МАРК САММЕРФИЛД

ПРОГРАММИРОВАНИЕ НА Python 3

ПОДРОБНОЕ РУКОВОДСТВО

HIGHTECH

Символ®

Programming in Python 3

A Complete Introduction to the Python Language

Mark Summerfield

Addison-Wesley

HIGH TECH

Программирование на Python 3

Подробное руководство

Марк Саммерфилд

Символ®

Санкт-Петербург — Москва
2009

Серия «High tech»

Марк Саммерфилд

Программирование на Python 3
Подробное руководство

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Ю. Бочина*
Корректор	*С. Николаева*
Верстка	*Д. Орлова*

Саммерфилд М.

Программирование на Python 3. Подробное руководство. – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 608 с., ил.
ISBN: 978-5-93286-161-5

Третья версия языка Python сделала его еще более мощным, удобным, логичным и выразительным. Книга «Программирование на Python 3» написана одним из ведущих специалистов по этому языку, обладающим многолетним опытом работы с ним. Издание содержит все необходимое для практического освоения языка: написания любых программ с использованием как стандартной библиотеки, так и сторонних библиотек для языка Python 3, а также создания собственных библиотечных модулей.

Автор начинает с описания ключевых элементов Python, знание которых необходимо в качестве базовых понятий. Затем обсуждаются более сложные темы, поданные так, чтобы читатель мог постепенно наращивать свой опыт: распределение вычислительной нагрузки между несколькими процессами и потоками, использование сложных типов данных, управляющих структур и функций, создание приложений для работы с базами данных SQL и с файлами DBM.

Книга может служить как учебником, так и справочником. Текст сопровождается многочисленными примерами, доступными на специальном сайте издания. Весь код примеров был протестирован с окончательным релизом Python 3 в ОС Windows, Linux и Mac OS X.

ISBN: 978-5-93286-161-5
ISBN: 978-0-13-712929-4 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 29.04.2009. Формат 70x100$^{1}/_{16}$. Печать офсетная.
Объем 38 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Памяти Франко Рабайотти
(Franco Rabaiotti)
1961–2001

Оглавление

Введение

Язык Python является, пожалуй, самым простым в изучении и самым приятным в использовании из языков программирования, получивших широкое распространение. Программный код на языке Python легко читать и писать, и, будучи лаконичным, он не выглядит загадочным. Python – очень выразительный язык, позволяющий уместить приложение в меньшее количество строк, чем на это потребовалось бы в других языках, таких как C++ или Java.

Python является кросс-платформенным языком: обычно одна и та же программа на языке Python может запускаться и в Windows, и в UNIX-подобных системах, таких как Linux, BSD и Mac OS, для чего достаточно просто скопировать файл или файлы, составляющие программу, на нужный компьютер; при этом даже не потребуется выполнять «сборку», или компилирование программы. Конечно, можно написать на языке Python программу, которая будет использовать некоторые характерные особенности конкретной операционной системы, но такая необходимость возникает крайне редко, т. к. практически вся стандартная библиотека языка Python и большинство библиотек сторонних производителей обеспечивают полную кросс-платформенность.

Одним из основных преимуществ языка Python является наличие полной стандартной библиотеки, позволяющей обеспечить загрузку файла из Интернета, распаковку архива или создание веб-сервера посредством написания нескольких строк программного кода. В дополнение к ней существуют тысячи дополнительных библиотек сторонних производителей, среди которых одни обеспечивают более сложные и более мощные возможности, чем стандартная – например, библиотека для организации сетевых взаимодействий Twisted и библиотека для решения вычислительных задач NumPy; а другие предоставляют функциональность, которая слишком узконаправленно специализирована, чтобы ее можно было включить в стандартную библиотеку – например, пакет моделирования SimPy. Большинство сторонних библиотек можно найти на сайте Python Package Index (каталог пакетов Python): *pypi.python.org/pypi*.

Python может использоваться для программирования в процедурном, в объектно-ориентированном и, в меньшей степени, в функциональном стиле программирования, хотя в глубине души Python – объектно-ориентированный язык программирования. Эта книга покажет,

как писать процедурные и объектно-ориентированные программы, а также расскажет об особенностях функционального программирования на языке Python.

Цель этой книги – показать вам, как писать программы на языке Python в стиле Python 3, а после прочтения – служить хорошим справочником по этому языку. Несмотря на то, что версия Python 3 является эволюционным, а не революционным усовершенствованием Python 2, тем не менее в Python 3 некоторые прежние приемы программирования стали неприменимы или в них отпала необходимость. При этом появились некоторые новые приемы, позволяющие использовать преимущества особенностей новой версии. Язык Python 3 более совершенен, чем Python 2 – он основан на опыте многих лет использования этого языка и привносит множество новых особенностей (и ликвидирует недостатки Python 2), делая его еще более приятным в использовании, более удобным, более простым и более последовательным.

Цель книги – научить *языку* Python, при этом в книге используются многие стандартные библиотеки, но далеко не все. Впрочем, это не проблема, так как после прочтения этой книги вы будете обладать объемом знаний, достаточным, чтобы суметь воспользоваться любыми стандартными или сторонними библиотеками языка Python и даже создавать свои собственные библиотечные модули.

Как предполагается, книга будет полезна различным группам читателей, включая тех, для кого программирование – это хобби, а также студентов, научных работников, инженеров и всех тех, для кого программирование является подручным средством в их работе, и, конечно же, – профессиональных программистов. Чтобы быть полезной такому широкому кругу читателей – не быть скучной для хорошо подготовленных специалистов и одновременно оставаться доступной для менее подготовленных читателей, книга предполагает наличие у читателя некоторого опыта программирования (на любом языке). В частности, предполагается наличие основных представлений о типах данных (таких как числа и строки), коллекциях данных (таких как множества и списки), управляющих структурах (таких как инструкции `if` и `while`) и функциях. Кроме того, некоторые примеры предполагают знание основ языка разметки HTML, а некоторые, более специализированные главы предполагают наличие базовых знаний по обсуждаемым темам, например, глава о базах данных предполагает знакомство с языком SQL.

Книга структурирована таким образом, чтобы вы могли быстро двигаться вперед. К концу первой главы вы уже сможете писать небольшие, но полезные программы на языке Python. Каждая последующая глава вводит новые темы и часто расширяет и углубляет темы, введенные в предыдущих главах. Это означает, что если вы читаете главы последовательно, вы сможете остановиться в любой момент и написать законченную программу на основе знаний, полученных к этому мо-

менту, после чего продолжить чтение и узнать о существовании более совершенных и более сложных приемов. По этой причине знакомство с некоторыми темами происходит в одной главе, а более глубокое их исследование – в последующих главах.

При изучении нового языка программирования обычно возникают две основные проблемы. Первая заключается в том, что иногда для объяснения одной концепции необходимо знать другую концепцию, которая, в свою очередь, прямо или косвенно опирается на первую. Вторая проблема состоит в том, что некоторые читатели изначально могут ничего не знать о данном языке программирования, и из-за этого бывает очень трудно подыскать интересные или полезные примеры. В этой книге предпринята попытка решить обе проблемы, во-первых, предположив наличие у читателя некоторого опыта программирования, и, во-вторых, представив в главе 1 «золотой запас» языка Python – восемь основных составляющих языка, знания которых вполне достаточно, чтобы начать самостоятельно писать программы. Вследствие принятого подхода некоторые примеры в первых главах отличаются определенной долей искусственности, так как в них используется только то, о чем до этого уже говорилось в книге. От главы к главе этот эффект уменьшается и полностью исчезает к главе 7; все последующие примеры полностью написаны в характерном для языка Python 3 стиле.

Книга опирается на практический подход в обучении и предлагает вам самостоятельно опробовать примеры и упражнения, чтобы приобрести практический опыт. Везде, где только возможно, в качестве примеров приводятся небольшие законченные программы, способные решать вполне реальные задачи. Все примеры и решения упражнений можно найти в Интернете по адресу *www.qtrac.eu/py3book.html* – каждый из них протестирован с использованием Python 3 в операционных системах Windows, Linux и Mac OS X.

Структура книги

В главе 1 будут представлены 8 составляющих языка Python, знания которых будет вполне достаточно для написания законченных программ. Здесь также описываются некоторые среды программирования на языке Python и приводятся два примера маленьких программ, в каждой из которых используются восемь составляющих языка Python, описанные ранее в этой главе.

В главах со 2 по 5 вводятся средства процедурного программирования на языке Python, включая базовые типы данных и коллекции данных, а также множество полезных встроенных функций и структур управления наряду с описанием простейших приемов обработки текстовых файлов. В главе 5 рассказывается, как создавать собственные модули и пакеты, и проводится общий обзор стандартной библиотеки Python, чтобы вы получили представление о том, что может предложить Python

«из коробки», и могли избежать необходимости заново изобретать колесо.

Глава 6 представляет собой полное введение в объектно-ориентированное программирование на языке Python. Все сведения о процедурном программировании, которые были получены в предыдущих главах, по-прежнему применимы, потому что объектно-ориентированное программирование имеет процедурную основу: например, в объектно-ориентированном программировании используются те же самые типы данных, коллекции данных и управляющие структуры.

Глава 7 охватывает темы записи в файлы и чтения из файлов. Для двоичных файлов среди всего прочего рассматривается применение операций сжатия и произвольного доступа к содержимому файлов, а для текстовых файлов – синтаксический анализ вручную и с применением регулярных выражений. Кроме того, в этой главе будет показано, как писать и читать файлы в формате XML, включая использование элементов деревьев, DOM (Document Object Model – объектная модель документа) и SAX (Simple API for XML – простой прикладной программный интерфейс для работы с XML).

В главе 8 повторно рассматривается материал, представленный в некоторых предыдущих главах, и исследуются многие современные особенности языка Python в таких областях, как типы данных, коллекции данных, управляющие структуры, функции и объектно-ориентированное программирование. В этой главе также будет представлено множество новых функций, классов и усовершенствованных приемов, включая функциональное программирование – материал, который будет и полезен, и необходим.

В остальной части книги рассматриваются более сложные темы. В главе 9 демонстрируются приемы распределения рабочей нагрузки между несколькими процессами и потоками выполнения. Глава 10 рассказывает, как создавать клиент-серверные приложения, используя стандартную поддержку сетевых взаимодействий языка Python. Глава 11 охватывает вопросы программирования приложений для работы с базами данных (как с простыми файлами «DBM», хранящими пары ключ-значение, так и с базами данных SQL). Глава 12 описывает и демонстрирует применение мини-языка регулярных выражений в Python и охватывает модуль регулярных выражений, а в главе 13 рассматриваются вопросы программирования GUI (Graphical User Interface – графический интерфейс пользователя).

Большинство глав в книге достаточно объемны, объединяя взаимосвязанные сведения в одном месте для упрощения их поиска в будущем. Кроме того, главы разбиты на разделы, подразделы, а иногда и подподразделы, что позволяет читать их в том темпе, который устроит вас больше всего, – например, по одному разделу или подразделу за раз.

Получение и установка Python 3

Если в вашем распоряжении имеется современная операционная система Mac или другая UNIX-подобная система, то вполне возможно, что Python 3 у вас уже установлен. Проверить это можно, введя команду `python -V` (обратите внимание, что символ *V* вводится в верхнем регистре) в консоли (`Terminal.app` – в Mac OS X). Если будет выведен номер версии 3, значит Python 3 у вас уже имеется и выполнять его установку не требуется, в противном случае вам следует продолжить чтение этого раздела.

Для операционных систем Windows и Mac OS X существуют простые в использовании установочные пакеты с графическим интерфейсом, которые проведут вас через все этапы процесса установки. Эти пакеты доступны по адресу *www.python.org/download*. Для Windows по указанному адресу имеется три различных инсталлятора; загружайте тот, который подписан «Windows installer», если заранее вам точно не известно, что ваша машина оснащена процессором AMD64 или Itanium; в противном случае загружайте версию для своего процессора. Как только установочный пакет будет загружен, просто запустите его и следуйте инструкциям, появляющимся на экране.

Для Linux, BSD и других версий UNIX самый простой способ установить Python состоит в том, чтобы воспользоваться имеющейся системой управления пакетами. В большинстве случаев Python поставляется в виде отдельных пакетов. Например, для Fedora Python поставляется в виде пакета `python`, а IDLE (простая среда разработки) – в виде пакета `python-tools`; при этом будьте внимательны: эти пакеты содержат Python 3, только если ваша версия Fedora достаточно свежая (версии 10 или выше). Аналогично в дистрибутивах на основе Debian, таких как Ubuntu, эти пакеты называются `python3` и `idle3` соответственно.

Если для вашей операционной системы пакеты с Python 3 отсутствуют, то вам потребуется загрузить исходные тексты по адресу *www.python.org/download* и собрать Python 3 из исходных текстов. Загрузите тарболл с исходными текстами и распакуйте его либо командой `tar xvfz Python-3.0.tgz`, если тарболл сжат архиватором gzip, либо командой `tar xvfj Python-3.0.tar.bz2`, если тарболл сжат архиватором bzip2. Конфигурирование и сборка выполняются стандартным способом. Сначала перейдите во вновь созданный каталог Python-3.0 с распакованными исходными текстами и запустите команду `./configure`. (Если вы хотите выполнить установку в свой локальный каталог, можно определить значение параметра `--prefix`.) Затем запустите команду `make`.

Вполне возможно, что в конце сборки вы получите сообщение о том, что не все модули удалось собрать. Обычно это означает, что у вас отсутствуют необходимые для этого библиотеки или заголовочные файлы. Например, если не удалось собрать модуль `readline`, воспользуйтесь системой управления пакетами и установите соответствующую

библиотеку для разработки `-readline-devel` в системах на базе Fedora или `readline-dev` в системах на базе Debian. (К сожалению, названия пакетов не всегда настолько очевидны.) После установки недостающих пакетов запустите команды `./configure` и `make` еще раз.

После успешной сборки можно попробовать запустить команду `make test`, чтобы убедиться, что все в порядке, хотя это не обязательно и к тому же может занять продолжительное время.

Если вы использовали параметр `--prefix`, то, чтобы выполнить локальную установку, просто запустите команду `make install`. Вероятно, будет желательно добавить символическую ссылку на выполняемый файл `python` (например, `ln -s ~/local/python3/bin/python3.0 ~/bin/python3`, если предполагать, что использовался параметр `--prefix=$HOME/local/python3` и каталог `$HOME/bin` присутствует в переменной окружения `PATH`). Возможно, также будет удобно добавить символическую ссылку на IDLE (например, `ln -s ~/local/python3/bin/idle ~/bin/idle3`, если исходить из тех же предположений, что и выше).

Если вы не использовали параметр `--prefix` и обладаете правами пользователя root, зарегистрируйтесь в системе как root и выполните команду `make install`. В системах с настроенной программой `sudo`, таких как Ubuntu, выполните команду `sudo make install`. Если в системе был установлен Python 2, файл `/usr/bin/python` не изменится и Python 3 будет доступен как `python3`, точно также среда разработки IDLE для Python 3 будет доступна как `idle3`.

Благодарности

Мои первые слова благодарности я хотел бы принести техническим рецензентам книги, начиная с Жасмин Бланшетт (Jasmin Blanchette), программиста, в соавторстве с которой я участвовал в создании двух книг о C++/Qt. Ее рекомендации о том, как расположить содержимое книги по главам, критические замечания по всем примерам и тщательное чтение текста в значительной степени способствовали улучшению книги.

Джордж Брэндл (Georg Brandl) – ведущий разработчик Python и ответственный за создание документации к новому комплекту инструментов Python. Джордж обнаружил множество мелких ошибок и упорно, очень терпеливо объяснял их, пока все они не были поняты и устранены. Он также внес множество усовершенствований в примеры.

Фил Томпсон (Phil Thompson) – эксперт по языку Python и создатель PyQt, одной из лучших библиотек создания графического интерфейса для языка Python. Проницательность и отзывы Фила позволили прояснить и исправить множество неточностей.

Трентон Шульц (Trenton Schulz) – старший инженер-программист подразделения Qt Software в компании Nokia (которое ранее было са-

мостоятельной компанией Trolltech), бывший ценным рецензентом всех моих предыдущих книг, вновь пришел мне на помощь. Внимательное отношение Трентона и множество внесенных им предложений помогли выявить ряд неувязок и способствовали значительному улучшению этой книги.

В дополнение к вышеупомянутым рецензентам, каждый из которых прочитал книгу целиком, я хотел бы поблагодарить Дэвида Бодди (David Boddie), старшего технического писателя подразделения Qt Software в компании Nokia, опытного практика языка Python, который прочитал некоторые части книги и дал ценные рекомендации по ним.

Спасибо также Гвидо ван Россуму (Guido van Rossum), создателю языка Python, а также обширному сообществу пользователей Python, приложившим невероятные усилия, чтобы сделать язык Python и в особенности, его библиотеки такими полезными и удобными в использовании.

Как всегда, спасибо Джеффу Кингстону (Jeff Kingston), создателю языка верстки Lout, который я использую уже более десяти лет.

Особое спасибо моему редактору Дебре Вильямс Коли (Debra Williams Cauley) за ее поддержку и за то, что максимально обеспечивала плавность протекания всего процесса работы над книгой. Спасибо также Анне Попик (Anna Popick), которая так здорово справлялась с управлением производственным процессом, и корректору Одри Дойл (Audrey Doyle), прекрасно выполнившему свою работу.

И напоследок, но не в последнюю очередь, я хочу поблагодарить мою супругу Андреа (Andrea) за то, что терпела мои пробуждения в 4 часа утра, когда часто появлялись новые идеи и вспоминались неточности в программном коде, требующие немедленной проверки и исправления, а также за ее любовь, верность и поддержку.

- Создание и запуск программ на языке Python
- «Золотой запас» Python

1

Быстрое введение в процедурное программирование

В этой главе приводится объем информации, достаточный, чтобы начать писать программы на языке Python. Мы настоятельно рекомендуем установить Python, если вы еще не сделали этого, чтобы иметь возможность получить практический опыт для закрепления изучаемого материала. (Во введении описывается, как получить и установить Python во всех основных платформах, подробности на стр. 17.)

Первый раздел этой главы показывает, как создавать и запускать программы на языке Python. Для написания программного кода вы можете использовать любой простой текстовый редактор по своему усмотрению, при этом среда разработки IDLE, обсуждаемая в этом разделе, предоставляет не только редактор программного кода, но и дополнительные функциональные возможности, включая средства отладки и проведения экспериментов с программным кодом Python.

Во втором разделе будут представлены восемь составляющих языка Python, знания которых вполне достаточно для создания программ, имеющих практическую ценность. Все эти составляющие будут рассмотрены глубже в последующих главах и по мере продвижения вперед будут дополняться остальными особенностями языка Python, чтобы к концу книги весь язык оказался полностью охваченным, а вы были в состоянии использовать в своих программах все, что он предлагает.

В заключительной части главы представлены две короткие программы, в которых используется подмножество особенностей языка Python, введенных во втором разделе, благодаря чему вы сможете получить представление о программировании на языке Python.

Создание и запуск программ на языке Python

Кодировки символов, стр. 112

Программный код на языке Python можно записать с помощью любого простого текстового редактора, который способен загружать и сохранять текст либо в кодировке ASCII, либо UTF-8. По умолчанию предполагается, что файлы с программным кодом на языке Python сохраняются в кодировке UTF-8, надмножестве кодировки ASCII, с помощью которой можно представить практически любой символ любого национального алфавита. Файлы с программным кодом на языке Python обычно имеют расширение *.py*, хотя в некоторых UNIX-подобных системах (таких как Linux и Mac OS X) некоторые приложения на языке Python не имеют расширения, а программы на языке Python с графическим интерфейсом, в частности в Windows и Mac OS X, обычно имеют расширение *.pyw*. В этой книге все время будет использоваться расширение *.py* для обозначения консольных программ и модулей Python и расширение *.pyw* – для программ с графическим интерфейсом. Все примеры, представленные в книге, не требуют изменений для запуска в любой из платформ, поддерживаемых Python 3.

