться на него приведет к возбуждению исключения NameError.

Что нам необходимо, так это тип, совместимый с обеими версиями. Именно такой тип вводит модуль six (рассмотрен в следующем разделе), но в данном случае, вместо того чтобы идти на издержки в виде целого модуля, мы напишем простой модуль в виде файла *tappio/compat.py*, обеспечивающий требуемую совместимость. Он создает подходящее определение имени unicode для версии, в которой выполняется. Следуя советам руководства по портированию программ, мы используем не запрос информации о версии, а обнаружение возможностей. Файл *compat.py* имеет следующее содержимое.

```
try:
    unicode = unicode
except NameError:
    unicode = str
```

Внимательный читатель мог бы задаться вопросом о том, зачем необходимо связывать имя unicode, а не просто ссылаться на него. Причина заключается в том, что в версии v2 имя unicode справа от оператора присваивания происходит из встроенного пространства имен, и без такого присваивания оно было бы недоступным для импорта из модуля.

Модули *tappio/writer.py* и *tappio/lexer.py* были изменены для импорта unicode из модуля tappio.compat. Выражение str(token) в файле *writer.py* было заменено на unicode(token). Выражение isinstance(ch, str) в файле *lexer.py* — на isinstance(ch, unicode). После внесения этих изменений все тесты проходят как в версии v2, так и в версии v3.

Поддержка v2/v3 с использованием единого дерева исходного кода

Как показало предыдущее упражнение, вы можете написать код на Python, способный выполняться в обоих окружениях. Однако для этого требуется проявить особую внимательность при написании кода и, как правило, добавить код, специально обеспечивающий совместимость программы с обоими окружениями. Официальный документ Python *Porting HOWTO* (https://docs.python.org/3.5/howto/pyporting.html) включает практические советы по обеспечению межверсионной совместимости, охватывая гораздо более широкий круг вопросов, чем рассмотренный нами.

Для поддержки разработки в рамках единой кодовой базы был разработан ряд библиотек, включая six, python-modernize и python-future. Последние две библиотеки позволяют исправлять v2-код и добавлять соответствующие операции импорта из __future__ для получения необходимых частей функциональности версии v3.

Библиотека six

Библиотека six была ранним проектом, предназначенным для обеспечения подобной совместимости, но некоторые считают, что она слишком громоздкая, а удобство в работе также оставляет желать лучшего. Однако она была успешно использована в таких крупных проектах, как Django, для обеспечения межверсионной совместимости, поэтому можете принять на веру, что данная библиотека способна сделать все, что вам нужно.

Библиотека six представляет особую ценность в ситуациях, когда необходимо сохранить совместимость с ранними реализациями версии v2, поскольку она создавалась в то время, когда текущей была версия 2.6, и поэтому учитывала дополнительные проблемы, связанные с совместимостью с Python 2.6. Однако мы рекомендуем вам ориентироваться только на обеспечении совместимости с версией Python 2.7, а не с более ранними версиями: эта задача и так непростая, поэтому не обременяйте себя дополнительными сложностями.

Библиотека python-modernize

Библиотека python-modernize появилась в то время, когда текущей была версия Python 3.3. Данная библиотека обеспечивает неплохую совместимость между версиями 2.6, 2.7 и 3.3, предоставляя тонкую оболочку вокруг утилиты 2to3, и работает в основном так же. Описание порядка работы с библиотекой и ряд ценных советов по портированию, которые будут полезны всем, кто заинтересован в коде, совместимом с двумя версиями Python, вы найдете в статье *Porting to Python Redux* (http://lucumr.pocoo.org/2013/5/21/porting-to-python-3-redux/) Армина Ронахера, автора самой библиотеки.

Библиотека python-modernize использует библиотеку six в качестве основы и работает во многом аналогично инструменту 2to3, читая код и записывая его версию, которую легче поддерживать в качестве объединенной кодовой базы, поддерживающей две версии Python. Установите ее в виртуальном окружении с помощью команды pip install modernize. Ее вывод стилистически сместит код в направлении версии v3.

Библиотека python-future

Самый современный из инструментов, предназначенных для поддержки единой кодовой базы, совместимой с обеими версиями, — библиотека python-future (python-future.org). Установите ее в виртуальном окружении с помощью команды **pip install future**. Если вы (как мы рекомендовали) поддерживаете только версии v2 (Python 2.7) и v3 (Python 3.5 и более новые), то она работает автономно без внешних зависимостей. Используя модули importlib, unittest2 и argparse, она может расширить поддержку до версии 2.6, если вам это действительно необходимо.

Трансляция кода осуществляется в два этапа: на первом этапе используются лишь "безопасные" функции-корректоры, вероятность разрушения кода которыми чрезвычайно мала и которые не пытаются использовать какие-либо функции

совместимости из пакета `future`. Вывод на этом этапе все еще представляет собой код v2 с вызовами стандартных функций v2. Целью двухэтапной организации этого процесса является внесение изменений, которые заведомо не приведут к возникновению конфликтов, чтобы на втором этапе можно было сосредоточиться на изменениях, для которых существует риск разрушения кода.