Ради того чтобы удостовериться, что установка выполнена корректно, и чтобы продемонстрировать классический первый пример, создадим файл с именем *hello.py* в простом текстовом редакторе (в Windows для этих целей вполне подойдет «Блокнот» (Notepad), а кроме того вскоре будет рассказано о более удобном редакторе) со следующим содержимым:

```
#!/usr/bin/env python3

print("Hello", "World!")
```

Первая строка – это комментарий. В языке Python комментарии начинаются с символа `#` и продолжаются до конца строки. (Через минуту мы объясним назначение этого странного комментария.) Вторая строка – пустая. Python игнорирует пустые строки, но их бывает резонно вставлять, разбивая протяженные блоки программного кода на более мелкие части, более удобные для восприятия. Третья строка – это программный код на языке Python. В данном случае вызывается функция `print()`, которой передается два аргумента, оба – типа `str` (`string`, или строка, то есть последовательность символов).

Все инструкции, которые встретятся в файле с расширением *.py*, выполняются последовательно, строка за строкой, начиная с первой строки. В этом заключается отличие от других языков программирования, таких как C++ или Java, в которых имеются функции или методы со специальными именами, которые запускаются в первую оче-

редь. Порядок выполнения инструкций может быть изменен, в чем можно будет убедиться при обсуждении управляющих структур языка Python в следующем разделе.

Будем считать, что пользователи Windows сохраняют файлы с программным кодом Python в каталоге *C:\py3eg*, а пользователи UNIX (то есть UNIX, Linux и Mac OS X) – в каталоге *$HOME/py3eg*. Сохраните файл *hello.py* в каталоге *py3eg* и закройте текстовый редактор.

Теперь, когда у нас имеется программа, ее можно запустить. Программы на языке Python выполняются интерпретатором Python, и, как правило, делается это в окне консоли. В операционной системе Windows консоль может называться «Командная строка» (Console), или «DOS Prompt», или «MS-DOS Prompt» или иметь похожее название, и обычно доступна в меню «Пуск→Все программы→Стандартные» (Start→All Programs→Accessories). В Mac OS X консоль представлена программой `Terminal.app` (по умолчанию находится в меню Applications/Utilities), которую можно отыскать с помощью инструмента `Finder`. В других UNIX-подобных системах можно использовать программу `xterm` или консоль, предоставляемую используемым оконным окружением, например, `konsole` или `gnome-terminal`.

Запустите консоль и в операционной системе Windows введите следующие команды (предполагается, что Python был установлен в каталог по умолчанию) – вывод консоли показан **жирным шрифтом**, а вводимые вами команды – `обычным моноширинным`:

```
C:\>cd c:\py3eg
C:\py3eg\>C:\Python30\python.exe hello.py
```

Поскольку в команде `cd` (change directory – изменить каталог) указан абсолютный путь, не имеет значения, в каком каталоге вы находились перед этим.

Пользователи UNIX должны ввести следующие команды (предполагается, что путь к Python 3 включен в переменную окружения `PATH`):[1]

```
$ cd $HOME/py3eg
$ python3 hello.py
```

В обоих случаях вывод программы должен быть одним и тем же:

```
Hello World!
```

Обратите внимание: если не указано иное, Python ведет себя в Mac OS X точно так же, как и в других системах UNIX. В действительности под общим термином «UNIX» мы подразумеваем Linux, BSD, Mac OS X и большинство других версий UNIX и UNIX-подобных систем.

---

[1] Внешний вид строки приглашения к вводу в системах UNIX может отличаться от $, как показано здесь, но для нас это не имеет никакого значения.

Функция print(), стр. 212

Несмотря на то, что в данной программе имеется всего одна выполняемая инструкция, запуская ее, мы можем сделать некоторые выводы о функции `print()`. С одной стороны, функция `print()` является встроенной частью языка Python – чтобы воспользоваться ею нам не пришлось «импортировать» или «подключать» какие-либо библиотеки. Кроме того, при выводе она отделяет аргументы одним пробелом, а после вывода последнего аргумента выполняет переход на новую строку. Это поведение по умолчанию, которое можно изменить, в чем вы убедитесь позднее. Другое ценное замечание о функции `print()` заключается в том, что она может принимать столько аргументов, сколько потребуется.

Ввод последовательности команд для выполнения программ на языке Python может быстро надоесть. К счастью, и в Windows, и в UNIX существуют более удобные способы. Предположим, что мы уже находимся в каталоге *py3eg*, тогда в Windows достаточно будет ввести команду:

```
C:\py3eg\>hello.py
```

Windows использует свой реестр соответствия программ расширениям файлов и при вводе имен файлов с расширением *.py* автоматически вызывает интерпретатор Python.

Если в консоли будет выведено:

```
('Hello', 'World!')
```

то это означает, что в системе установлен Python 2, который вызывается вместо Python 3. Один из вариантов следующих действий – переключить ассоциацию расширения *.py* с Python 2 на Python 3. Другое решение (менее удобное, но более безопасное) – включить каталог с Python 3 в путь поиска (предполагается, что он был установлен в каталог по умолчанию) и всякий раз явно вызывать его:

```
C:\py3eg\>path=c:\python30;%path%
C:\py3eg\>python hello.py
```

Возможно, было бы удобнее создать единственный пакетный файл *py3.bat* с единственной строкой `path=c:\python30;%path%` и сохранить его в каталоге *C:\Windows*. После этого при запуске консоли для выполнения программ Python 3 достаточно будет выполнять команду `py3.bat`. При желании можно настроить автоматический запуск файла *py3.bat*. Для этого следует изменить свойства консоли: отыщите консоль в меню Пуск (Start), щелкните на ярлыке правой кнопкой мыши и выберите в контекстном меню пункт Свойства (Properties), затем во вкладке Ярлык (Shortcut) в конец строки в поле Объект (Target) добавьте « `/u /k c:\windows\py3.bat`» (обратите внимание на пробелы перед, между и после

параметров «/u» и «/k» и убедитесь, что указанная строка добавлена после «cmd.exe»).

В UNIX сначала файл следует сделать выполняемым, после чего его можно будет запускать, просто введя его имя:

```
$ chmod +x hello.py
$ ./hello.py
```

Конечно, команду chmod придется запускать всего один раз, после этого достаточно будет просто вводить имя программы ./hello.py, чтобы запустить ее.

Когда в UNIX программа запускается в консоли, она читает первые два байта.[1] Если это последовательность ASCII-символов #!, командная оболочка предполагает, что файл должен выполняться интерпретатором, а первая строка файла определяет, какой интерпретатор должен использоваться. Данная строка называется строкой *shebang* (выполняется командной оболочкой) и всегда должна быть первой строкой в файле.

Строка shebang обычно записывается в одной из двух форм:

```
#!/usr/bin/python3
```

или:

```
#!/usr/bin/env python3
```

В первом случае она определяет используемый интерпретатор. Вторая форма может потребоваться для программ на языке Python, запускаемых веб-сервером, хотя абсолютный путь в каждом конкретном случае может отличаться от того, что показан здесь. Во втором случае будет использован первый интерпретатор python3, найденный в текущем окружении. Вторая форма является более универсальной, потому что допускает, что интерпретатор Python 3 может находиться не в каталоге */usr/bin* (то есть он может находиться, например, в каталоге */usr/local/bin* или может быть установлен в каталоге $HOME). Строка shebang не требуется (хотя и не мешает) в операционной системе Windows; все примеры в этой книге имеют строку shebang во второй ее форме, хотя она может быть и не приведена.

Обратите внимание: когда мы говорим о системах UNIX, предполагается, что выполняемый файл Python 3 (или символическая ссылка на него) находится в пути поиска PATH и имеет имя *python3*. Если это не так, вам потребуется изменять строки shebang в примерах, подставив туда корректное имя файла (или корректное имя и путь, если вы пред-

[1] Взаимодействие между пользователем и консолью обслуживается программой «командной оболочки». Нас не интересуют различия, существующие между консолью и командной оболочкой, поэтому эти термины мы будем считать взаимозаменяемыми.

почтете использовать первую форму), или создать символическую ссылку с именем *python3* на выполняемый файл Python 3, поместив ее в один из каталогов, находящийся в пути поиска PATH.

Получение и установка Python, стр. 17

Многие мощные текстовые редакторы, такие как Vim и Emacs, обладают встроенной поддержкой редактирования программ на языке Python. Обычно эта поддержка выражается в предоставлении подсветки синтаксиса и в корректном оформлении отступов в строках. В качестве альтернативы можно использовать IDLE – среду программирования на языке Python. В Windows и Mac OS X IDLE устанавливается по умолчанию, а в UNIX часто предоставляется в виде отдельного пакета, как уже говорилось во введении.

Как показано на снимке с экрана, что приводится на рис. 1.1, среда IDLE имеет весьма непритязательный внешний вид, напоминающий времена Motif в UNIX и Windows 95. Это обусловлено тем, что для создания графического интерфейса среды используется библиотека Tkinter (описывается в главе 13), а не одна из современных и мощных библиотек, таких как PyGtk, PyQt или wxPython. Причины выбора Tkinter корнями уходят в прошлое и в значительной степени обусловлены либеральностью лицензии и тем фактом, что библиотека Tkinter намного меньше других библиотек создания графического интерфейса. Положительная сторона этого выбора – IDLE входит в стандартную поставку Python и очень проста в изучении и использовании.

Рис. 1.1. *Командная оболочка Python в IDLE*

Среда IDLE обеспечивает три ключевые возможности: ввод выражений и программного кода на языке Python с получением результатов прямо в командной оболочке Python; предоставляет редактор программного кода с подсветкой синтаксиса языка Python и поддержкой функции оформления отступов и отладчик, который может использоваться в режиме пошагового выполнения программного кода, облегчая поиск и устранение ошибок. Командная оболочка Python особенно удобна при опробовании простых алгоритмов, фрагментов программного кода и регулярных выражений и может использоваться как очень мощный и гибкий калькулятор.

Для языка Python существуют и другие среды разработки, но мы рекомендуем использовать IDLE – по крайней мере на начальном этапе. При желании для создания программ вы можете использовать простой текстовый редактор, а отладку выполнять посредством инструкций `print()`.

Интерпретатор Python можно запускать самостоятельно, не указывая ему программу на языке Python. В этом случае интерпретатор запускается в интерактивном режиме. В этом режиме можно вводить инструкции языка Python и получать те же результаты, что и в командной оболочке Python среды IDLE, при этом будет выводиться все та же строка приглашения к вводу `>>>`. Но пользоваться IDLE гораздо проще, поэтому мы рекомендуем применять ее для проведения экспериментов с фрагментами программного кода. Короткие интерактивные примеры, которые приводятся в книге, могут вводиться как в интерпретаторе Python, работающем в интерактивном режиме, так и в командной оболочке Python, в среде IDLE.

Теперь мы знаем, как создавать и запускать программы на языке Python, но совершенно очевидно, что мы далеко не уедем, зная всего одну функцию. В следующем разделе мы существенно расширим наши познания о языке Python. Они позволят нам писать пусть и короткие, но уже полезные программы на языке Python, как те, что приводятся в последнем разделе.

«Золотой запас» Python

В этом разделе мы узнаем о восьми ключевых составляющих языка Python, а в следующем разделе увидим, как используются эти составляющие на примере пары маленьких, но практичных программ. Обсуждение описываемых здесь тем ведется не только в этой главе, поэтому, если вы почувствуете, что информации не хватает или что-то выглядит слишком громоздко, воспользуйтесь ссылками вперед, содержанием или предметным указателем; практически всегда обнаружится, что Python предоставляет нужную вам особенность, к тому же в более краткой и выразительной форме, чем показано здесь, и, кроме того, обнаружится еще много чего вокруг.

Составляющая №1: типы данных

Одна из фундаментальных особенностей любого языка программирования заключается в способности представлять элементы данных. Язык Python предоставляет несколько встроенных типов данных, но пока интерес для нас представляют только два из них. В языке Python для представления целых чисел (положительных и отрицательных) используется тип `int`, а для представления строк (последовательностей символов Юникода) используется тип `str`. Ниже приводятся несколько примеров литералов целых чисел и строк:

```
-973
210624583337114373395836055367340864637790190801098222508621955072
0
"Infinitely Demanding"
'Simon Critchley'
'positively αβγ€÷©'
''
```

Между прочим, второе число в этом примере – это число 2^{217} – размер целых чисел в языке Python ограничивается только объемом памяти, имеющейся в компьютере, а не фиксированным числом байтов. Строки могут ограничиваться кавычками или апострофами при условии, что с обоих концов используются однотипные кавычки, а поскольку для представления строк Python использует Юникод, строки могут содержать не только символы из набора ASCII, как показано в предпоследней строке примера. Пустые строки – это просто кавычки, внутри которых ничего нет.

Для доступа к элементам последовательностей, таким как символы в строках, в языке Python используются квадратные скобки (`[]`). Например, находясь в командной оболочке Python (когда интерпретатор запущен в интерактивном режиме или в среде IDLE) мы можем ввести следующее – вывод командной оболочки Python выделен **жирным** шрифтом, а то, что вводится с клавиатуры – обычным моноширинным шрифтом:

```
>>> "Hard Times"[5]
'T'
>>> "giraffe"[0]
'g'
```

Традиционно в командной оболочке Python строка приглашения к вводу имеет вид `>>>`, но ее можно изменить. Квадратные скобки могут использоваться для доступа к элементам любых типов данных, являющихся последовательностями, таких как строки и списки. Такая непротиворечивость синтаксиса – одно из оснований красоты языка Python. Обратите внимание: индексы в языке Python начинаются с 0.

В Python тип `str` и элементарные числовые типы, такие как `int`, являются *неизменяемыми*, то есть однажды установив значение, его уже нельзя будет изменить. На первый взгляд, такое ограничение кажется

странным, но на практике это не влечет за собой никаких проблем. Единственная причина, по которой об этом было упомянуто здесь, заключается в том, что имея возможность с помощью квадратных скобок извлекать отдельные символы, мы не имеем возможности изменять их. (Обратите внимание: в языке Python под символом понимается строка, имеющая длину, равную 1.)

Для преобразования элемента данных из одного типа в другой мы можем использовать конструкцию *datatype(item)*. Например:

```
>>> int("45")
45
>>> str(912)
'912'
```

Преобразование `int()` терпимо относится к начальным и конечным пробелам, поэтому оператор `int("45")` также будет работать. Преобразование `str()` может применяться практически к любым типам данных. Мы легко можем наделять поддержкой преобразований `str()`, `int()` и других преобразований свои собственные типы данных, если в этом имеется какой-то смысл; это будет показано в главе 6. Если преобразование терпит неудачу, возбуждается исключение – мы коротко затронем тему обработки исключений, когда будем рассматривать составляющую №5, а полное обсуждение исключений приводится в главе 4.

Строки и целые числа подробно будут обсуждаться в главе 2 наряду с другими встроенными типами данных и некоторыми другими типами из стандартной библиотеки Python. В этой главе также будут рассматриваться операции, применимые к неизменяемым последовательностям, таким как строки.

Составляющая №2: ссылки на объекты

Теперь, зная о существовании некоторых типов данных, нам необходимо рассмотреть переменные, которые хранят эти данные. В языке Python нет переменных как таковых – вместо них используются *ссылки на объекты*. Когда речь заходит о неизменяемых объектах, таких как `int` или `str`, между переменной и ссылкой на объект нет никакой разницы. Однако различия начинают проявляться, когда дело доходит до изменяемых объектов, но эти различия редко имеют практическое значение. Мы будем использовать термины *переменная* и *ссылка на объект* как взаимозаменяемые.

Поверхностное и глубокое копирование, стр. 173

Взгляните на следующие крошечные примеры, а затем мы обсудим их подробнее.

```
x = "blue"
y = "green"
z = x
```

Синтаксис выглядит очень просто: *objectReference = value*. Нет никакой необходимости в предварительном объявлении или определении типов значений. Когда интерпретатор Python выполняет первую инструкцию, он создает объект типа `str` с текстом `"blue"`, а затем создает ссылку на объект с именем `x`, которая ссылается на объект типа `str`. Практически можно сказать, что «переменной `x` была присвоена строка `'blue'`». Вторая инструкция похожа на первую. Третья инструкция создает новую ссылку на объект с именем `z` и записывает в нее ссылку на тот же самый объект, на который указывает ссылка `x` (в данном случае это объект типа `str` с текстом `"blue"`).

Оператор `=` — это не оператор присваивания значения переменной, как в некоторых других языках программирования. Оператор `=` связывает ссылку на объект с объектом, находящимся в памяти. Если ссылка на объект уже существует, ее легко можно связать с другим объектом, указав этот объект справа от оператора `=`. Если ссылка на объект еще не существует, она будет создана оператором `=`.

Продолжим пример со ссылками `x`, `y` и `z` и выполним перепривязку. Как уже упоминалось ранее, с символа `#` начинаются комментарии, которые продолжаются до конца строки:

```
print(x, y, z) # выведет: blue green blue
z = y
print(x, y, z) # выведет: blue green green
x = z
print(x, y, z) # выведет: green green green
```

После выполнения четвертой инструкции (`x = z`) все три ссылки на объекты будут ссылаться на один и тот же объект типа `str`. Поскольку теперь не осталось ни одной ссылки, которая ссылалась бы на строку `"blue"`, Python сможет утилизировать ее.

На рис. 1.2 схематически изображены взаимоотношения между объектами и ссылками на объекты.

Рис. 1.2. *Объекты и ссылки на объекты*

Имена, используемые для идентификации ссылок на объекты (называются *идентификаторами*), имеют определенные ограничения. В частности, они не могут совпадать с ключевыми словами языка Python и должны начинаться с алфавитного символа или с символа подчеркивания, за которым следует ноль или более алфавитных символов, символов подчеркивания или цифр. Ограничений на длину имен не накладывается, а алфавитные и цифровые символы – это те символы, что определяются Юникодом, включая, но не ограничиваясь цифрами и символами ASCII («a», «b», ..., «z», «A», «B», ..., «Z», «0», «1», ..., «9»). Идентификаторы в языке Python чувствительны к регистру символов, то есть имена `LIMIT`, `Limit` и `limit` – это три разных имени. Дополнительные подробности по этому вопросу и ряд необычных примеров приводятся в главе 2.

Идентификаторы и ключевые слова, стр. 68

В языке Python используется *динамический контроль типов*, то есть ссылки на объекты в любой момент могут повторно привязываться к различным объектам (которые могут относиться к данным различных типов). В языках со строгим контролем типов (таких как C++ и Java) разрешается выполнять только те операции, которые допустимы для данных, участвующих в операции. В языке Python также имеется подобное ограничение, но в данном случае это не называется строгим типизированием, потому что допустимость операции может измениться, например, когда ссылка на объект будет повторно связана с объектом другого типа. Например:

```
route = 866
print(route, type(route)) # выведет: 866 <class 'int'>
route = "North"
print(route, type(route)) # выведет: North <class 'str'>
```

Здесь была создана новая ссылка на объект с именем `route` и связана с новым значением 866 типа `int`. С этого момента мы можем использовать оператор / применительно к `route`, потому что деление – это допустимая операция для целых чисел. После этого мы вновь воспользовались ссылкой `route` и связали ее с новым объектом типа `str`, имеющим значение «North», а объект типа `int` был утилизирован сборщиком мусора, так как не осталось ни одной ссылки, которая ссылалась бы на него. С этого момента применение оператора / будет вызывать ошибку `TypeError`, так как деление не является допустимой операцией для строк.

Функция `type()` возвращает тип данных (который также называется «классом») для указанного элемента. Эта функция может быть очень полезной на этапе тестирования и отладки, но в готовом программном коде, как

Функция isinstance(), стр. 284

правило, не используется, так как существует более удобная альтернатива, в чем вы убедитесь в главе 6.

При экспериментировании с программным кодом на языке Python в интерактивной оболочке интерпретатора или в командной оболочке Python – такой, как предоставляется средой IDLE, достаточно просто ввести имя ссылки на объект, чтобы интерпретатор вывел значение связанного с ней объекта. Например:

```
>>> x = "blue"
>>> y = "green"
>>> z = x
>>> x
'blue'
>>> x, y, z
('blue', 'green', 'blue')
```

Это намного удобнее, чем постоянно вызывать функцию `print()`, но эта особенность работает только при использовании интерпретатора Python в интерактивном режиме – в любых создаваемых программах и модулях для вывода значений следует использовать функцию `print()` или подобные ей. Обратите внимание, что в последнем случае Python вывел значения в круглых скобках, разделив их запятыми – так обозначается тип данных `tuple` (кортеж), то есть упорядоченная, неизменяемая последовательность объектов. О кортежах мы поговорим в следующем разделе.

Составляющая №3: коллекции данных

Часто бывает удобно хранить целую коллекцию элементов данных. В языке Python для этого имеется несколько типов коллекций, способных хранить элементы данных, включая ассоциативные массивы и множества. Но в этом разделе мы рассмотрим только два типа коллекций: `tuple` (кортежи) и `list` (списки). Кортежи и списки в языке Python могут использоваться для хранения произвольного числа элементов данных любых типов. Кортежи относятся к разряду *неизменяемых* объектов, поэтому после создания кортеж нельзя изменить. Списки относятся к разряду *изменяемых* объектов, поэтому мы легко можем вставлять и удалять элементы списка по своему желанию.

Кортежи создаются с помощью запятых (`,`), как показано ниже:

```
>>> "Denmark", "Norway", "Sweden"
('Denmark', 'Norway', 'Sweden')
>>> "one",
('one',)
```

Создание и вызов функций, стр. 52

При выводе кортежа интерпретатор Python заключает его в круглые скобки. Многие программисты имитируют такое поведение и заключают литералы кортежей в круглые скобки. Если создается кортеж с одним элементом,

то даже при наличии круглых скобок мы обязаны использовать запятую, например: `(1,)`. Пустой кортеж создается с помощью пустых круглых скобок `()`. Запятая также используется для отделения аргументов при вызове функции, поэтому, если в качестве аргумента требуется передать литерал кортежа, мы должны заключать его в круглые скобки, чтобы избежать неоднозначности.

Ниже приводятся несколько примеров списков:

```
[1, 4, 9, 16, 25, 36, 49]
['alpha', 'bravo', 'charlie', 'delta', 'echo']
['zebra', 49, -879, 'aardvark', 200]
[]
```

Как показано здесь, списки могут создаваться с помощью квадратных скобок (`[]`), но позднее мы познакомимся с другими способами создания списков. Четвертый список в примере – это пустой список.

Списки и кортежи хранят не сами элементы данных, а ссылки на объекты. При создании списков и кортежей (а также при добавлении новых элементов в списки) создаются копии указанных ссылок на объекты. В случае значений-литералов, таких как целые числа и строки, в памяти создаются и инициализируются объекты соответствующих типов, затем создаются ссылки, указывающие на эти объекты, и эти ссылки помещаются в список или в кортеж.

Как и все остальное в языке Python, коллекции данных – это объекты: благодаря этому имеется возможность вкладывать одни объекты-коллекции в другие, например, без лишних формальностей можно создать список списков. В некоторых ситуациях тот факт, что кортежи и списки, а также большинство коллекций других типов, хранят ссылки на объекты, а не сами объекты, имеет большое значение – об этом будет рассказываться в главе 3 (начиная со стр. 136).

Поверхностное и глубокое копирование, стр. 173

В процедурном программировании мы вызываем функции и часто передаем им данные в виде аргументов. Например, мы уже познакомились с функцией `print()`. Другая часто используемая функция в языке Python – это функция `len()`, которая в качестве аргумента принимает единственный элемент данных и возвращает его «длину» в виде значения типа `int`. Ниже приводятся несколько примеров вызова функции `len()` – в этом примере мы не стали выделять вывод интерпретатора жирным шрифтом, полагая, что вы уже сами сможете отличить, что вводится с клавиатуры, а что выводится интерпретатором:

```
>>> len(("one",))
1
>>> len([3, 5, 1, 2, "pause", 5])
6
>>> len("automatically")
13
```

Понятие «размер», стр. 443

Кортежи, списки и строки имеют «размер», то есть это типы данных, которые обладают категорией размера, и элементы данных таких типов могут передаваться функции `len()`. (Если функции `len()` передать элемент, тип которого не предполагает такого понятия, как размер, будет возбуждено исключение.)

Все элементы данных в языке Python являются *объектами* (называемых также *экземплярами*) определенных типов данных (называемых также *классами*). Мы будем использовать термины *тип данных* и *класс* как взаимозаменяемые. Одно из основных отличий между объектом и простым элементом данных в некоторых других языках программирования (например, встроенные числовые типы в C++ или Java) состоит в том, что объект может обладать *методами*. В сущности, метод – это обычная функция, которая вызывается в контексте конкретного объекта. Например, тип `list` имеет метод `append()`, с помощью которого можно добавить новый объект в список, как показано ниже:

```
>>> x = ["zebra", 49, -879, "aardvark", 200]
>>> x.append("more")
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more']
```

Объект `x` знает, что принадлежит к типу `list` (все объекты в языке Python знают, к какому типу они принадлежат), поэтому нам не требуется явно указывать тип данных. Первым аргументом методу `append()` передается сам объект `x` – делается это интерпретатором Python автоматически, в порядке реализованной в нем поддержки методов.

Метод `append()` изменяет первоначальный список. Это возможно благодаря тому, что списки относятся к категории изменяемых объектов. Потенциально это более эффективно, чем создание нового списка, включающего первоначальные элементы и дополнительный элемент с последующим связыванием ссылки на объект с новым списком, особенно для очень длинных списков.

В процедурном языке то же самое могло бы быть достигнуто с использованием метода `append()`, как показано ниже (что является совершенно допустимым вариантом в языке Python):

```
>>> list.append(x, "extra")
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more', 'extra']
```

Здесь указываются тип данных и метод этого типа данных, а в качестве первого аргумента передается элемент данных того типа, метод которого вызывается, за которым следуют дополнительные параметры. (С точки зрения наследования между этими двумя подходами существует малозаметное семантическое отличие; на практике наиболее часто используется первая форма. О наследовании рассказывается в главе 6.)

Если вы еще не знакомы с объектно-ориентированным программированием, на первый взгляд такая форма вызова функций может показаться немного странной. Пока вам достаточно будет знать, что обычные функции в языке Python вызываются так: *functionName(argiments)*, а методы вызываются так: *objectName.methodName(arguments)*. (Об объектно-ориентированном программировании рассказывается в главе 6.)

Оператор точки («оператор доступа к атрибуту») используется для доступа к атрибутам объектов. До сих пор мы видели только одну разновидность атрибутов – методы, но атрибут может быть объектом любого типа. Поскольку атрибут может быть объектом, имеющим свои атрибуты, которые также могут быть объектами, обладающими атрибутами и т. д., мы можем использовать столько операторов точки, сколько потребуется для доступа к необходимому атрибуту.

Тип `list` имеет множество других методов, включая `insert()`, который используется для вставки элемента в позицию с определенным индексом, и `remove()`, который удаляет элемент с указанным индексом. Как уже упоминалось ранее, счет индексов в языке Python начинается с 0.

Ранее мы уже видели, что существует возможность извлечь из строки символ, используя оператор квадратных скобок, и отметили, что этот оператор может использоваться с любыми последовательностями. Списки – это последовательности, поэтому мы можем выполнять следующие операции:

```
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more', 'extra']
>>> x[0]
'zebra'
>>> x[4]
200
```

Кортежи также являются последовательностями, поэтому, если бы ссылка `x` указывала на кортеж, мы могли бы извлекать элементы с помощью квадратных скобок точно так же, как это было сделано со ссылкой `x`, представляющей список. Но так как списки являются изменяемыми объектами (в отличие от строк и кортежей, которые являются неизменяемыми объектами), мы можем использовать оператор квадратных скобок еще и для изменения элементов списка. Например:

```
>>> x[1] = "forty nine"
>>> x
['zebra', 'forty nine', -879, 'aardvark', 200, 'more', 'extra']
```

Если указать индекс, выходящий за пределы списка, будет возбуждено исключение – обработка исключений вкратце будет рассматриваться в разделе с описанием составляющей №5, а полный охват этой темы дается в главе 4.

Мы уже несколько раз использовали термин *последовательность*, полагаясь на неформальное понимание его смысла, и продолжим его

использовать в том же духе. Однако язык Python дает точное определение последовательностей и какие особенности должны ими поддерживаться, точно так же он точно определяет, какие особенности должны поддерживаться объектами, имеющими размер, и так далее для других категорий, которым могут принадлежать типы данных, но об этом будет рассказываться в главе 8.

Списки, кортежи и другие типы коллекций, встроенные в язык Python, описываются в главе 3.

Составляющая №4: логические операции

Одна из фундаментальных особенностей любого языка программирования заключается в поддержке логических операций. Язык Python предоставляет четыре набора логических операций, и здесь мы рассмотрим основные принципы их использования.

Оператор идентичности

Поскольку все переменные в языке Python фактически являются ссылками, иногда возникает необходимость определить, не ссылаются ли две или более ссылок на один и тот же объект. Оператор `is` – это двухместный оператор, который возвращает `True`, если ссылка слева указывает на тот же самый объект, что и ссылка справа. Ниже приводятся несколько примеров:

```
>>> a = ["Retention", 3, None]
>>> b = ["Retention", 3, None]
>>> a is b
False
>>> b = a
>>> a is b
True
```

Обратите внимание, что обычно не имеет смысла использовать оператор `is` для сравнения типов данных `int`, `str` и некоторых других, так как обычно в этом случае бывает необходимо сравнить их значения. Фактически результаты сравнения данных с помощью оператора `is` могут приводить в замешательство, как это видно из предыдущего примера, где `a` и `b` первоначально имеют одинаковые значения-списки, однако сами списки хранятся в виде отдельных объектов типа `list`, и поэтому в первом случае оператор `is` вернул `False`.

Одно из преимуществ операции проверки идентичности заключается в высокой скорости ее выполнения. Это обусловлено тем, что сравниваются ссылки на объекты, а не сами объекты. Оператор `is` сравнивает только адреса памяти, в которых располагаются объекты – если адреса равны, следовательно, ссылки указывают на один и тот же объект.

Чаще всего оператор `is` используется для сравнения элемента данных со встроенным пустым объектом `None`, который часто применяется, чтобы показать, что значение «неизвестно» или «не существует»:

```
>>> a = "Something"
>>> b = None
>>> a is not None, b is None
(True, True)
```

Для выполнения проверки на неидентичность, используется оператор `is not`.

Назначение оператора проверки идентичности состоит в том, чтобы узнать, ссылаются ли две ссылки на один и тот же объект, или проверить, не ссылается ли ссылка на объект `None`. Если требуется сравнить значения объектов, мы должны использовать операторы сравнения.

Операторы сравнения

Язык Python предоставляет стандартный набор двухместных операторов сравнения с предсказуемой семантикой: `<` меньше чем, `<=` меньше либо равно, `==` равно, `!=` не равно, `>=` больше либо равно и `>` больше чем. Эти операторы сравнивают значения объектов, то есть объекты, на которые указывают ссылки, участвующие в сравнении. Ниже приводятся примеры, набранные в командной оболочке Python:

```
>>> a = 2
>>> b = 6
>>> a == b
False
>>> a < b
True
>>> a <= b, a != b, a >= b, a > b
(True, True, False, False)
```

Для целых чисел они действуют, как и следовало ожидать. Точно так же корректно выполняется сравнение строк:

```
>>> a = "many paths"
>>> b = "many paths"
>>> a is b
False
>>> a == b
True
```

Хотя `a` и `b` ссылаются на разные объекты (с различными идентификаторами), тем не менее они имеют одинаковые значения, поэтому результат сравнения говорит о том, что они равны. Однако не следует забывать, что в языке Python для представления строк используется кодировка Юникод, поэтому сравнение строк, содержащих сим-

Сравнение строк, стр. 88

волы, не входящие в множество ASCII-символов, может оказаться делом более сложным, чем кажется на первый взгляд – подробнее эта проблема будет рассматриваться в главе 2.

Одна из интересных особенностей операторов сравнения в языке Python заключается в том, что они могут объединяться в цепочки. Например:

```
>>> a = 9
>>> 0 <= a <= 10
True
```

Это отличный способ убедиться, что некоторый элемент данных находится в пределах диапазона, вместо того чтобы выполнять два отдельных сравнения, результаты которых объединяются логическим оператором `and`, как это принято во многих других языках программирования. Кроме того, у этого подхода имеется дополнительное преимущество, так как оценка значения элемента данных производится только один раз (поскольку в выражении он появляется всего один раз), что может иметь большое значение, особенно когда вычисление значения элемента данных является достаточно дорогостоящей операцией или когда обращение к элементу данных имеет побочные эффекты.

Благодаря «строгости» механизма динамического контроля типов в языке Python попытка сравнения несовместимых значений вызывает исключение. Например:

```
>>> "three" < 4
Traceback (most recent call last):
(Трассировочная информация (самый последний вызов внизу)
...
TypeError: unorderable types: str() < int()
(TypeError: несопоставимые типы: str() < int())
```

В случае, когда возбуждается необрабатываемое исключение, интерпретатор Python выводит диагностическую информацию и текст сообщения об ошибке. Для простоты мы опустили диагностическую информацию здесь, заменив ее многоточием.[1] То же самое исключение `TypeError` возникнет, если записать `"3" < 4`, потому что Python никогда не пытается строить предположения о наших намерениях – правильный подход заключается в том, чтобы выполнить явное преобразование, например: `int("3") < 4`, или использовать сопоставимые типы, то есть оба значения должны быть либо целыми числами, либо строками.

[1] Диагностическая информация (иногда называется обратной трассировкой (backtrace)) – это список всех вызовов функций, которые были произведены к моменту возбуждения необрабатываемого исключения, в обратном порядке.

Язык Python позволяет нам легко создавать свои собственные, легко интегрируемые типы данных, благодаря чему, например, мы могли бы создать свой собственный числовой тип, который смог бы участвовать в операциях сравнения со встроенным типом `int` и другими встроенными или нашими собственными числовыми типами, но не со строками или другими нечисловыми типами.

Альтернативный тип `FuzzyBool`, стр. 300

Оператор членства

Для типов данных, являющихся последовательностями или коллекциями, таких как строки, списки и кортежи, мы можем выполнить проверку членства с помощью оператора `in`, а проверку обратного утверждения – с помощью оператора `not in`. Например:

```
>>> p = (4, "frog", 9, -33, 9, 2)
>>> 2 in p
True
>>> "dog" not in p
True
```

Применительно к спискам и кортежам оператор `in` выполняет линейный поиск, который может оказаться очень медленным для огромных коллекций (десятки тысяч элементов или больше). С другой стороны, со словарями или со множествами оператор `in` работает очень быстро – оба этих типа данных рассматриваются в главе 3. Ниже демонстрируется, как оператор `in` может использоваться со строками:

```
>>> phrase = "Peace is no longer permitted during Winterval"
>>> "v" in phrase
True
>>> "ring" in phrase
True
```

Применительно к строкам оператор `in` удобно использовать для проверки вхождения в строку подстроки произвольной длины. (Как отмечалось ранее, символ – это всего лишь строка, имеющая длину, равную 1.)

Логические операторы

Язык Python предоставляет три логических оператора: `and`, `or` и `not`. Операторы `and` и `or` вычисляются по короткой схеме и возвращают операнд, определяющий результат, – они не возвращают значение типа Boolean (если операнды не являются значениями типа Boolean). Взгляните, что это означает на практике:

```
>>> five = 5
>>> two = 2
>>> zero = 0
>>> five and two
2
```

```
>>> two and five
5
>>> five and zero
0
```

Если выражение участвует в логическом контексте, результат оценивается как значение типа Boolean, поэтому предыдущие выражения могли бы рассматриваться, как имеющие значения `True`, `True` и `False`, например, в контексте инструкции `if`.

```
>>> nought = 0
>>> five or two
5
>>> two or five
2
>>> zero or five
5
>>> zero or nought
0
```

Оператор `or` напоминает оператор `and` – здесь в булевом контексте были бы получены значения `True`, `True`, `True` и `False`.

Унарный оператор `not` оценивает свой операнд в булевом контексте и всегда возвращает значение типа Boolean, поэтому, продолжая предыдущий пример, выражение `not (zero or nought)` вернет значение `True`, а выражение `not two` – `False`.

Составляющая №5: инструкции управления потоком выполнения

Мы упоминали ранее, что все инструкции, встречающиеся в файле с расширением *.py*, выполняются по очереди, строка за строкой. Порядок движения потока управления может изменяться вызовами функций и методов или структурами управления, такими как условные операторы или операторы циклов. Поток управления также отклоняется, когда возбуждаются исключения.

В этом подразделе мы рассмотрим условный оператор `if`, а также операторы циклов `while` и `for`, отложив рассмотрение функций до раздела с описанием составляющей №8, а рассмотрение методов – до главы 6. Мы также коротко коснемся вопросов обработки исключений, которые подробнее будут обсуждаться в главе 4. Но для начала мы определим пару терминов.

Булево (Boolean) выражение – это выражение, которое может оцениваться как булево (Boolean) значение (`True` или `False`). В языке Python выражение оценивается как `False`, если это предопределенная константа `False`, специальный объект `None`, пустая последовательность или коллекция (например, пустая строка, список или кортеж), или числовой элемент данных со значением 0. Все остальное оценивается как

`True`. При создании своих собственных типов данных (как будет рассказываться в главе 6) мы сами сможем определять, какое значение возвращать в булевом контексте.

В языке Python используется понятие блока программного кода – *suite*[1], представляющего собой последовательность одной или более инструкций. Так как некоторые синтаксические конструкции языка Python *требуют* наличия блока кода, Python предоставляет ключевое слово `pass`, которое представляет собой инструкцию, не делающую ровным счетом ничего, и которая может использоваться везде, где требуется блок кода (или когда мы хотим указать на наличие особого случая), но никаких действий выполнять не требуется.

Инструкция if

В общем случае инструкция `if` имеет следующий синтаксис:[2]

```
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
...
elif boolean_expressionN:
    suiteN
else:
    else_suite
```

В инструкции может быть ноль или более предложений `elif`, а заключительное предложение `else` является необязательным. Если необходимо принять в учет какой-то особый случай, не требующий обработки, мы можем использовать ключевое слово `pass` в качестве блока кода соответствующей ветки.

Первое, что бросается в глаза программистам, использовавшим язык C++ или Java, – это отсутствие круглых и фигурных скобок. Еще одна особенность, на которую следует обратить внимание, – это двоеточие, которое является частью синтаксиса и о котором легко забыть на первых порах. Двоеточия используются с предложениями `else`, `elif` и практически везде, где вслед за предложением должен следовать блок кода.

В отличие от большинства других языков программирования, отступы в языке Python используются для обозначения блочной структуры. Некоторым программистам не нравится эта черта, особенно до того, пока они не испытают ее на практике, а некоторые весьма эмоционально высказываются по этому поводу. Однако чтобы привыкнуть к ней, требуется всего несколько дней, а спустя несколько недель или

---

[1] Во многих других языках программирования такой блок программного кода называется *составным оператором*. – *Прим. перев.*

[2] В этой книге многоточия (...) приводятся взамен строк, которые не показаны.

месяцев программный код без скобок начинает восприниматься как более удобочитаемый и менее захламленный, чем программный код со скобками.

Так как блоки кода оформляются посредством отступов, естественно, возникает вопрос, какие отступы использовать. Руководства по оформлению программного кода на языке Python рекомендуют использовать четыре пробела на каждый уровень и использовать только пробелы (а не символы табуляции). Большинство современных текстовых редакторов могут быть настроены на автоматическую обработку отступов (редактор среды IDLE, конечно же, предусматривает такую возможность, как и большинство других редакторов, поддерживающих язык программирования Python). Интерпретатор Python прекрасно будет справляться с любыми отступами, содержащими любое число пробелов или символов табуляции или смесь из тех и других – главное, чтобы оформление отступов выполнялось непротиворечивым образом. В этой книге мы следуем официальным рекомендациям Python.