Изменения, вносимые на втором этапе, добавляют зависимости от будущих пакетов и преобразуют исходный код в версию для v3, способную выполняться также в версии v2 за счет использования зависимостей. Пользователи v2 могут обнаружить, что стиль преобразованного вывода очень похож на стиль кода, к которому они привыкли.

Пакет `future` поставляется с командой `pasteurize`, предназначенной для преобразования кода v3 в код v2. Однако мы обнаружили, что в режиме этой команды, используемом по умолчанию, ей не удалось внести изменения для обеспечения совместимости с версией v2, которые обсуждались в конце раздела "Сохранение совместимости с версией v2".

Поддержка только версии v3

Для всех, кто создает новые Python-проекты, не включающие никаких зависимостей, которые содержатся только в библиотеках v2, наиболее предпочтительной будет следующая стратегия: пишите код для v3 и никогда не оглядывайтесь назад! Конечно, если вы создаете библиотеки или фреймворки, предназначенные для использования в обеих версиях, то не можете позволить себе этой роскоши — вы обязаны обеспечить совместимость, которую требуют от вас клиенты.

Не забывайте о том, что совсем скоро официальная поддержка версии v2 прекратится, поскольку стратегия, ориентированная на написание кода, поддерживающего только версию v3, с каждым месяцем укрепляет свои позиции. Ежедневно выпускается все больше и больше кода для v3, и со временем новый код будет доминировать. Можно ожидать, что еще большее количество проектов откажется от поддержки v2, о чем уже заявили команды разработчиков многих значительных научных проектов (`python3statement.org`). Поэтому, если вы решите создавать только код для v3, с течением времени количество жалоб, скорее всего, будет уменьшаться, а не увеличиваться. Не отвлекайтесь и снизьте себе нагрузку по сопровождению кода!

Предметный указатель

К

П

Р

С

СТАНДАРТНАЯ БИБЛИОТЕКА PYTHON 3

СПРАВОЧНИК С ПРИМЕРАМИ 2-Е ИЗДАНИЕ

Даг Хеллман

www.dialektika.com

В этой книге Даг Хеллман, эксперт по языку Python, описывает все основные разделы библиотеки Python 3.x, сопровождая изложение материала компактными примерами исходного кода и результатами их выполнения. Приведенные примеры наглядно демонстрируют возможности всех модулей, предлагаемых библиотекой. Каждому модулю посвящен отдельный раздел, содержащий ссылки на дополнительные ресурсы, что делает эту книгу идеальным учебным и справочным руководством. Основные темы книги:

- манипулирование текстом с помощью модулей **string**, **textwrap**, **re** (регулярные выражения) и **difflib**;
- использование структур данных: модули **enum**, **collections**, **array**, **heapq**, **queue**, **struct**, **copy** и множество других;
- элегантная и компактная реализация алгоритмов с использованием модулей **functools**, **itertools** и **contextlib**;
- обработка значений даты и времени и решение сложных математических задач;
- архивирование и сжатие данных.

ISBN: 978-5-6040043-8-8

в продаже

АВТОМАТИЗАЦИЯ РУТИННЫХ ЗАДАЧ С ПОМОЩЬЮ PYTHON

практическое руководство для начинающих

Эл Свейгарт

www.williamspublishing.com

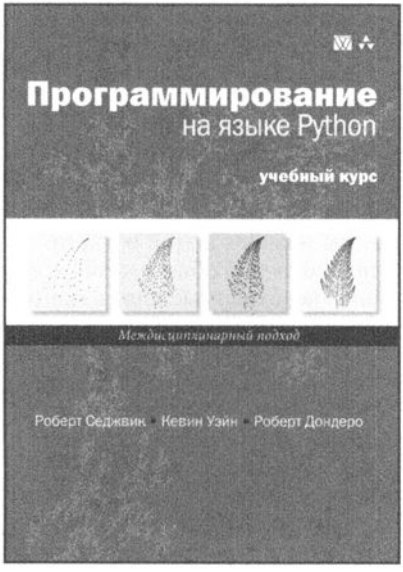
Книга научит вас использовать Python для написания программ, способных в считанные секунды сделать то, на что раньше у вас уходили часы ручного труда, причем никакого опыта программирования от вас не требуется. Как только вы овладеете основами программирования, вы сможете создавать программы на языке Python, которые будут без труда выполнять в автоматическом режиме различные полезные задачи, такие как:

- поиск определенного текста в файле или в множестве файлов;
- создание, обновление, перемещение и переименование файлов и папок;
- поиск в Интернете и загрузка онлайн-контента;
- обновление и форматирование данных в электронных таблицах Excel любого размера;
- разбиение, слияние, разметка водяными знаками и шифрование PDF-документов;
- рассылка напоминаний в виде сообщений электронной почты или текстовых уведомлений;
- заполнение онлайновых форм.