Ниже приводится пример простой инструкции `if`:

```
if x:
    print("x is nonzero")
```

В данном случае, если условное выражение (`x`) оценивается как `True`, блок кода (вызов функции `print()`) будет выполнен.

```
if lines < 1000:
    print("small")
elif lines < 10000:
    print("medium")
else:
    print("large")
```

Это немного более сложная инструкция `if`, которая выводит оценку, описывающую значение переменной `lines`.

Инструкция while

Инструкция `while` используется для выполнения своего блока кода ноль или более раз, причем число раз зависит от булева выражения в инструкции `while`. Ниже приводится синтаксис этой инструкции:

```
while boolean_expression:
    suite
```

В действительности полный синтаксис цикла `while` – более сложный, чем показано здесь, так как дополнительно поддерживаются инструкции `break` и `continue`, а также предложение `else`, о чем подробно будет рассказываться в главе 4. Инструкция `break` передает управление за пределы самого внутреннего цикла, в котором она встречается, то есть прерывает выполнение цикла. Инструкция `continue` передает управление в начало цикла. Как правило, инструкции `break` и `continue` исполь-

зуются внутри инструкций if, чтобы в зависимости от складывающихся условий изменять поведение цикла.

```
while True:
    item = get_next_item()
    if not item:
        break
    process_item(item)
```

Этот цикл while имеет вполне типичную структуру и выполняется до тех пор, пока не будут исчерпаны элементы для обработки. (Предполагается, что функции get_next_item() и process_item() – это наши собственные функции, определенные где-то в другом месте.) В этом примере блок кода инструкции while содержит инструкцию if, имеющую, как ей и положено, свой собственный блок кода – в данном случае состоящий из единственной инструкции break.

Инструкция for ... in

Инструкция цикла for в языке Python повторно использует ключевое слово in (которое в других контекстах играет роль оператора проверки членства) и имеет следующий синтаксис:

```
for variable in iterable:
    suite
```

Точно так же, как и в случае с циклом while, инструкция for поддерживает инструкции break и continue, а также необязательное предложение else. Переменная *variable* поочередно указывает на каждый объект в объекте *iterable*. В качестве *iterable* может использоваться любой тип данных, допускающий выполнение итераций по нему, включая строки (где итерации выполняются по символам строки), списки, кортежи и другие типы коллекций языка Python.

```
for country in ["Denmark", "Finland", "Norway", "Sweden"]:
    print(country)
```

Здесь мы использовали весьма упрощенный подход к выводу списка стран. На практике то же самое обычно делается с помощью переменных:

```
countries = ["Denmark", "Finland", "Norway", "Sweden"]
for country in countries:
    print(country)
```

На самом деле весь список (или кортеж) можно вывести единственным вызовом функции print(), например, print(countries), но часто предпочтительнее выводить содержимое коллекций в цикле for (или в генераторах списков, о которых будет рассказываться позже), чтобы иметь полный контроль над форматированием.

Генераторы списков, стр. 142

```
for letter in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
    if letter in "AEIOU":
        print(letter, "is a vowel")
    else:
        print(letter, "is a consonant")
```

В первой строке этого фрагмента мы использовали ключевое слово `in`, как часть инструкции `for`, и переменную `letter`, последовательно принимающую значения `"A"`, `"B"` и так далее до `"Z"` на каждом проходе цикла. Во второй строке мы снова использовали ключевое слово `in`, но на этот раз в качестве оператора проверки членства. Обратите также внимание на то, что в этом примере присутствуют вложенные блоки кода. Блок кода цикла `for` представлен инструкцией `if ...else`, а обе ветки – и `if`, и `else` – имеют свои собственные блоки кода.

Основы обработки исключений

Многие функции и методы в языке Python в случае появления ошибок или наступления других важных событий возбуждают исключения. Исключение – это объект, такой же как любые другие объекты в языке Python, который при преобразовании в строку (то есть при выводе на экран) воспроизводится как строка с текстом сообщения. Ниже показана простейшая форма обработки исключений:

```
try:
    try_suite
except exception1 as variable1:
    exception_suite1
…
except exceptionN as variableN:
    exception_suiteN
```

Обратите внимание на то, что часть `as variable` является необязательной – нас может волновать только сам факт возникновения исключения, а текст сообщения может быть для нас неинтересен.

Полный синтаксис имеет более сложный вид, например, каждое предложение `except` может обрабатывать несколько исключений, а кроме того, можно использовать необязательное предложение `else`. Обо всем этом подробно рассказывается в главе 4.

Эта конструкция работает следующим образом. Если инструкции в блоке кода `try` выполняются без ошибок, блоки `except` просто пропускаются. Если в блоке `try` будет возбуждено исключение, управление немедленно будет передано блоку первого соответствующего предложения `except` – то есть любые инструкции в блоке `try`, расположенные ниже инструкции, вызвавшей исключение, выполняться не будут. В случае исключения и если указана часть `as variable`, внутри блока обработки исключения переменная *variable* будет ссылаться на объект исключения.

Если исключение возникнет в блоке except или для возникшего исключения не будет обнаружено соответствующего предложения except, то Python начнет поиск блока except в следующем объемлющем контексте. Поиск подходящего обработчика исключения будет выполняться в направлении наружу и вверх по стеку вызовов, пока не будет найдено соответствие. Если соответствия найдено не будет, то программа будет аварийно завершена с выводом сообщения об ошибке. В случае появления необработанного исключения интерпретатор Python выводит диагностическую информацию и текст сообщения.

Например:

```
s = input("enter an integer: ")
try:
    i = int(s)
    print("valid integer entered:", i)
except ValueError as err:
    print(err)
```

Если пользователь введет 3.5, будет выведено сообщение:

```
invalid literal for int() with base 10: '3.5'
(неверный литерал типа int() по основанию 10: '3.5')
```

Но если он введет 13, вывод будет иной:

```
valid integer entered: 13
```

В многих книгах считается, что обработка исключений – это тема повышенной сложности, и ее рассмотрение откладывается как можно дальше. Но возбуждение и особенно обработка исключений – это фундаментальный способ работы Python. поэтому мы затронули этот вопрос в самом начале. И, как будет показано позже, обработчики исключений могут сделать программный код более удобочитаемым, отделяя «исключительные» случаи от обработки, которая для нас представляет наибольший интерес.

Составляющая №6: арифметические операторы

Язык Python предоставляет полный комплект арифметических операторов, включая двухместные операторы, выполняющие четыре основных арифметических действия: + (сложение), - (вычитание), * (умножение) и / (деление). Кроме того, многие типы данных в языке Python допускают использование комбинированных операторов присваивания (или, в соответствии с другой используемой системой терминов, – операторов *дополняющего присваивания*), таких как += и *=. Операторы +, - и * действуют так, как и следовало ожидать, когда в качестве операндов выступают целые числа:

```
>>> 5 + 6
11
>>> 3 - 7
```

```
-4
>>> 4 * 8
32
```

Обратите внимание, что оператор может использоваться как унарный (оператор отрицания) и как двухместный оператор (вычитание), что является вполне обычным для большинства языков программирования. Язык Python начинает выделяться из общей массы других языков, когда дело доходит до оператора деления:

```
>>> 12 / 3
4.0
>>> 3 / 2
1.5
```

Числовые операторы и функции, стр. 74

Оператор деления возвращает значение с плавающей точкой, а не целое значение. Во многих других языках он возвращает целое значение, отсекая дробную часть. Если требуется получить целочисленный результат, его всегда можно преобразовать с помощью `int()` (или использовать оператор деления с усечением `//`, который будет рассматриваться позже).

```
>>> a = 5
>>> a
5
>>> a += 8
>>> a
13
```

На первый взгляд в предыдущих инструкциях нет ничего особенного, особенно для тех, кто знаком с C-подобными языками программирования. В таких языках программирования комбинированные операторы присваивания являются сокращенной формой записи присваивания результата операции, например, выражение `a += 8` эквивалентно выражению `a = a + 8`. Однако следует иметь в виду две важные тонкости: одна имеет отношение только к языку Python и одна характерна для комбинированных операторов присваивания во всех языках.

Первое, что важно запомнить, это то, что тип `int` является неизменяемым, то есть после присваивания значение типа `int` нельзя изменить. Поэтому при выполнении комбинированного оператора присваивания с неизменяемыми объектами в действительности создается новый объект с результатом, а затем целевая ссылка на объект привязывается к этому новому объекту. То есть когда в предыдущем случае интерпретатор Python встретит инструкцию `a += 8`, он вычислит значение `a + 8`, сохранит результат в новом объекте типа `int` и запишет в `a` ссылку на этот новый объект типа `int`. (Если после этого в программе не останется ни одной ссылки, указывающей на первоначальный объект, он будет утилизирован сборщиком мусора.) Рис. 1.3 иллюстрирует, как это происходит.

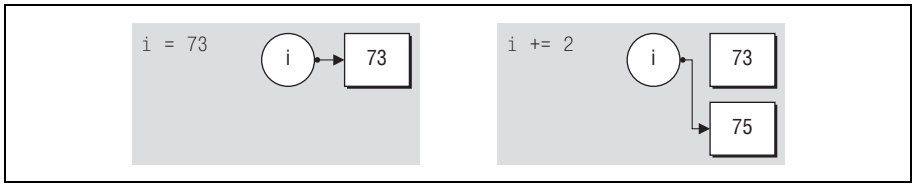

***Рис. 1.3.** Комбинированные операторы присваивания с неизменяемыми объектами*

Вторая особенность заключается в том, что выражение `a operator= b` — это не совсем то же самое, что выражение `a = a operator b`. Комбинированная версия ищет значение `a` всего один раз, поэтому потенциально она выполняется быстрее. Кроме того, если `a` — это сложное выражение (например, элемент списка с вычисляемым индексом, таким как `items[offset + index]`), то комбинированная версия может оказаться менее подверженной ошибкам в случае, когда выражение, вычисляющее индекс, потребуется изменить, потому что программисту придется изменить всего одно выражение, а не два.

Язык Python перегружает (то есть позволяет использовать с данными разных типов) операторы + и += для строк и для списков. Первый из них выполняет операцию конкатенации, а второй — добавление для строк и расширение (добавление другого списка в конец) для списков:

```
>>> name = "John"
>>> name + "Doe"
'JohnDoe'
>>> name += " Doe"
>>> name
'John Doe'
```

Подобно целым числам, строки являются неизменяемыми, поэтому, когда используется оператор +=, создается новая строка, и ссылка, расположенная слева от оператора, связывается с новым объектом точно так же, как это было описано выше для случая с объектами типа `int`. Списки поддерживают аналогичный синтаксис, но за кулисами выполняются другие действия:

```
>>> seeds = ["sesame", "sunflower"]
>>> seeds += ["pumpkin"]
>>> seeds
['sesame', 'sunflower', 'pumpkin']
```

Поскольку списки относятся к категории изменяемых объектов, оператор += модифицирует существующий объект списка, поэтому повторной привязки ссылки `seeds` не происходит. Рис. 1.4 демонстрирует, как это происходит.

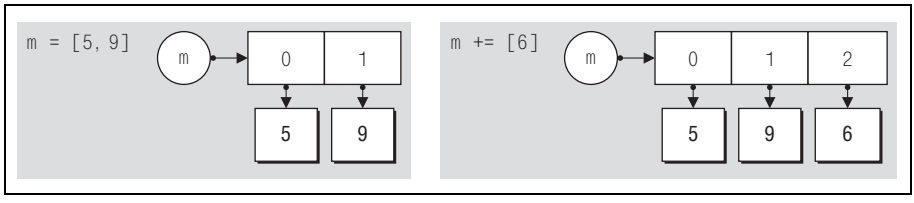

***Рис. 1.4.** Комбинированные операторы присваивания с изменяемыми объектами*

Если синтаксис языка Python так хитроумно скрывает различия между изменяемыми и неизменяемыми типами данных, то зачем вообще делаются такие различия? Причина главным образом заключается в обеспечении высокой производительности. Действия с неизменяемыми типами потенциально имеют более эффективную реализацию (так как они никогда не изменяются), чем действия с изменяемыми типами. Кроме того, некоторые типы коллекций, такие как множества, могут включать в себя только данные неизменяемых типов. С другой стороны, изменяемые типы удобнее в использовании. Случаи, когда различия между этими двумя категориями приобретают особую важность, мы будем рассматривать, например, в главе 4 – при обсуждении аргументов функций, имеющих значения по умолчанию; в главе 3 – при обсуждении списков, множеств и некоторых других типов данных; и в главе 6, где будет показано, как создавать свои собственные типы данных.

Правый операнд в операторе += для списков должен быть итерируемым объектом, в противном случае будет возбуждено исключение:

```
>>> seeds += 5
Traceback (most recent call last):
(Трассировочная информация (самый последний вызов внизу)
...
TypeError: 'int' object is not iterable
(TypeError: объект 'int' не является итерируемым)
```

Правильный способ расширения списка заключается в том, чтобы использовать итерируемый объект, например, список:

```
>>> seeds += [5]
>>> seeds
['sesame', 'sunflower', 'pumpkin', 5]
```

И, конечно же, сам итерируемый объект, за счет которого выполняется расширение списка, может содержать более одного элемента:

```
>>> seeds += [9, 1, 5, "poppy"]
>>> seeds
['sesame', 'sunflower', 'pumpkin', 5, 9, 1, 5, 'poppy']
```

Попытка добавить простую строку (например, `"durian"`), а не список, содержащий ее (`["durian"]`), приведет к логичному, но, возможно, неожиданному результату:

```
>>> seeds = ["sesame", "sunflower", "pumpkin"]
>>> seeds += "durian"
>>> seeds
['sesame', 'sunflower', 'pumpkin', 'd', 'u', 'r', 'i', 'a', 'n']
```

Оператор += списков расширяет список, добавляя к нему каждый элемент итерируемого объекта, находящегося справа, а поскольку строка – это итерируемый объект, то каждый символ строки будет добавлен как отдельный элемент. При использовании метода `append()` его аргумент всегда добавляется в список как единый элемент.

Составляющая №7: ввод/вывод

Чтобы писать по-настоящему полезные программы, нам необходимо иметь возможность читать входные данные – например, с клавиатуры или из файла и выводить результаты – либо на консоль, либо в файлы. Мы уже использовали встроенную функцию `print()`, но ее рассмотрение отложим до главы 4. В этом подразделе мы сосредоточим свое внимание на консольном вводе/выводе, а для чтения и записи файлов будем использовать механизм перенаправления командной оболочки.

В языке Python имеется встроенная функция `input()`, с помощью которой можно читать ввод пользователя. Эта функция принимает необязательный строковый аргумент (который выводится в консоли как строка приглашения к вводу) и ожидает, пока пользователь введет ответ и завершит ввод клавишей Enter (или Return). Если пользователь не введет никакой текст и просто нажмет клавишу Enter, функция `input()` вернет пустую строку, в противном случае она возвращает строку, содержащую ввод пользователя без символа завершения строки.

Ниже приводится первая «полезная» программа. В ней использовано большинство из уже описанных составляющих, единственное новое в ней – это вызов функции `input()`:

```
print("Type integers, each followed by Enter; or just Enter to finish")

total = 0
count = 0

while True:
    line = input("integer: ")
    if line:
        try:
            number = int(line)
        except ValueError as err:
            print(err)
            continue
        total += number
```

```
            count += 1
        else:
            break

    if count:
        print("count =", count, "total =", total, "mean =", total / count)
```

Примеры к книге, стр. 15

Эта программа (файл *sum1.py* в примерах к книге) состоит всего лишь из 17 строк выполняемого программного кода. Ниже приводятся результаты типичного сеанса работы с программой:

```
Type integers, each followed by Enter; or just Enter to finish
number: 12
number: 7
number: 1x
invalid literal for int() with base 10: '1x'
(неверный литерал типа int() по основанию 10: '1x')
number: 15
number: 5
number:
count = 4 total = 39 mean = 9.75
```

Несмотря на небольшие размеры, программа обладает достаточно высокой устойчивостью к ошибкам. Если пользователь введет строку, которая не может быть преобразована в целое число, проблема будет обнаружена обработчиком исключений, который выведет соответствующее сообщение и передаст управление в начало цикла (инструкция `continues`). А заключительная инструкция `if` предотвратит вывод статистической информации, если пользователь в течение сеанса не ввел ни одного числа, избежав тем самым выполнения операции деления на ноль.

Работа с файлами подробно будет описана в главе 7, а пока мы можем создавать файлы за счет простого перенаправления вывода функции `print()` средствами командной оболочки. Например, команда

```
C:\>test.py > results.txt
```

приведет к тому, что весь вывод, производимый простой функцией `print()`, которая вызывается в вымышленной программе *test.py*, будет записан в файл *results.txt*. Этот прием одинаково хорошо работает как в консоли Windows, так и в консоли UNIX. Если в системе Windows уже установлена версия Python 2 и она используется по умолчанию, то команда должна выглядеть так: `C:\Python30\python.exe test.py > results.txt`; если путь к интерпретатору Python 3 в переменной окружения `PATH` стоит первым (мы больше не будем напоминать об этом), то команда должна выглядеть так: `python test.py > results.txt`. В системах UNIX нам необходимо сделать программу выполняемой (`chmod +x test.py`) и затем вызвать ее командой `./test.py`, если каталог, в котором она находится, не содержится в переменной окружения `PATH`.

Чтение данных можно реализовать за счет перенаправления файла с данными на вход программы – по аналогии с перенаправлением вывода. Однако если использовать перенаправление ввода с программой *sum1.py*, она завершится с ошибкой. Это обусловлено тем, что функция `input()` возбуждает исключение, когда принимает символ EOF (end of file – конец файла). Ниже приводится более устойчивая версия (*sum2.py*), которая способна принимать ввод с клавиатуры или за счет перенаправления файла:

```
print("Type integers, each followed by Enter; or ^D or ^Z to finish")

total = 0
count = 0

while True:
    try:
        line = input()
        if line:
            number = int(line)
            total += number
            count += 1
    except ValueError as err:
        print(err)
        continue
    except EOFError:
        break

if count:
    print("count =", count, "total =", total, "mean =", total / count)
```

Если выполнить команду `sum2.py < data\sum2.dat` (где *sum2.dat* – это файл, содержащий список чисел, по одному числу в строке, и находящийся в каталоге *data* в примерах к книге), на консоль будет выведено:

```
Type integers, each followed by Enter; or ^D or ^Z to finish
count = 37 total = 1839 mean = 49.7027027027
```

Мы внесли в программу некоторые незначительные изменения, чтобы она могла принимать ввод как в интерактивном режиме, так и из перенаправляемого файла. Во-первых, мы изменили признак завершения программы с пустой строки на символ EOF (`Ctrl+D` – в UNIX, `Ctrl+Z`, `Enter` – в Windows). Это сделало программу более устойчивой при работе с файлами, содержащими пустые строки. Во-вторых, мы больше не выводим строку подсказки перед вводом каждого числа, поскольку в этом нет никакого смысла при перенаправлении ввода. И в-третьих, мы используем единый блок `try` с двумя обработчиками исключений.

Обратите внимание: в случае ввода неправильного целого числа (с клавиатуры или в случае «ошибочной» строки в перенаправляемом файле) преобразование `int()` возбудит исключение `ValueError` и управление немедленно будет передано соответствующему блоку `except`. Это озна-

чает, что значения total и count не будут увеличены при встрече с ошибочными данными, то есть именно то, что нам и требуется.

Точно так же мы могли бы использовать два отдельных блока try:

```
while True:
    try:
        line = input()
        if line:
            try:
                number = int(line)
            except ValueError as err:
                print(err)
                continue
            total += number
            count += 1
    except EOFError:
        break
```

Но мы предпочли сгруппировать обработку исключений в конце, чтобы не загромождать лишними конструкциями основной блок обработки.

Составляющая №8: создание и вызов функций

Вполне возможно написать программу, пользуясь исключительно типами данных и управляющими структурами, которые мы рассмотрели в предыдущих подразделах. Однако очень часто бывает необходимо повторно выполнять одну и ту же обработку, но с некоторыми отличиями в начальных значениях. Язык Python предоставляет возможность оформления блоков программного кода в виде функций, параметризуемых аргументами, которые им передаются. Ниже приводится общий синтаксис определения функции:

```
def functionName(arguments):
    suite
```

Инструкция return, стр. 205

Часть *arguments* не является обязательной; для передачи нескольких аргументов их необходимо отделить друг от друга запятыми. Любая функция в языке Python возвращает некоторое значение – по умолчанию это значение None; если мы явно не возвращаем значение с помощью инструкции return *value*, где *value* – это возвращаемое значение. Возвращаемое значение может быть единственным значением или кортежем возвращаемых значений. Возвращаемое значение может игнорироваться вызывающей программой, в этом случае оно просто уничтожается.

Примечательно, что инструкция def действует как оператор присваивания. При выполнении инструкции def создаются новый объект-

функция и ссылка на объект с указанным именем, которая указывает на объект-функцию. Поскольку функции – это объекты, они могут сохраняться в виде элементов коллекций и передаваться в качестве аргументов другим функциям, в чем вы сможете убедиться в последующих главах.

Часто в интерактивных консольных приложениях возникает необходимость получить целое число от пользователя. Ниже приводится функция, которая выполняет эту операцию:

```
def get_int(msg):
    while True:
        try:
            i = int(input(msg))
            return i
        except ValueError as err:
            print(err)
```

Эта функция принимает единственный аргумент, `msg`. Внутри цикла `while` пользователю предлагается ввести целое число. Если он вводит что-то неправильное, возбуждается исключение `ValueError`. В этом случае выводится сообщение об ошибке, и цикл повторяется. После ввода правильного целого числа оно возвращается вызывающей программе. Ниже показано, как можно использовать эту функцию:

```
age = get_int("enter your age: ")
```

В этом примере единственный параметр является обязательным, потому что мы не предусматриваем значение по умолчанию для него. В действительности язык Python предоставляет очень сложный и гибкий синтаксис для описания параметров функции, включая значения аргументов по умолчанию, а также позиционные и именованные аргументы. Полный синтаксис объявления функций будет рассматриваться в главе 4.

Создавая собственные функции, мы можем удовлетворить любые наши потребности, но во многих случаях в этом нет необходимости, потому что в составе языка Python имеется масса встроенных функций и намного больше функций содержится в модулях стандартной библиотеки. Поэтому многое из того, что нам может потребоваться, уже существует.

Модуль в языке Python – это обычный файл *.py* с программным кодом на языке Python, содержащим определения функций и классов (нестандартных типов данных) и некоторых переменных. Чтобы получить доступ к функциональным возможностям модуля, его сначала необходимо импортировать. Например:

```
import sys
```

Для импортирования модуля используется инструкция `import`, за которой следует имя файла *.py*, но без расширения.[1] Как только модуль будет импортирован, мы можем обращаться к его функциям, классам или переменным, содержащимся в нем. Например:

```
print(sys.argv)
```

Модуль `sys` содержит переменную `argv` – список, первым элементом которого является имя, под которым была вызвана программа, а вторым и последующими элементами – аргументы командной строки. Две вышеприведенные строки составляют целую программу *echoargs.py*. Если вызвать эту программу командой `echoargs.py -v`, она выведет `['echoargs.py', '-v']`. (В системах UNIX первым элементом может быть строка `'./echoargs.py'`.)

Оператор точки, стр. 35

В общем случае синтаксис обращения к функции из модуля выглядит так: *moduleName.functionName*(*arguments*). Здесь используется оператор точки («доступ к атрибуту»), который был представлен в составляющей №3. Стандартная библиотека содержит огромное число модулей, и многие из них мы будем использовать в этой книге. Имена всех стандартных модулей состоят только из символов нижнего регистра, поэтому для отличия своих модулей при именовании модулей многие программисты используют прием чередования регистра символов (например, `MyModule`).

В качестве примера приведем модуль `random` (в стандартной библиотеке хранится в файле *random.py*), содержащий множество полезных функций:

```
import random
x = random.randint(1, 6)
y = random.choice(["apple", "banana", "cherry", "durian"])
```

После выполнения этих инструкций переменная `x` будет содержать целое число в диапазоне от 1 до 6 включительно, а переменная `y` – одну из строк из списка, переданного функции `random.choice()`.

Shebang – строка (#!), стр. 25

Как правило, все инструкции `import` помещаются в начало файлов *.py*, после строки shebang и после комментариев с описанием модуля. (Порядок описания модулей рассматривается в главе 5.) Мы рекомендуем сначала импортировать модули из стандартной библиотеки, затем модули сторонних разработчиков, а потом ваши собственные.

---

[1] Модуль `sys`, некоторые другие модули и модули, реализованные на языке C, не обязательно должны храниться в виде файлов с расширением *.py*, но используются они точно так же.

Примеры

В предыдущем разделе мы узнали достаточно много о языке Python, чтобы иметь возможность приступить к созданию своих программ. В этом разделе мы рассмотрим две законченные программы, которые используют только те возможности языка Python, которые были описаны выше. Примеры должны продемонстрировать уже доступные возможности и помочь закрепить полученные знания.

В последующих главах мы узнаем значительно больше о языке Python и о библиотеке, благодаря чему мы сможем писать более короткие и более надежные программы, чем те, что представлены здесь. Но для начала необходимо заложить фундамент, на котором будет строиться остальное здание.

bigdigits.py

Первая программа, которую мы рассмотрим, очень короткая, хотя в ней имеется несколько интересных особенностей, включая список списков. Эта программа принимает число в виде аргумента командной строки и выводит его на экран, используя «большие» цифры.

На серверах, где множество пользователей совместно пользуются высокоскоростным принтером, обычной практикой считается, когда задание для печати каждого пользователя предваряется титульным листом, на котором с помощью описываемой методики выводится имя пользователя и некоторая дополнительная идентификационная информация.

Мы будем рассматривать программный код в три этапа: импортирование, создание списков, хранящих данные, используемые программой, и собственно обработка. Но для начала посмотрим пример вывода:

```
bigdigits.py 41072819
   *    *    ***   *****   ***    ***    *   ****
  **   **   *   *      *  *   *  *   *  **  *   *
 * *    *  *     *    *   *   *  *   *   *  *   *
*  *    *  *     *   *       *    ***    *   ****
******  *  *     *  *       *    *   *   *      *
   *    *   *   *  *       *     *   *   *      *
   *   ***   ***   *      *****   ***   ***     *
```

Мы не показали здесь строку приглашения к вводу в консоли (или ведущую комбинацию `./` для пользователей UNIX) – к настоящему времени мы считаем их наличие чем-то подразумеваемым.

```
import sys
```

Поскольку нам потребуется читать аргумент командной строки (выводимое число), нам необходимо будет обращаться к списку `sys.argv`, поэтому начнем с того, что импортируем модуль `sys`.

Каждая цифра будет представлена списком строк. Например, ниже показано представление цифры 0:

```
Zero = ["  ***   ",
        " *   *  ",
        "*     *",
        "*     *",
        "*     *",
        " *   * ",
        "  ***  "]
```

Тип set, стр. 144

Тип dict, стр. 151

Обратите внимание, что список строк `Zero` располагается в нескольких строках. Обычно инструкции языка Python занимают одну строку, но они могут занимать и несколько строк, если они представлены выражением в скобках. Литералы списков, множеств, словарей, список аргументов функции или многострочные инструкции, в которых символ конца строки экранируется символом обратного слеша (\) – во всех этих случаях инструкции могут занимать несколько строк, при этом отступы во второй и последующих строках не учитываются.

Каждый список, представляющий цифру, содержит семь строк, все одинаковой длины, причем эта длина может изменяться от цифры к цифре. Списки с изображением остальных цифр строятся аналогичным образом, но они выведены несколько иначе – для компактности, а не для ясности:

```
One = [" * ", "** ", " * ", " * ", " * ", " * ", "***"]
Two = [" *** ", "*   *", "*  * ", "   *  ", " *   ", "*    ", "*****"]
# ...
Nine = [" ****", "*   *", "*   *", " ****", "    *", "    *", "    *"]
```

Последний элемент данных, который нам потребуется, – это список всех списков цифр:

```
Digits = [Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine]
```

Мы могли бы создать список списков `Digits`, не создавая промежуточные переменные. Например:

```
Digits = [
    ["  ***  ", " *   * ", "*     *", "*     *", "*     *",
     " *   * ", "  ***  "], # Zero
    [" * ", "** ", " * ", " * ", " * ", " * ", "***"], # One
    # ...
    [" ****", "*   *", "*   *", " ****", "    *", "    *",
     "    *"] # Nine
    ]
```

Мы предпочли использовать отдельные переменные для каждого числа для удобочитаемости и потому, что вариант на основе переменных выглядит более аккуратным.

Далее приведена остальная часть программного кода, чтобы вы могли ознакомиться с ней, прежде чем продолжить читать пояснения.

```
try:
    digits = sys.argv[1]
    row = 0
    while row < 7:
        line = ""
        column = 0
        while column < len(digits):
            number = int(digits[column])
            digit = Digits[number]
            line += digit[row] + "  "
            column += 1
        print(line)
        row += 1
except IndexError:
    print("usage: bigdigits.py <number>")
except ValueError as err:
    print(err, "in", digits)
```

Программный код полностью обернут в конструкцию обработки исключений, которая способна перехватывать два вида исключений, когда что-то пойдет не так. Сначала мы пытаемся извлечь параметр командной строки. Индексация элементов в списке `sys.argv` начинается с нуля, как и в любых других списках в языке Python. Элемент списка с индексом 0 – это имя программы, под которым она была вызвана, поэтому данный список в любой программе имеет как минимум один элемент. Если при вызове программы аргумент не был указан, то при попытке обратиться ко второму элементу списка, состоящему из одного элемента, будет возбуждено исключение `IndexError`. В этом случае управление немедленно будет передано соответствующему блоку обработки исключений, где мы просто выводим информацию о порядке использования программы. После этого выполнение программы продолжается за концом блока `try`, но так как за его пределами больше нет программного кода, то программа просто завершается.

Если исключение `IndexError` не возбуждается, в строку `digits` записывается аргумент командной строки, которая, как мы надеемся, состоит из одних цифр. (Вспомните, как в описании составляющей №2 мы говорили, что идентификаторы в языке Python чувствительны к регистру символов, поэтому `digits` и `Digits` – это разные идентификаторы.) Каждая большая цифра представлена семью строками, и чтобы правильно вывести число, мы должны сначала вывести верхние строки всех цифр, затем следующие строки всех цифр и т. д., пока не будут выведены все семь строк. Для обхода всех строк мы используем цикл `while`. Вместо него можно было бы использовать цикл `for row in (0, 1, 2, 3, 4, 5, 6):`,

Функция range(), стр. 167

а позднее будет представлен более интересный способ, основанный на использовании встроенной функции `range()`.

Переменная `line` используется для сборки строки из отдельных строк всех имеющихся цифр. Далее выполняется обход в цикле всех колонок, то есть всех цифр в аргументе командной строки. Мы извлекаем каждый символ, используя выражение `digits[column]`, и преобразуем полученную цифру в целое число `number`. Если при преобразовании возникает ошибка, возбуждается исключение `ValueError` и управление немедленно передается соответствующему обработчику ошибок. В этом случае мы выводим сообщение об ошибке, и выполнение продолжается за пределами блока `try`. Как уже отмечалось ранее, поскольку за его пределами больше нет программного кода, то программа просто завершается.

В случае благополучного преобразования значение `number` используется как индекс в списке `Digits`, из которого извлекается список строк `digit`. Затем мы добавляем строку с индексом `row` из этого списка в создаваемую строку `line` и добавляем два пробела, чтобы отделить цифры друг от друга.

Каждый раз, когда внутренний цикл `while` завершает работу, мы выводим собранную строку. Ключом к пониманию этой программы является фрагмент кода, где выполняется сборка строки `line` из строк отдельных цифр. Попробуйте запустить эту программу, чтобы получить представление о том, как она работает. Мы еще вернемся к этой программе в упражнениях, чтобы попытаться несколько улучшить вывод.

generate_grid.py

Часто бывает необходимо сгенерировать тестовые данные. Не существует какой-либо универсальной программы, которая делала бы это, поскольку требования к тестовым данным могут изменяться очень сильно. Язык Python часто используется для создания массивов тестовых данных, потому что программы на языке Python писать и модифицировать не составляет большого труда. В этом подразделе мы создадим программу, которая будет генерировать матрицу из случайных целых чисел, где пользователь сможет указать число строк и столбцов, а также диапазон целых чисел. Для начала рассмотрим пример запуска программы:

```
generate_grid.py
rows: 4x
invalid literal for int() with base 10: '4x'
(ошибочный литерал типа int() по основанию 10: '4x')
rows: 4
columns: 7
minimum (or Enter for 0): -100
maximum (or Enter for 1000):
       554       720       550       217       810       649       912
```

```
-24     908     742     -65     -74     724     825
711     968     824     505     741      55     723
180     -60     794     173     487       4     -35
```

Программа работает в интерактивном режиме, и при первой попытке мы допустили ошибку при вводе числа строк. Программа ответила выводом сообщения об ошибке и затем попросила повторить ввод числа строк. При запросе ввести максимальное число мы просто нажали клавишу Enter, чтобы использовать число по умолчанию.

Мы будем рассматривать программный код в четыре приема: импорт, определение функции `get_int()` (более сложная версия, чем была показана в составляющей №8), взаимодействие с пользователем, во время которого принимаются вводимые числа, и собственно обработка.

```
import random
```

Чтобы получить доступ к функции `random.randint()`, нам необходимо импортировать модуль `random`.

Функция random.randint(), стр. 54

```
def get_int(msg, minimum, default):
    while True:
        try:
            line = input(msg)
            if not line and default is not None:
                return default
            i = int(line)
            if i < minimum:
                print("must be >=", minimum)
            else:
                return i
        except ValueError as err:
            print(err)
```

Функция требует три аргумента: строку приглашения к вводу, минимальное значение и значение по умолчанию. Если пользователь просто нажимает клавишу Enter, у функции имеется две возможности. Если аргумент `default` имеет значение `None`, то есть значение по умолчанию не задано, то управление передается строке `i = int(line)`. Попытка преобразования терпит неудачу (потому что пустая строка не может быть преобразована в целое число), и возбуждается исключение `ValueError`. Но если аргумент `default` имеет значение, отличное от `None`, оно возвращается вызывающей программе. В противном случае функция попытается преобразовать в целое число текст, введенный пользователем, и если преобразование будет выполнено благополучно, функция проверит, чтобы введенное число было не меньше аргумента `minimum`.

Таким образом, функция всегда будет возвращать либо значение аргумента `default` (если пользователь просто нажмет клавишу Enter), либо целое число, большее или равное значению аргумента `minimum`.

```
rows = get_int("rows: ", 1, None)
columns = get_int("columns: ", 1, None)
```

```
minimum = get_int("minimum (or Enter for 0): ", -1000000, 0)

default = 1000
if default < minimum:
    default = 2 * minimum
maximum = get_int("maximum (or Enter for " + str(default) + "): ",
                  minimum, default)
```

Наша функция `get_int()` упрощает получение числа строк и столбцов, а также минимальное и максимальное значения желаемого диапазона случайных чисел. Для числа столбцов и строк мы передаем в аргументе `default` значение `None`, что означает отсутствие значения по умолчанию, то есть пользователь должен ввести целое число. При запросе минимального значения мы указали значение по умолчанию 0, а при запросе максимального значения выбирается значение по умолчанию 1000, или в два раза больше минимального значения, если минимальное значение окажется больше или равно 1000.

Как уже отмечалось в предыдущем примере, список аргументов при вызове функции может занимать любое число строк в исходном программном коде, при этом величина отступов во второй и последующих строках не будет иметь значения.

Получив от пользователя число строк и столбцов, а также желательные минимальное и максимальное значения случайных чисел, можно приступать к работе.

```
row = 0
while row < rows:
    line = ""
    column = 0
    while column < columns:
        i = random.randint(minimum, maximum)
        s = str(i)
        while len(s) < 10:
            s = " " + s
        line += s
        column += 1
    print(line)
    row += 1
```

Для создания матрицы мы используем три цикла `while`. Внешний цикл обрабатывает строки, средний – столбцы и внутренний – символы. В среднем цикле мы получаем случайное число из указанного диапазона и преобразуем его в строку. Внутренний цикл `while` обеспечивает дополнение строки ведущими пробелами так, чтобы каждое число было представлено строкой из 10 символов. Строка `line` используется для накопления чисел в строке матрицы, которая выводится после того, как будут добавлены числа для всех колонок. На этом мы завершаем наш второй пример.

Язык Python предоставляет весьма широкие возможности по форматированию строк, а также обеспечивает поддержку цикла `for ... in`. Поэтому в более реалистичных версиях *bigdigits.py* и *generate_grid.py* можно было бы использовать циклы `for ... in`, а в *generate_grid.py* можно было бы также использовать возможности языка Python по форматированию строк вместо неуклюжего приема дополнения пробелами. Но мы ограничили себя восемью составляющими языка Python, представленными в этой главе, которых оказалось вполне достаточно для написания законченных и полезных программ. В каждой последующей главе мы будем знакомиться с новыми особенностями языка Python, поэтому по мере продвижения вперед мы будем видеть (и обретать способность писать) все более сложные программы.

Метод `str.format()`, стр. 100

В заключение

В этой главе мы узнали, как редактировать и запускать программы на языке Python. и рассмотрели пару маленьких, но законченных программ. Но большая часть главы была посвящена восьми составляющим «золотого запаса» языка Python, знания которых вполне достаточно для создания настоящих программ.

Мы начали с двух основных типов данных в языке Python – `int` и `str`. Целочисленные литералы записываются точно так же, как и во многих других языках программирования. Строковые литералы записываются либо с помощью кавычек, либо с помощью апострофов. Неважно, что будет использоваться – кавычки или апострофы, главное, чтобы с обоих концов литерала использовался один и тот же тип кавычек. У нас имеется возможность выполнять преобразования между строками и целыми числами, например, `int("250")` и `str(125)`. Если преобразование строки в целое число терпит неудачу, возбуждается исключение `ValueError`; с другой стороны, в строку может быть преобразовано практически все, что угодно.

Строки – это последовательности, поэтому те функции и операции, которые могут применяться к последовательностям, могут применяться и к строкам. Например, обратиться к определенному символу можно с помощью оператора доступа (`[]`), конкатенацию строк можно выполнить с помощью оператора `+`, а дополнение одной строки другой – с помощью оператора `+=`. Так как строки являются неизменными объектами, операция дополнения строки за кулисами создает новую строку, представляющую собой результат конкатенации заданных строк, и перепривязывает ссылку на строковый объект, указанный слева от знака =, на результирующую строку. Мы также имеем возможность выполнять итерации по символам в строке, используя цикл `for ... in`,

а чтобы определить количество символов в строке, можно использовать функцию `len()`.

При использовании неизменяемых объектов, таких как строки, целые числа и кортежи, мы можем писать программный код, как если бы ссылки на объекты были переменными, то есть как если бы ссылка на объект была самим объектом, на который она ссылается. Точно так же можно поступать и в случае изменяемых объектов, только в этом случае изменения в объекте будут затрагивать все ссылки, указывающие на этот объект, – эта проблема будет описываться в главе 3.

В языке Python имеется несколько встроенных типов коллекций, а также ряд типов коллекций имеется в стандартной библиотеке. Мы познакомились с типами `list` и `tuple`, в частности, мы узнали, как создавать кортежи и списки из литералов, например, `even = [2, 4, 6, 8]`. Списки, как и все остальное в языке Python, являются объектами, поэтому мы можем пользоваться их методами, например, вызов метода `even.append(10)` добавит дополнительный элемент в список. Подобно строкам, списки и кортежи являются последовательностями, благодаря чему можно выполнять итерации по их элементам, используя цикл `for ... in`, и определять количество элементов в коллекции с помощью функции `len()`. Кроме того, имеется возможность извлекать из списков или кортежей отдельные элементы, используя оператор доступа к элементам (`[]`), объединять два списка или кортежа с помощью оператора `+` и дополнять один список другим с помощью оператора `+=`. Если потребуется добавить в конец списка единственный элемент, можно использовать метод `list.append()` или оператор `+=`, которому следует передать список, состоящий из одного элемента, например, `even += [12]`. Поскольку списки являются изменяемыми объектами, для изменения отдельных элементов можно использовать оператор `[]`, например, `even[1] = 16`.