ISBN 978-5-8459-2090-4 **в продаже**

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PYTHON
учебный курс

Роберт Седжвик
Кевин Уэйн
Роберт Дондеро



www.dialektika.com

Авторы книги сосредоточиваются на самых полезных и важных средствах языка Python и не стремятся к его абсолютно полному охвату. Весь код этой книги был отработан и проверен на совместимость как с языком Python 2, так и Python 3, что делает его подходящим для каждого программиста и любого курса на много лет вперед.

Особенности книги:

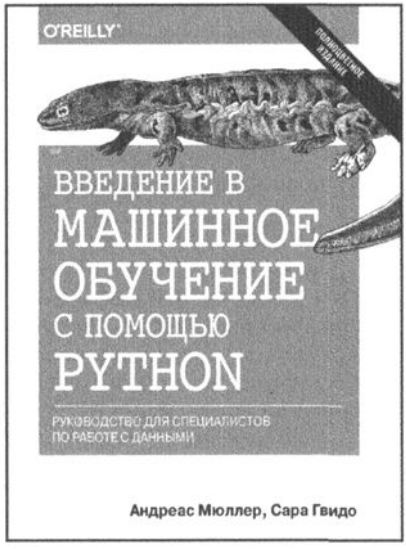
- всеобъемлющий, основанный на приложениях подход: изучение языка Python на примерах из области науки, математики, техники и коммерческой деятельности;
- основное внимание главному: самым полезным и важным средствам языка Python;
- совместимость примеров кода проверена на языках Python 2.x и Python 3.x;
- во все главы включены разделы с вопросами и ответами, упражнениями и практическими упражнениями.

ISBN 978-5-9908462-1-0 **в продаже**

ВВЕДЕНИЕ В МАШИННОЕ ОБУЧЕНИЕ С ПОМОЩЬЮ PYTHON

РУКОВОДСТВО ДЛЯ СПЕЦИАЛИСТОВ ПО РАБОТЕ С ДАННЫМИ

Андреас Мюллер
Сара Гвидо



www.williamspublishing.com

Эта полноцветная книга — отличный источник информации для каждого, кто собирается использовать машинное обучение на практике. Ныне машинное обучение стало неотъемлемой частью различных коммерческих и исследовательских проектов, и не следует думать, что эта область — прерогатива исключительно крупных компаний с мощными командами аналитиков. Эта книга научит вас практическим способам построения систем МО, даже если вы еще новичок в этой области. В ней подробно объясняются все этапы, необходимые для создания успешного проекта машинного обучения, с использованием языка Python и библиотек scikit-learn, NumPy и matplotlib. Авторы сосредоточили свое внимание исключительно на практических аспектах применения алгоритмов машинного обучения, оставив за рамками книги их математическое обоснование. Данная книга адресована специалистам, решающим реальные задачи, а поскольку область применения методов МО практически безгранична, прочитав эту книгу, вы сможете собственными силами построить действующую систему машинного обучения в любой научной или коммерческой сфере.

ISBN 978-5-9908910-8-1

в продаже

O'REILLY®

Python. Справочник

Python — один из наиболее популярных современных языков программирования. Третье издание этого практического руководства представляет собой исчерпывающий справочник, содержащий описание большинства модулей обширной стандартной библиотеки Python и наиболее часто используемых модулей сторонних производителей. Справочник ориентирован на версии Python 2.7 и 3.5, но также включает информацию об изменениях и новшествах, появившихся в версии Python 3.6.

Книга охватывает чрезвычайно широкий спектр областей применения Python, включая веб-приложения, сетевое программирование, обработку XML-документов, взаимодействие с базами данных и высокоскоростные вычисления. Она станет идеальным подспорьем как для тех, кто решил изучить Python, имея предварительный опыт программирования на других языках, так и для тех, кто уже использует этот язык в своих разработках.

Основные темы книги:

- Синтаксис Python, объектно-ориентированные возможности языка, модули стандартной библиотеки и пакеты расширений
- Операции с файлами, работа с текстом, базы данных, многозадачность и обработка числовых данных
- Основы работы с сетями, управляемые событиями программы и клиентские модули сетевых протоколов
- Модули расширения Python, средства пакетирования и распространения расширений, модулей и приложений

"Это не только *полное* описание языка (если что-то есть в Python, то оно рассмотрено в книге), но и *понятное*: авторы четко объясняют, почему тот или иной элемент включен в Python и как правильно организовать взаимодействие различных элементов".

Питер Норвиг
директор по исследованиям
в корпорации Google

Алекс Мартелли — инженер компании Google, активно публикуется на сайте Stack Overflow, часто выступает с докладами на технических конференциях, номинированный член организации PSF, обладатель награды *2006 Frank Willison Memorial Award* за большой вклад в сообщество Python.

Анна Рейвенскрофт — опытный лектор и специалист по обучению персонала, энтузиаст Python, номинированный член организации PSF, обладатель награды *2013 Frank Willison Memorial Award*.

Стив Холден — технический директор компании Felix, номинированный член организации PSF, директором и председателем правления которой является, обладатель награды *2007 Frank Willison Memorial Award*.

Категория: программирование/Python
Уровень: для пользователей средней и высокой квалификаций

ISBN: 978-5-6040723-8-7

9 785604 072387 18037

Диалектика

www.williamspublishing.com