Быстрые операторы `is` и `not is` удобно использовать, чтобы выяснить, не ссылаются ли две ссылки на один и тот же объект, что очень удобно, особенно когда выполняется проверка ссылки на уникальный встроенный объект `None`. В вашем распоряжении имеются все обычные операторы сравнения (`<`, `<=`, `==`, `!=`, `>=`, `>`), но они могут использоваться только с совместимыми типами данных и только с теми, которые поддерживают эти операции. Мы пока познакомились только с типами данных `int`, `str`, `list` и `tuple` – все они поддерживают полный набор операторов сравнения. Попытка сравнения несовместимых типов, например, сравнение типа `int` с типом `str` или `list`, будет приводить к возбуждению исключения `TypeError`.

Язык Python поддерживает стандартные логические операторы `and`, `or` и `not`. Выражения с участием операторов `and` и `or` вычисляются по сокращенной схеме и возвращают операнд, определяющий результат выражения, причем результат не обязательно будет иметь тип Boolean

(хотя и может быть приведен к типу Boolean). Оператор `not` всегда дает в результате либо `True`, либо `False`.

У нас имеется возможность проверить вхождение элементов в последовательности, включая строки, списки и кортежи, используя операторы `in` и `not in`. Проверка на вхождение в списки и кортежи производится с использованием медленного алгоритма линейного поиска, а проверка вхождения в строки реализует потенциально более скоростной гибридный алгоритм, но производительность в этих случаях редко является проблемой, за исключением случаев использования очень длинных строк, списков и кортежей. В главе 3 мы познакомимся с такими типами коллекций, как ассоциативные массивы и множества, для которых операция проверки на вхождение выполняется очень быстро. Кроме того, с помощью функции `type()` можно определить тип объекта, на который указывает ссылка, но обычно эта функция используется только для нужд тестирования и отладки.

Язык Python предоставляет несколько управляющих структур, включая условный оператор `if … elif … else`, цикл с предусловием `while`, цикл по последовательности `for ... in` и конструкцию обработки исключений `try ... except`. Имеется возможность преждевременно прерывать циклы `while` и `for ... in` с помощью инструкции `break` или передавать управление в начало цикла с помощью инструкции `continue`.

В языке Python имеется поддержка обычных арифметических операторов, включая `+`, `-`, `*` и `/`, единственная необычность состоит в том, что оператор `/` всегда возвращает число с плавающей точкой, даже если оба операнда являются целыми числами. (Целочисленное деление, имеющееся во многих других языках программирования, реализовано и в языке Python – в виде оператора `//`.) Кроме того, язык Python предоставляет комбинированные операторы присваивания, такие как `+=` и `*=`. Они за кулисами создают новые объекты, если слева находится неизменяемый операнд. Как уже отмечалось ранее, арифметические операторы перегружены для применения к операндам типов `str` и `list`.

Консольный ввод/вывод можно реализовать с помощью функций `input()` и `print()`, а благодаря возможности перенаправлять ввод/вывод в файлы, мы можем использовать те же самые встроенные функции для чтения и записи файлов.

В дополнение к богатому набору встроенных функциональных возможностей имеется обширная стандартная библиотека; модули становятся доступными после импортирования их с помощью инструкции `import`.

Одним из наиболее часто импортируемых модулей является модуль `sys`, в котором имеется список `sys.argv`, хранящий аргументы командной строки. Если в языке Python отсутствует какая-либо необходимая нам функция, с помощью инструкции `def` мы легко можем создать свою собственную функцию, действующую так, как нам нужно.

Используя функциональные возможности, описанные в этой главе, уже можно писать короткие и полезные программы на языке Python. В следующей главе мы больше узнаем о типах данных в языке Python, более подробно рассмотрим типы `int` и `str`, а также познакомимся с некоторыми совершенно новыми типами данных. В главе 3 мы больше узнаем о кортежах, списках, а также о некоторых других типах коллекций в языке Python. Затем в главе 4 мы более подробно рассмотрим управляющие структуры языка Python и узнаем, как создавать свои функции, позволяющие избежать дублирования программного кода и способствующие многократному его использованию.

Упражнения

Примеры к книге, стр. 15

Цель упражнений, которые приводятся здесь и будут приводиться на протяжении всей книги, состоит в том, чтобы стимулировать вас на проведение экспериментов с языком Python и помочь получить практический опыт с одновременным закреплением пройденного материала. В примерах и упражнениях рассматриваются проблемы числовой обработки и обработки текста, что может интересовать самую широкую аудиторию, а кроме того, размеры упражнений настолько невелики, что при их решении вам придется главным образом думать и учиться, а не просто вводить программный код. Решение для каждого упражнения можно найти в примерах книги.

1. Было бы довольно интересно написать версию программы *bigdigits.py*, которая для рисования цифр использовала бы не символ *, а соответствующие цифровые символы. Например:

```
bigdigits_ans.py 719428306
77777   1   9999     4     222    888    333     000     666
    7  11  9   9    44    2   2  8   8  3   3   0   0   6
   7    1  9   9   4 4    2   2  8   8      3  0     0  6
  7     1   9999  4  4      2     888     33   0     0  6666
 7      1      9  444444   2     8   8      3  0     0  6   6
7       1      9     4    2      8   8  3   3   0   0   6   6
7      111     9     4    22222   888    333     000     666
```

Эта задача может быть решена двумя способами. Самый простой способ заключается в том, чтобы просто заменить символы * в списках. Но этот путь не слишком гибкий, и хотелось бы, чтобы вы пошли другим путем. Попробуйте изменить программный код так, чтобы вместо добавления в строку за один проход целых строк (`digit[row]`), вырезанных из изображений цифр, в строку добавлялись бы символ за символом, и при встрече символа * он заменялся бы соответствующим цифровым символом.

Сделать это можно, скопировав исходный программный код из *bigdigits.py* и изменив пять строк. Это упражнение не столько сложное, сколько с подвохом. Решение приводится в файле *bigdigits_ans.py*.

2. Среда разработки IDLE может использоваться как мощный калькулятор, но иногда бывает удобно иметь калькулятор, специализированный для решения определенного круга задач. Напишите программу, которая в цикле `while` предлагала бы пользователю ввести число, постепенно накапливая список введенных чисел. Затем, когда пользователь завершит работу с программой (простым нажатием клавиши Enter), она выводила бы числа, введенные пользователем, количество введенных чисел, их сумму, наименьшее и наибольшее число и среднее значение (сумма / количество). Ниже приводится пример сеанса работы с программой:

```
average1_ans.py
enter a number or Enter to finish: 5
enter a number or Enter to finish: 4
enter a number or Enter to finish: 1
enter a number or Enter to finish: 8
enter a number or Enter to finish: 5
enter a number or Enter to finish: 2
enter a number or Enter to finish:
numbers: [5, 4, 1, 8, 5, 2]
count = 6 sum = 25 lowest = 1 highest = 8 mean = 4.16666666667
```

Решение этого упражнения потребует примерно четыре строки для инициализации необходимых переменных (пустой список – это просто литерал `[]`) и не более 15 строк для цикла `while`, включая обработку ошибок. Для вывода результатов в конце потребуется всего пара строк, поэтому вся программа, включая пустые строки для лучшей читаемости, должна уместиться примерно в 25 строк.

3. В некоторых ситуациях нам может потребоваться сгенерировать тестовый текст, который пригодится, например, при разработке дизайна веб-сайта, когда действительное содержимое еще отсутствует, или при разработке программы составления отчетов. Напишите программу, которая создавала бы жуткие поэмы (способные посрамить поэзию Вогона (Vogon)).

Создайте списки слов, например, артиклей («the», «a» и других), имен существительных («cat», «dog», «man», «woman»), глаголов («sang», «ran», «jumped») и наречий («loudly», «quietly», «well», «badly»). Затем выполните пять циклов и на каждой итерации с помощью функции `random.choice()` выберите артикль, существительное, глагол и наречие. С помощью функции `random.randint()` выберите одну из двух структур предложений: артикль, существительное,

Функции `random.randint()` и `random.choice()`, стр. 54

глагол и наречие, или артикль, существительное и глагол, — и выведите предложение. Ниже приводится пример запуска такой программы:

```
awfulpoetry1_ans.py
her man heard politely
his boy sang
another woman hoped
her girl sang slowly
the cat heard loudly
```

Для решения этого упражнения вам потребуется импортировать модуль `random`. Списки могут занимать порядка 4–10 строк, в зависимости от того, как много слов вы подберете для каждого из них, и сам цикл будет занимать не более 10 строк, поэтому вся программа, включая пустые строки для лучшей читаемости, должна уместиться примерно в 20 строк. Решение приводится в файле *awfulpoetry1_ans.py*.

4. Чтобы сделать поэтическую программу более универсальной, добавьте в нее программный код, дающий пользователю возможность определить количество выводимых строк (от 1 до 10 включительно), передавая число в виде аргумента командной строки. Если программа вызывается без аргумента, она должна по умолчанию выводить пять строк, как и раньше. Для решения этого упражнения вам потребуется изменить главный цикл (это может быть цикл `while`). Не забывайте, что операторы сравнения в языке Python могут объединяться в цепочки, поэтому здесь вам не потребуется использовать логический оператор `and` при проверке вхождения аргумента командной строки в заданный диапазон. Функциональность программы может быть расширена за счет приблизительно десяти строк программного кода. Решение приводится в файле *awfulpoetry2_ans.py*.
5. В программе из упражнения 2 было бы неплохо реализовать нахождение не только среднего значения, но и медианы, но для этого придется отсортировать список. Сортировка списков в языке Python легко осуществляется с помощью метода `list.sort()`, но мы еще не рассматривали этот метод, поэтому вам не следует использовать его. Дополните программу вычисления среднего значения программным кодом, который сортировал бы список чисел. Эффективность не имеет значения, поэтому используйте самый простой способ сортировки, какой только придет вам на ум. Отсортировав список, можно будет найти и медиану, которая будет являться значением элемента в середине, если список содержит нечетное число элементов, и средним значением от двух средних элементов, если список содержит четное число элементов. Найдите медиану и выведите ее вместе с остальной информацией.

Решение этого упражнения может оказаться не совсем простым делом, особенно для неопытных программистов. Даже если у вас имеется опыт работы с языком Python, вы все равно можете столкнуться с трудностями, так как вы ограничены только тем кругом возможностей, которые мы рассмотрели в этой главе. Реализация сортировки займет примерно дюжину строк, и вычисление медианы (нельзя использовать оператор деления по модулю, так как он еще не рассматривался) еще четыре строки. Решение приводится в файле *average2_ans.py*.

2

- Идентификаторы и ключевые слова
- Целочисленные типы
- Числа с плавающей точкой
- Строки

Типы данных

В этой главе мы приступаем к более подробному изучению языка Python. Для начала мы обсудим правила создания имен, которые мы даем ссылкам на объекты, и познакомимся со списком ключевых слов языка Python. Затем мы рассмотрим наиболее важные типы данных, исключая коллекции, которые будут рассматриваться в главе 3. Типы данных считаются встроенными за исключением тех, что определены в стандартной библиотеке. Единственное отличие встроенных типов данных от библиотечных состоит в том, что, прежде чем воспользоваться последними, нам необходимо импортировать соответствующие модули и мы должны квалифицировать имена типов именами модулей, в которых они определяются. Более подробно об импортировании мы поговорим в главе 5.

Идентификаторы и ключевые слова

Ссылки на объекты, стр. 29

Создавая элемент данных, мы можем либо присвоить его переменной, либо вставить в коллекцию. (Как уже отмечалось в предыдущей главе, когда в языке Python выполняется операция присваивания, в действительности происходит связывание ссылки на объект с объектом в памяти, который хранит данные.) Имена, которые даются ссылкам на объекты, называются *идентификаторами*, или просто *именами*.

Допустимый идентификатор в языке Python – это последовательность символов произвольной длины, содержащей «начальный символ» и ноль или более «символов продолжения». Такой идентификатор должен следовать определенным правилам и соглашениям.

Первое правило касается начального символа и символов продолжения. Начальным символом может быть любой символ, который в кодировке Юникод рассматривается как принадлежащий диапазону алфавитных символов ASCII («a», «b», ..., «z», «A», «B», ..., «Z»), символ подчеркивания («_»), а также символы большинства национальных (не английских) алфавитов. Каждый символ продолжения может быть любым символом из тех, что пригодны в качестве начального символа, а также любым непробельным символом, включая символы, которые в кодировке Юникод считаются цифрами, такие как («0», «1», ..., «9»), и символ Каталана «·». Идентификаторы чувствительны к регистру, поэтому `TAXRATE`, `Taxrate`, `TaxRate`, `taxRate` и `taxrate` – это пять разных идентификаторов.

Точный перечень символов, допустимых для использования в качестве начального символа и символов продолжения, описывается в документации по языку Python (справочник «Language reference», раздел «Lexical analysis», подраздел «Identifiers and keywords»[1]) или в PEP 3131[2] (раздел «Supporting Non-ASCII Identifiers»).

Второе правило гласит, что идентификатор не должен совпадать с каким-либо из ключевых слов языка Python, поэтому мы не можем использовать имена, которые приводятся в табл. 2.1.

Таблица 2.1. Ключевые слова языка Python

`and`	`continue`	`except`	`global`	`lambda`	`pass`	`while`
`as`	`def`	`False`	`if`	`None`	`raise`	`with`
`assert`	`del`	`finally`	`import`	`nonlocal`	`return`	`yield`
`break`	`elif`	`for`	`in`	`not`	`True`	
`class`	`else`	`from`	`is`	`or`	`try`	

С многими из них мы уже встречались в предыдущей главе, хотя 11 ключевых слов – `assert`, `class`, `del`, `finally`, `from`, `global`, `lambda`, `nonlocal`, `raise`, `with` и `yield` мы еще не рассматривали.

Первое соглашение выглядит так: «Не использовать в качестве своих идентификаторов любые предопределенные имена». Поэтому старай-

---

[1] *http://docs.python.org/3.0/reference/lexical_analysis.html# identifiers-and-keywords. – Прим. перев.*

[2] Аббревиатура «PEP» расшифровывается как Python Enhancement Proposal (предложение по расширению Python). Если кто-то желает изменить или дополнить язык Python, и его стремление пользуется широкой поддержкой сообщества, он посылает PEP с подробным описанием своего предложения, чтобы его можно было рассмотреть в официальном порядке; в некоторых случаях, как это произошло с PEP 3131, предложение принимается и реализуется. Все предложения PEP можно найти на странице *www.python.org/dev/peps/*.

тесь не использовать такие идентификаторы, как `NotImplemented` и `Ellipsis`, имена любых встроенных типов (таких как `int`, `float`, `list`, `str` и `tuple`), а также имена любых встроенных функций или исключений. Как определить, относится ли тот или иной идентификатор к этим категориям? В языке Python имеется встроенная функция `dir()`, которая возвращает список атрибутов объекта. Если эта функция вызывается без аргументов, она возвращает список встроенных атрибутов языка Python. Например:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
```

Атрибут `__builtins__` в действительности является модулем, в котором определены все встроенные атрибуты языка Python. Его можно использовать в качестве аргумента функции `dir()`:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
...
'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

В списке присутствует более 130 имен, поэтому мы опустили значительную их часть. Имена, начинающиеся с символов верхнего регистра, являются именами встроенных исключений. Остальные имена представляют функции и типы данных.

Если запоминание или поиск идентификаторов, использования которых следует избегать, кажется вам слишком утомительным, то можно воспользоваться инструментом проверки программного кода на языке Python, таким как PyLint (*www.logilab.org/project/name/pylint*). Этот инструмент поможет вам также выявлять множество других фактических или потенциальных проблем в программах на языке Python.

Второе соглашение касается использования символа подчеркивания (_). Не должны использоваться имена, начинающиеся и заканчивающиеся двумя символами подчеркивания (такие как `__lt__`). В языке Python определено множество различных специальных методов и переменных с такими именами (и в случае специальных методов мы можем заменять их, то есть создать свои версии этих методов), но мы не должны вводить новые имена такого рода. Такие имена будут рассматриваться в главе 6. Имена, начинающиеся с одного или двух символов подчеркивания (и не завершающиеся двумя символами подчеркивания), в некоторых контекстах интерпретируются как специальные. Мы продемонстрируем это в главе 5, когда будем использовать имена, начинающиеся с одного символа подчеркивания, и в главе 6, когда будем использовать имена, начинающиеся с двух символов подчеркивания.

Символ подчеркивания сам по себе может использоваться в качестве идентификатора; внутри интерактивной оболочки интерпретатора или в командной оболочке Python в переменной с именем _ сохраняется ре-

зультат последнего вычисленного выражения. Во время выполнения обычной программы идентификатор _ отсутствует, если мы явно не определяем его в своем программном коде. Некоторые программисты любят использовать _ в качестве идентификатора переменной цикла в циклах `for ... in`, когда не требуется обращаться к элементам, по которым выполняются итерации. Например:

```
for _ in (0, 1, 2, 3, 4, 5):
    print("Hello")
```

Но имейте в виду, что те, кто пишет программы, которые затем интернационализируются, часто используют идентификатор _ в качестве имени функции перевода. Делается это, чтобы вместо необходимости писать всякий раз `gettext.gettext("Translate me")` можно было писать `_("Translate me")`. (Чтобы можно было выполнить такой вызов, мы сначала должны импортировать модуль `gettext`, чтобы получить доступ к функции `gettext()`, находящейся в этом модуле).

Инструкция `import`, стр. 53, 230

Давайте рассмотрим примеры допустимых идентификаторов во фрагменте программного кода, написанного программистом, говорящим на испанском языке. Здесь предполагается, что была выполнена инструкция `import math` и где-то выше в программе определены переменные `radio` и `vieja_área`:

```
π[1] = math.pi
ε = 0.0000001
nueva_área = π * radio * radio
if abs(nueva_área - vieja_área) < ε:
    print("las áreas han convergido")
```

Мы использовали здесь модуль `math`, записали в эпсилон (ε) очень маленькое число с плавающей точкой и с помощью функции `abs()` нашли абсолютное значение разницы площадей – обо всем об этом мы поговорим ниже, в этой же главе. Здесь следует обратить внимание на то, что мы можем использовать в идентификаторах национальные символы и символы греческого алфавита. С той же легкостью мы могли бы использовать арабские, китайские, еврейские, японские и русские символы и практически любые другие алфавитные символы, поддерживаемые кодировкой Юникод.

Самый простой способ проверить допустимость идентификатора состоит в том, чтобы попробовать присвоить ему некоторое значение в интерактивной оболочке интерпретатора Python или в командной оболочке Python среды IDLE. Ниже приводятся несколько примеров:

```
>>> stretch-factor = 1
SyntaxError: can't assign to operator (...)
```

[1] Это символ «пи» греческого алфавита (π). – *Прим. перев.*

```
(SyntaxError: невозможно выполнить присваивание оператору (...))
>>> 2miles = 2
SyntaxError: invalid syntax (...)
(SyntaxError: синтаксическая ошибка (...))
>>> str = 3 # Допустимо, но неправильно
>>> l'impЂt31 = 4
SyntaxError: EOL while scanning single-quoted string (...)
(SyntaxError: встречен конец строки при анализе строки в апострофах (...))
>>> l_impЂt31 = 5
>>>
```

Попытка использовать недопустимый идентификатор вызывает исключение `SyntaxError`. В каждом конкретном случае изменяется часть текста сообщения, которая окружена круглыми скобками, поэтому эту часть мы заменили многоточием. В первом примере ошибка обусловлена попыткой использовать символ «-», который не является алфавитным символом Юникода, цифрой или символом подчеркивания. Во втором случае ошибка произошла из-за того, что начальный символ не является алфавитным символом Юникода или символом подчеркивания. Если идентификатор допустим, исключение не возникает, даже если выбранный идентификатор совпадает с именем встроенного типа данных, исключения или функции, поэтому третий пример присваивания не вызвал ошибку, хотя выбор такого идентификатора – опрометчивый шаг. В четвертом примере ошибка вызвана использованием символа апострофа, который не является алфавитным символом Юникода, цифрой или символом подчеркивания. Пятый пример – пример подходящего варианта.

Целочисленные типы

В языке Python имеется два целочисленных типа, `int` и `bool`.[1] И целые числа, и логические значения являются неизменяемыми объектами, но благодаря присутствию в языке Python комбинированных операторов присваивания эта особенность практически незаметна. В логических выражениях число 0 и значение `False` представляют `False`, а любое другое целое число и значение `True` представляют `True`. В числовых выражениях значение `True` представляет 1, а `False` – 0. Это означает, что можно записывать весьма странные выражения, например, выражение `i += True` увеличит значение `i` на единицу. Естественно, более правильным будет записывать подобные выражения как `i += 1`.

[1] В стандартной библиотеке также определяется тип `fractions.Fraction` (рациональные числа неограниченной точности), который может пригодиться при выполнении некоторых математических и научных вычислений.

Целые числа

Размер целого числа ограничивается только объемом памяти компьютера, поэтому легко можно создать и обрабатывать целое число, состоящее из тысяч цифр, правда, скорость работы с такими числами существенно медленнее, чем с числами, которые соответствуют машинному представлению.

Литералы целых чисел по умолчанию записываются в десятичной системе счисления, но при желании можно использовать другие системы счисления:

```
>>> 14600926                    # десятичное число
14600926
>>> 0b110111101100101011011110 # двоичное число
14600926
>>> 0o67545336                  # восьмеричное число
14600926
>>> 0xDECADE                    # шестнадцатеричное число
14600926
```

Двоичные числа записываются с префиксом `0b`, восьмеричные – в префиксом `0o`[1] и шестнадцатеричные – с префиксом `0x`. В префиксах допускается использовать символы верхнего регистра.

При работе с целыми числами могут использоваться обычные математические функции и операторы, как показано в табл. 2.2. Некоторые из функциональных возможностей представлены встроенными функциями, такими как `abs()` (например, вызов `abs(i)` вернет абсолютное значение целого числа `i`), а другие – операторами, применимыми к типу `int` (например, выражение `i + j` вернет сумму целых чисел `i` и `j`).

Для всех двухместных арифметических операторов (`+`, `-`, `/`, `//`, `%` и `**`) имеются соответствующие комбинированные операторы присваивания (`+=`, `-=`, `/=`, `//=`, `%=` и `**=`), где выражение `x` *`op`*`= y` является логическим эквивалентом выражения `x = x` *`op`* `y`, когда в обычной ситуации обращение к значению `x` не имеет побочных эффектов.

Объекты могут создаваться путем присваивания литералов переменным, например, `x = 17`, или обращением к имени соответствующего типа как к функции, например, `x = int(17)`. Некоторые объекты (например, типа `decimal.Decimal`) могут создаваться только посредством использования их типов, так как они не имеют литерального представления. Создание объекта посредством использования его типа может быть выполнено одним из трех способов.

---

[1] Пользователи языка C должны обратить внимание, что одного ведущего `0` недостаточно, чтобы определить восьмеричное число – в языке Python следует использовать комбинацию `0o` (**0** и символ `o`).

Таблица 2.2. Арифметические операторы и функции

Синтаксис	Описание
`x + y`	Складывает число `x` и число `y`
`x - y`	Вычитает число `y` из числа `x`
`x * y`	Умножает `x` на `y`
`x / y`	Делит `x` на `y` – результатом всегда является значение типа `float` (или `complex`, если `x` или `y` является комплексным числом)
`x // y`	Делит `x` на `y`, при этом усекает дробную часть, поэтому результатом всегда является значение типа `int`, смотрите также функцию `round()`
`x % y`	Возвращает модуль (остаток) от деления `x` на `y`
`x ** y`	Возводит `x` в степень `y`, смотрите также функцию `pow()`
`-x`	Изменяет знак числа `x`, если оно не является нулем, если ноль – ничего не происходит
`+x`	Ничего не делает, иногда используется для повышения удобочитаемости программного кода
`abs(x)`	Возвращает абсолютное значение `x`
`divmod(x, y)`	Возвращает частное и остаток деления `x` на `y` в виде кортежа двух значений типа `int`
`pow(x, y)`	Возводит `x` в степень `y`; то же самое, что и оператор `**`
`pow(x, y, z)`	Более быстрая альтернатива выражению `(x ** y) % z`
`round(x, n)`	Возвращает значение типа `int`, соответствующее значению `x` типа `float`, округленному до ближайшего целого числа (или значение типа `float`, округленное до *n*-го знака после запятой, если задан аргумент *n*)

Кортежи, стр. 32, 130

Первый вариант – вызов типа данных без аргументов. В этом случае объект приобретает значение по умолчанию, например, выражение `x = int()` создаст целое число 0. Любые встроенные типы могут вызываться без аргументов.

Поверхностное и глубокое копирование, стр. 173

Второй вариант – тип вызывается с единственным аргументом. Если указан аргумент соответствующего типа, будет создана поверхностная копия оригинального объекта. (Поверхностное копирование рассматривается в главе 3.) Если задан аргумент другого типа, будет предпринята попытка выполнить преобразование. Такой способ использования описывается в табл. 2.3. Если аргумент имеет тип, для которого поддерживается преобразование в требуемый тип, и преобразование терпит неудачу, возбуждается исключение `ValueError`, в противном случае возвращается результат преобразования – объект

требуемого типа. Если тип аргумента не поддерживает преобразование в требуемый тип, возбуждается исключение `TypeError`. Встроенные типы `float` и `str` поддерживают возможность преобразования в целое число. Точно так же возможно обеспечить преобразование в целое число ваших собственных типов данных, как будет показано в главе 6.

Преобразование типов, стр. 295

Таблица 2.3. Функции преобразования целых чисел

Синтаксис	Описание
`bin(i)`	Возвращает двоичное представление целого числа `i` в виде строки, например, `bin(1980) == '0b11110111100'`
`hex(i)`	Возвращает шестнадцатеричное представление целого числа `i` в виде строки, например, `hex(1980) == '0x7bc'`
`int(x)`	Преобразует объект `x` в целое число; в случае ошибки во время преобразования возбуждает исключение `ValueError`, а если тип объекта `x` не поддерживает преобразование в целое число, возбуждает исключение `TypeError`. Если `x` является числом с плавающей точкой, оно преобразуется в целое число путем усечения дробной части.
`int(s, base)`	Преобразует строку `s` в целое число, в случае ошибки возбуждает исключение `ValueError`. Если задан необязательный аргумент *base*, он должен быть целым числом в диапазоне от 2 до 36 включительно.
`oct(i)`	Возвращает восьмеричное представление целого числа `i` в виде строки, например, `oct(1980) == '0o3674'`

Третий вариант – когда передается два или более аргументов; не все типы поддерживают такую возможность, а для тех типов, что поддерживают ее, типы аргументов и их назначение отличаются. В случае типа `int` допускается передавать два аргумента, где первый аргумент – это строка с представлением целого числа, а второй аргумент – число основания системы счисления. Например, вызов `int("A4", 16)` создаст десятичное значение 164. Этот вариант использования продемонстрирован в табл. 2.3.

В табл. 2.4 перечислены битовые операторы. Все битовые операторы (`|`, `^`, `&`, `<<` и `>>`) имеют соответствующие комбинированные операторы присваивания (`|=`, `^=`, `&=`, `<<=` и `>>=`), где выражение `i` *op*`= j` является логическим эквивалентом выражения `i = i` *op* `j` в случае, когда обращение к значению `i` не имеет побочных эффектов.

Если имеется необходимость хранить множество флагов, способных иметь всего два состояния, можно использовать единственное целое число и проверять значения отдельных его битов с помощью битовых операторов. То же самое можно делать менее компактным, но более удобным способом, воспользовавшись логическим типом.

Таблица 2.4. Битовые операторы, применимые к целым числам

Синтаксис	Описание
`i | j`	Битовая операция `OR` (ИЛИ) над целыми числами `i` и `j`; отрицательные числа представляются как двоичное дополнение
`i ^ j`	Битовая операция `XOR` (исключающее ИЛИ) над целыми числами `i` и `j`
`i & j`	Битовая операция `AND` (И) над целыми числами `i` и `j`
`i << j`	Сдвигает значение `i` влево на `j` битов аналогично операции `i * (2 ** j)` без проверки на переполнение
`i >> j`	Сдвигает значение `i` вправо на `j` битов аналогично операции `i // (2 ** j)` без проверки на переполнение
`~i`	Инвертирует биты числа `i`

Логические значения

Существует два встроенных логических объекта: `True` и `False`. Как и все остальные типы данных в языке Python (встроенные, библиотечные или ваши собственные), тип данных `bool` может вызываться как функция – при вызове без аргументов возвращается значение `False`, при вызове с аргументом типа `bool` возвращается копия аргумента, а при вызове с любым другим аргументом предпринимается попытка преобразовать указанный объект в тип `bool`. Все встроенные типы данных и типы данных из стандартной библиотеки могут быть преобразованы в тип `bool`, а добавить поддержку такого преобразования в свои собственные типы данных не представляет никакой сложности. Ниже приводится пара присваиваний логических значений и пара логических выражений:

```
>>> t = True
>>> f = False
>>> t and f
False
>>> t and True
True
```

Логические операторы, стр. 39

Как уже отмечалось ранее, в языке Python имеется три логических оператора: `and`, `or` и `not`. Выражения с участием операторов `and` и `or` вычисляются в соответствии с логикой сокращенных вычислений (*short-circuit logic*), и возвращается операнд, определяющий значение всего выражения, тогда как результатом оператора `not` всегда является либо `True`, либо `False`.

Программисты, использовавшие старые версии языка Python, иногда вместо `True` и `False` используют числа `1` и `0` – такой прием срабатывает практически всегда, но в новых программах, когда возникает необхо-

димость в логическом значении, следует использовать встроенные логические объекты.

Тип чисел с плавающей точкой

Язык Python предоставляет три типа значений с плавающей точкой: встроенные типы `float` и `complex` и тип `decimal.Decimal` в стандартной библиотеке. Все три типа данных относятся к категории неизменяемых. Тип `float` представляет числа с плавающей точкой двойной точности, диапазон значений которых зависит от компилятора языка C (или C# или Java), применявшегося для компиляции интерпретатора Python. Числа этого типа имеют ограниченную точность и не могут надежно сравниваться на равенство значений. Числа типа `float` записываются с десятичной точкой или в экспоненциальной форме записи, например, 0.0, 4., 5.7, -2.5, -2e9, 8.9e-4.

В машинном представлении числа с плавающей точкой хранятся как двоичные числа. Это означает, что одни дробные значения могут быть представлены точно (такие как 0.5), а другие – только приблизительно (такие как 0.1 и 0.2). Кроме того, для представления используется фиксированное число битов, поэтому существует ограничение на количество цифр в представлении таких чисел. Ниже приводится поясняющий пример, полученный в IDLE:

```
>>> 0.0, 5.4, -2.5, 8.9e-4
(0.0, 5.4000000000000004, -2.5, 0.0008899999999999995)
```

Проблема потери точности – это не проблема, свойственная только языку Python; все языки программирования обнаруживают проблему с точным представлением чисел с плавающей точкой.

Если вам действительно необходимо обеспечить высокую точность, можно использовать числа типа `decimal.Decimal`. Эти числа обеспечивают уровень точности, который вы укажете (по умолчанию 28 знаков после запятой), и могут точно представлять периодические числа, такие как 0.1[1], но скорость работы с такими числами существенно ниже, чем с обычными числами типа `float`. Вследствие высокой точности числа типа `decimal.Decimal` прекрасно подходят для производства финансовых вычислений.

Смешанная арифметика поддерживается таким образом, что результатом выражения с участием чисел типов `int` и `float` является число типа `float`, а с участием типов `float` и `complex` результатом является число типа `complex`. Поскольку числа типа `decimal.Decimal` имеют фиксированную точность, они могут участвовать в выражениях только

[1] В десятичной системе счисления число 0.1 не является периодической дробью, но в двоичной (то есть в машинном представлении) – это действительно периодическая дробь. – *Прим. перев.*

с другими числами decimal.Decimal и с числами типа int; результатом таких выражений является число decimal.Decimal. В случае попытки выполнить операцию над несовместимыми типами возбуждается исключение TypeError.

Числа с плавающей точкой

Все числовые операторы и функции, представленные в табл. 2.2 (стр. 74), могут применяться к числам типа float, включая комбинированные операторы присваивания. Тип данных float может вызываться как функция – без аргументов возвращается число 0.0, с аргументом типа float возвращается копия аргумента, а с аргументом любого другого типа предпринимается попытка выполнить преобразование указанного объекта в тип float. При преобразовании строки аргумент может содержать либо простую форму записи числа с десятичной точкой, либо экспоненциальное представление числа. При выполнении операций с числами типа float может возникнуть ситуация, когда в результате получается значение NaN (not a number – не число) или «бесконечность». К сожалению, поведение интерпретатора в таких ситуациях может отличаться в разных реализациях и зависит от математической библиотеки системы.

Ниже приводится пример простой функции, выполняющей сравнение чисел типа float на равенство в пределах машинной точности:

```
def equal_float(a, b):
    return abs(a - b) <= sys.float_info.epsilon
```

Чтобы воспользоваться этой функцией, необходимо импортировать модуль sys. Объект sys.float_info имеет множество атрибутов. Так, sys.float_info.epsilon хранит минимально возможную разницу между двумя числами с плавающей точкой. На одной из 32-разрядных машин автора книги это число чуть больше 0.000 000 000 000 000 2. (Epsilon – это традиционное название чисел такого рода.) Тип float в языке Python обеспечивает надежную точность до 17 значащих цифр.

Если ввести sys.float_info в среде IDLE, будут выведены все атрибуты этого объекта, куда входят минимальное и максимальное значения чисел с плавающей точкой, которые могут быть представлены машиной. А если ввести команду help(sys.float_info), будет выведена некоторая информация об объекте sys.float_info.

Числа с плавающей точкой можно преобразовать в целые числа с помощью функции int(), которая возвращает целую часть и отбрасывает дробную часть, или с помощью функции round(), которая учитывает величину дробной части, или с помощью функций math.floor() и math.ceil(), которые округляют вверх или вниз до ближайшего целого. Метод float.is_integer() возвращает значение True, если дробная часть числа равна 0. Представление дробной части числа можно получить с помощью метода float.as_integer_ratio(). Например, пусть x = 2.75,

тогда метод `x.as_integer_ratio()` вернет `(11, 4)`. Преобразование целых чисел в тип `float` можно выполнить с помощью функции `float()`.

Числа с плавающей точкой также могут быть представлены в виде строк в шестнадцатеричном формате с помощью метода `float.hex()`. Обратное преобразование может быть выполнено с помощью метода `float.fromhex()`.[1] Например:

```
s = 14.25.hex()      # str s == '0x1.c800000000000p+3'
f = float.fromhex(s) # float f == 14.25
t = f.hex()          # str t == '0x1.c800000000000p+3'
```

Экспонента отмечается с помощью символа `p` («power» – «степень»), а не `e`, так как символ `e` представляет допустимую шестнадцатеричную цифру.

В дополнение к встроенным функциональным возможностям работы с числами типа `float` модуль `math` предоставляет множество функций, которые приводятся в табл. 2.5. Ниже приводятся несколько фрагментов программного кода, демонстрирующих, как можно использовать функциональные возможности модуля:

```
>>> import math
>>> math.pi * (5 ** 2)
78.539816339744831
>>> math.hypot(5, 12)
13.0
>>> math.modf(13.732)
(0.73199999999999932, 13.0)
```

Функция `math.hypot()` вычисляет расстояние от начала координат до точки (x, y) и дает тот же результат, что и выражение `math.sqrt((x ** 2) + (y ** 2))`.

Модуль `math` в значительной степени опирается на математическую библиотеку, с которой был собран интерпретатор Python. Это означает, что при некоторых условиях и в граничных случаях функции модуля могут иметь различное поведение на различных платформах.

Таблица 2.5. Функции и константы модуля math

Синтаксис	Описание
`math.acos(x)`	Возвращает арккосинус `x` в радианах
`math.acosh(x)`	Возвращает гиперболический арккосинус `x` в радианах
`math.asin(x)`	Возвращает арксинус `x` в радианах
`math.asinh(x)`	Возвращает гиперболический арксинус `x` в радианах
`math.atan(x)`	Возвращает арктангенс `x` в радианах

[1] Примечание для программистов, использующих объектно-ориентированный стиль: `float.fromhex()` – это метод класса.

Таблица 2.5 (продолжение)

Синтаксис	Описание
`math.atan2(y, x)`	Возвращает арктангенс `y/x` в радианах
`math.atanh(x)`	Возвращает гиперболический арктангенс `x` в радианах
`math.ceil(x)`	Возвращает $\lceil x \rceil$, то есть наименьшее целое число типа `int`, большее и равное `x`, например, `math.ceil(5.4) == 6`
`math.copysign(x,y)`	Возвращает `x` со знаком числа `y`
`math.cos(x)`	Возвращает косинус `x` в радианах
`math.cosh(x)`	Возвращает гиперболический косинус `x` в радианах
`math.degrees(r)`	Преобразует число `r`, типа `float`, из радианов в градусы
`math.e`	Константа e, примерно равная значению 2.7182818284590451
`math.exp(x)`	Возвращает e^x, то есть `math.e ** x`
`math.fabs(x)`	Возвращает $\lvert x \rvert$, то есть абсолютное значение `x` в виде числа типа `float`
`math.factorial(x)`	Возвращает $x!$
`math.floor(x)`	Возвращает $\lfloor x \rfloor$, то есть наименьшее целое число типа `int`, меньшее и равное `x`, например, `math.floor(5.4) == 5`
`math.fmod(x, y)`	Выполняет деление по модулю (возвращает остаток) числа `x` на число `y`; дает более точный результат, чем оператор `%`, применительно к числам типа `float`
`math.frexp(x)`	Возвращает кортеж из двух элементов с мантиссой (в виде числа типа `float`) и экспонентой (в виде числа типа `int`) Кортежи, стр. 32, 130
`math.fsum(i)`	Возвращает сумму значений в итерируемом объекте `i` в виде числа типа `float`
`math.hypot(x, y)`	Возвращает $\sqrt{x^2 + y^2}$
`math.isinf(x)`	Возвращает `True`, если значение `x` типа `float` является бесконечностью (`±inf(±∞)`)
`math.isnan(x)`	Возвращает `True`, если значение `x` типа `float` не является числом
`math.ldexp(m, e)`	Возвращает $m \times 2^e$ – операция, обратная `math.frexp()`
`math.log(x, b)`	Возвращает $log_b x$, аргумент b является необязательным и по умолчанию имеет значение `math.e`
`math.log10(x)`	Возвращает $log_{10} x$
`math.log1p(x)`	Возвращает $log_e(1+x)$; дает точные значения, даже когда значение `x` близко к 0
`math.modf(x)`	Возвращает дробную и целую часть числа `x` в виде двух значений типа `float`

Синтаксис	Описание
`math.pi`	Константа π, примерно равна 3.1415926535897931
`math.pow(x, y)`	Возвращает x^y в виде числа типа `float`
`math.radians(d)`	Преобразует число `d`, типа `float`, из градусов в радианы
`math.sin(x)`	Возвращает синус `x` в радианах
`math.sinh(x)`	Возвращает гиперболический синус `x` в радианах
`math.sqrt(x)`	Возвращает $\sqrt{x}$
`math.sum(i)`	Возвращает сумму значений в итерируемом объекте `i` в виде числа типа `float`[a]
`math.tan(x)`	Возвращает тангенс `x` в радианах
`math.tanh(x)`	Возвращает гиперболический тангенс `x` в радианах
`math.trunc(x)`	Возвращает целую часть числа `x` в виде значения типа `int`; то же самое, что и `int(x)`

[a] Функции `math.sum` в модуле `math` нет; предполагаю, что эта функция в Python 3.0 вытеснена функцией `math.fsum`. – *Прим. перев.*

Комплексные числа

Тип данных `complex` относится к категории неизменяемых и хранит пару значений типа `float`, одно из которых представляет действительную часть комплексного числа, а другое – мнимую. Литералы комплексных чисел записываются как действительная и мнимая части, объединенные знаком + или -, а за мнимой частью числа следует символ `j`.[1] Вот примеры нескольких комплексных чисел: `3.5+2j`, `0.5j`, `4+0j`, `-1-3.7j`. Обратите внимание, что если действительная часть числа равна 0, ее можно вообще опустить.

Отдельные части комплексного числа доступны в виде атрибутов `real` и `imag`. Например:

```
>>> z = -89.5+2.125j
>>> z.real, z.imag
(-89.5, 2.125)
```

За исключением `//`, `%`, `divmod()` и версии `pow()` с тремя аргументами все остальные арифметические операторы и функции, перечисленные в табл. 2.2 (стр. 74), могут использоваться для работы с комплексными числами, так же как и соответствующие комбинированные операторы присваивания. Кроме того, значения типа `complex` имеют метод `conjugate()`, который изменяет знак мнимой части. Например:

[1] В математике, чтобы показать $\sqrt{-1}$, используется символ i, но в Python, следуя инженерной традиции, используется символ j.

```
>>> z.conjugate()
(-89.5-2.125j)
>>> 3-4j.conjugate()
(3+4j)
```

Обратите внимание, что в этом примере был вызван метод для литерала комплексного числа. Вообще, язык Python позволяет вызывать методы любого литерала или обращаться к его атрибутам при условии, что тип данных литерала имеет вызываемый метод или атрибут. Однако это не относится к специальным методам, так как им всегда соответствуют определенные операторы, которые и должны использоваться. Например, выражение `4j.real` вернет `0.0`, выражение `4j.imag` вернет `4.0` и выражение `4j + 3+2j` вернет `3+6j`.

Тип данных `complex` может вызываться как функция – без аргументов она вернет значение `0j`, с аргументом типа `complex` она вернет копию аргумента, а с аргументом любого другого типа она попытается преобразовать указанный объект в значение типа `complex`. При использовании для преобразования функция `complex()` принимает либо единственный строковый аргумент, либо одно или два значения типа `float`. Если ей передается единственное значение типа `float`, возвращается комплексное число с мнимой частью, равной `0j`.

Функции в модуле `math` не работают с комплексными числами. Это сделано преднамеренно, чтобы гарантировать, что пользователи модуля `math` будут получать исключения вместо получения комплексных чисел в некоторых случаях.

Если возникает необходимость использовать комплексные числа, можно воспользоваться модулем `cmath`, который содержит комплексные версии большинства тригонометрических и логарифмических функций, присутствующих в модуле `math`, плюс ряд функций, специально предназначенных для работы с комплексными числами, таких как `cmath.phase()`, `cmath.polar()` и `cmath.rect()`, а также константы `cmath.pi` и `cmath.e`, которые хранят те же самые значения типа `float`, что и родственные им константы в модуле `math`.

Числа типа Decimal

Во многих приложениях недостаток точности, свойственный числам типа `float`, не имеет существенного значения, и эта неточность окупается скоростью вычислений. Но в некоторых случаях предпочтение отдается точности, даже в обмен на снижение скорости работы. Модуль `decimal` реализует неизменяемый числовой тип `Decimal`, который представляет числа с задаваемой точностью. Вычисления с участием таких чисел производятся значительно медленнее, чем в случае использования значений типа `float`, но насколько это важно, будет зависеть от приложения.

Чтобы создать объект типа `Decimal`, необходимо импортировать модуль `decimal`. Например:

```
>>> import decimal
>>> a = decimal.Decimal(9876)
>>> b = decimal.Decimal("54321.012345678987654321")
>>> a + b
Decimal('64197.012345678987654321')
```

Числа типа `Decimal` создаются с помощью функции `decimal.Decimal()`. Эта функция может принимать целочисленный или строковый аргумент, но не значение типа `float`, потому что числа типа `float` не всегда имеют точное представление, а числа типа `Decimal` всегда представляются точно. Если в качестве аргумента используется строка, она может содержать изображение числа как в обычной десятичной форме записи, так и в экспоненциальной. Кроме того, возможность явно определить точность представления числа `decimal.Decimal` означает, что они надежно могут проверяться на равенство.

Все арифметические операторы и функции, перечисленные в табл. 2.2 (стр. 74), включая соответствующие им комбинированные операторы присваивания, могут использоваться применительно к значениям типа `decimal.Decimal`, но с некоторыми ограничениями. Если слева от оператора ** находится объект типа `decimal.Decimal`, то справа от оператора должно быть целое число. Точно так же, если первый аргумент функции `pow()` имеет тип `decimal.Decimal`, то второй и необязательный третий аргументы должны быть целыми числами.

Модули `math` и `cmath` не могут использоваться для работы с числами типа `decimal.Decimal`, однако некоторые функции, присутствующие в модуле `math`, реализованы как методы типа `decimal.Decimal`. Например, чтобы вычислить e^x, где `x` имеет тип `float`, вызывается функция `math.exp(x)`, а когда `x` имеет тип `decimal.Decimal`, следует использовать метод `x.exp()`. Из обсуждения составляющей №3 (стр. 34) мы можем видеть, что `x.exp()` – это фактически разновидность записи `decimal.Decimal.exp(x)`.

Кроме того, тип данных `decimal.Decimal` предоставляет метод `ln()`, который вычисляет натуральные (по основанию e) логарифмы (точно так же, как функция `math.log()` с одним аргументом), `log10()` и `sqrt()`, а также множество других методов, адаптированных для обработки значений типа `decimal.Decimal`.

Значения типа `decimal.Decimal` работают в пределах *контекста*, где контекст – это коллекция параметров настройки, определяющих поведение чисел `decimal.Decimal`. Контекст определяет точность представления (по умолчанию 28 десятичных знаков), методику округления и некоторые другие особенности.

В некоторых ситуациях различия в точности представления между типами `float` и `decimal.Decimal` становятся очевидными:

```
>>> 23 / 1.05
21.904761904761905
>>> print(23 / 1.05)
21.9047619048
>>> print(decimal.Decimal(23) / decimal.Decimal("1.05"))
21.904761904761904761904761905
>>> decimal.Decimal(23) / decimal.Decimal("1.05")
Decimal('21.904761904761904761904761905')
```

Хотя деление чисел `decimal.Decimal` выполняется с более высокой точностью, чем деление чисел `float`, в данном случае (в 32-битной системе) различия обнаруживаются только в пятнадцатом знаке после десятичной точки. Во многих ситуациях такая неточность не имеет большого значения, например, в этой книге, где во всех примерах с участием чисел с плавающей точкой используются числа типа `float`.

С другой стороны, следует отметить, что последние два приведенных примера впервые демонстрируют, что печать объекта вызывает некоторое закулисное форматирование. Когда для вывода результата выражения `decimal.Decimal(23)/decimal.Decimal("1.05")` вызывается функция `print()`, выводится «голое» число – вывод имеет *строковую форму*. Если же просто вводится выражение, то на экран выводится `decimal.Decimal` – в данном случае вывод имеет *репрезентативную форму*. Все объекты в языке Python имеют две формы вывода. Строковая форма предназначена для вывода информации для человека. Репрезентативная форма является представлением объекта, которое можно передать интерпретатору Python (если это необходимо) и воспроизвести представляемый объект. Мы вернемся к этой теме в следующем разделе, когда будем обсуждать строки, и еще раз – в главе 6, когда будем обсуждать вопросы реализации строковой и репрезентативной формы для наших собственных типов данных.

В документе «Library Reference» модуль `decimal` описывается во всех подробностях, которые выходят далеко за рамки этой книги. Кроме того, там приводится множество примеров и список часто задаваемых вопросов с ответами на них.

Строки

Кодировки символов, стр. 112

Строки в языке Python представлены неизменяемым типом данных `str`, который хранит последовательность символов Юникода. Тип данных `str` может вызываться как функция для создания строковых объектов – без аргументов возвращается пустая строка; с аргументом, который не является строкой, возвращается строковое представление аргумента; а в случае, когда аргумент является строкой, возвращается его копия. Функция `str()` может также использоваться как функция преобразования. В этом случае первый аргумент должен быть стро-

кой или объектом, который можно преобразовать в строку, а, кроме того, функции может быть передано до двух необязательных строковых аргументов, один из которых определяет используемую кодировку, а второй определяет порядок обработки ошибок кодирования.

Ранее мы уже упоминали, что литералы строк создаются с использованием кавычек или апострофов, при этом важно, чтобы с обоих концов литерала использовались кавычки одного и того же типа. В дополнение к этому мы можем использовать *строки в тройных кавычках*, то есть строки, которые начинаются и заканчиваются тремя символами кавычки (либо тремя кавычками, либо тремя апострофами). Например:

```
text = """Строки в тройных кавычках могут включать 'апострофы' и "кавычки"
без лишних формальностей. Мы можем даже экранировать символ перевода строки \,
благодаря чему данная конкретная строка будет занимать всего две строки."""
```

Если нам потребуется использовать кавычки в строке, это можно сделать без лишних формальностей – при условии, что они отличаются от кавычек, ограничивающих строку; в противном случае символы кавычек или апострофов внутри строки следует экранировать:

```
a = "Здесь 'апострофы' можно не экранировать, а \"кавычки\" придется."
b = 'Здесь \'апострофы\' придется экранировать, а "кавычки" не обязательно.'
```

В языке Python символ перевода строки интерпретируется как завершающий символ инструкции, но не внутри круглых скобок (()), квадратных скобок ([]), фигурных скобок ({}) и строк в тройных кавычках. Символы перевода строки могут без лишних формальностей использоваться в строках в тройных кавычках, и мы можем включать символы перевода строки в любые строковые литералы с помощью экранированной последовательности \n. Все экранированные последовательности, допустимые в языке Python, перечислены в табл. 2.6. В некоторых ситуациях, например, при записи регулярных выражений, приходится создавать строки с большим количеством символов обратного слеша. (Регулярные выражения будут темой обсуждения главы 12.) Это может доставлять определенные неудобства, так как каждый такой символ придется экранировать:

```
import re
phone1 = re.compile("^((?:[(]\\d+[)])?\\s*\\d+(?:-\\d+)?)$")
```

Решить эту проблему можно, используя «*сырые*» (raw) строки. Это обычные строки в кавычках или в тройных кавычках, в которые перед первой кавычкой добавлен символ *r*. Внутри таких строк все символы интерпретируются как обычные символы, поэтому отпадает необходимость экранировать символы, которые в других типах строк имеют специальное значение. Ниже приводится регулярное выражение для номера телефона в виде «сырой» строки:

```
phone2 = re.compile(r"^((?:[(]\d+[)])?\s*\d+(?:-\d+)?)$")
```

Таблица 2.6. Экранированные последовательности в языке Python

Последовательность	Значение
перевод_строки	Экранирует (то есть игнорирует) символ перевода строки
\\	Символ обратного слеша (\)
\'	Апостроф (')
\"	Кавычка (")
\a	Символ ASCII «сигнал» (bell, BEL)
\b	Символ ASCII «забой» (backspace, BS)
\f	Символ ASCII «перевод формата» (formfeed, FF)
\n	Символ ASCII «перевод строки» (linefeed, LF)
\N{название}	Символ Юникода с заданным названием
ooo	Символ с заданным восьмеричным кодом
\r	Символ ASCII «возврат каретки» (carriage return, CR)
\t	Символ ASCII «табуляция» (tab, TAB)
\u*hhhh*	Символ Юникода с указанным 16-битовым шестнадцатеричным значением
\U*hhhhhhhh*	Символ Юникода с указанным 32-битовым шестнадцатеричным значением
\v	Символ ASCII «вертикальная табуляция» (vertical tab, VT)
\x*hh*	Символ с указанным 8-битовым шестнадцатеричным значением

Если потребуется записать длинный строковый литерал, занимающий две или более строк, но без использования тройных кавычек, то можно использовать один из приемов, показанных ниже:

```
t = "Это не самый лучший способ объединения двух длинных строк, " + \
    "потому что он основан на использовании неуклюжего экранирования"
s = ("Это отличный способ объединить две длинные строки, "
     " потому что он основан на конкатенации строковых литералов.")
```

Обратите внимание, что во втором случае для создания единственного выражения мы должны были использовать круглые скобки – без этих скобок переменной `s` была бы присвоена только первая строка, а наличие второй строки вызвало бы исключение `IndentationError`. В руководстве «Idioms and Anti-Idioms», которое можно отыскать в документации к языку Python, рекомендуется вместо экранирования перевода строки всегда использовать круглые скобки для объединения операторов, не умещающихся в одну строку; именно этой рекомендации мы и будем следовать.

Поскольку содержимое файлов *.py* по умолчанию представляет собой текст в кодировке UTF-8 Юникод, мы можем использовать в строковых литералах любые символы Юникода. Мы можем даже помещать в строковые литералы символы Юникода, используя шестнадцатеричные экранированные последовательности или названия символов Юникода, например:

```
>>> euros = "Ъ \N{euro sign} \u20AC \U000020AC"
>>> print(euros)
Є Є Є Є
```

В данном случае мы не можем использовать обычную шестнадцатеричную экранированную последовательность, так как она ограничена двумя цифрами и не может представлять символы с кодом больше, чем `0xFF`. Обратите внимание, что названия символов Юникода не чувствительны к регистру и пробелы внутри них являются необязательными.

Для определения кода символа Юникода (целое число, связанное с символом в кодировке Юникод) в строке, можно использовать встроенную функцию `ord()`. Например:

Кодировки символов, стр. 112

```
>>> ord(euros[0])
8364
>>> hex(ord(euros[0]))
'0x20ac'
```

Точно так же можно преобразовать любое целое число, представляющее собой допустимый код некоторого символа Юникода, воспользовавшись функцией `chr()`:

```
>>> s = "anarchists are " + chr(8734) + chr(0x23B7)
>>> s
'anarchists are ∞⎷'
>>> ascii(s)
"'anarchists are \u221e\u23b7'"
```

Если ввести `s` в среде IDLE, содержимое объекта будет выведено в строковой форме; для строк это означает, что содержимое выводится в кавычках. Если нам потребуется вывести только символы ASCII, мы можем воспользоваться встроенной функцией `ascii()`, которая возвращает репрезентативную форму своего аргумента, используя 7-битовые символы ASCII, где это возможно; в противном случае используется наиболее краткая экранированная последовательность из возможных: `\x`*hh*, `\u`*hhhh* или `\U`*hhhhhhhh*. Ниже в этой главе будет показано, как получить полный контроль над выводом строк.

Метод str.format(), стр. 100

Сравнение строк

Строки поддерживают обычные операторы сравнения <, <=, ==, !=, > и >=. Эти операторы выполняют побайтовое сравнение строк в памяти. К сожалению, возникают две проблемы при сравнении, например, строк в отсортированных списках. Обе проблемы проявляются во всех языках программирования и не являются характерной особенностью Python.

Кодировки символов, стр. 112

Первая проблема связана с тем, что символы Юникода могут быть представлены двумя и более последовательностями байтов. Например, *Å* (символ Юникода с кодом 0x00C5) в кодировке UTF-8 может быть представлен тремя различными способами: [0xE2, 0x84, 0xAB], [0xC3, 0x85] и [0x41, 0xCC, 0x8A]. К счастью, мы можем решить эту проблему. Если импортировать модуль unicodedata и вызвать функцию unicodedata.normalize() со значением «NFKD» в первом аргументе (эта аббревиатура определяет способ нормализации «Normalization Form Compatibility Decomposition» – нормализация в форме совместимой декомпозиции), то, передав ей строку, содержащую символ *Å*, представленный любой из допустимых последовательностей байтов, мы получим строку с символами в кодировке UTF-8, где интересующий нас символ всегда будет представлен последовательностью [0x41, 0xCC, 0x8A].

Вторая проблема заключается в том, что порядок сортировки некоторых символов зависит от конкретного языка. Например, в шведском языке при сортировке символ *ä* следует после символа *z*, тогда как в немецком языке символ *ä* сортируется так, как если бы он был представлен последовательностью символов *ae*. Еще один пример: в английском языке символ *ø* сортируется как символ *o*, а в датском и норвежском языках он следует после символа *z*. Со строками Юникода связана масса проблем, которые становятся трудноразрешимыми, когда одним и тем же приложением могут пользоваться люди разных национальностей (привыкшие к различным порядкам расположения символов), когда строки содержат текст сразу на нескольких языках (например, часть строки на испанском, а часть на английском), и особенно, если учесть, что некоторые символы (такие как стрелки, декоративные и математические символы) не имеют определенного порядка сортировки.

Будучи настоящим политиком, во избежание трудноуловимых ошибок Python не делает никаких предположений. В смысле сравнения строк это означает, что выполняется побайтовое сравнение строк в памяти. При таком подходе порядок сортировки определяется кодами Юникода, что для английского языка дает сортировку в соответствии с кодами ASCII. Перевод всех символов строк в нижний или в верхний

регистр обеспечит более естественный порядок сортировки для английского языка. Нормализация может потребоваться, только когда текстовые строки поступают из внешних источников, таких как файлы или сетевые сокеты, но даже в этих случаях едва ли стоит применять ее, если нет веских доказательств в ее необходимости. При этом мы, конечно, можем настроить методы сортировки, как будет показано в главе 3. Проблема сортировки строк Юникода подробно рассматривается в документе «Unicode Collation Algorithm» (*unicode.org/reports/tr10*).

Получение срезов строк

Составляющая №3, стр. 32

Из описания составляющей №3 нам известно, что отдельные элементы последовательности, а, следовательно, и отдельные символы в строках, могут извлекаться с помощью оператора доступа к элементам (`[]`). В действительности этот оператор намного более универсальный и может использоваться для извлечения не только одного символа, но и целых комбинаций (подпоследовательностей) элементов или символов, когда этот оператор используется в контексте оператора извлечения среза.

Для начала мы рассмотрим возможность извлечения отдельных символов. Нумерация позиций символов в строках начинается с 0 и продолжается до значений длины строки минус 1. Однако допускается использовать и отрицательные индексы – в этом случае отсчет начинается с последнего символа и ведется в обратном направлении к первому символу. На рис. 2.1 показано, как нумеруются позиции символов в строке, если предположить, что было выполнено присваивание `s = "Light ray"`.

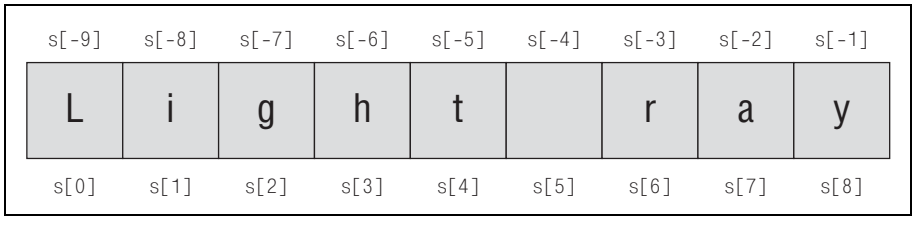

***Рис. 2.1.** Номера позиций символов в строке*

Отрицательные индексы удивительно удобны, особенно индекс -1, который всегда соответствует последнему символу строки. Попытка обращения к индексу, находящемуся за пределами строки (или к любому индексу в пустой строке), будет вызывать исключение `IndexError`.

Оператор получения среза имеет три формы записи:

```
seq[start]
seq[start:end]
seq[start:end:step]
```

Ссылка `seq` может представлять любую последовательность, такую как список, строку или кортеж. Значения *start*, *end* и *step* должны быть целыми числами (или переменными, хранящими целые числа). Мы уже использовали первую форму записи оператора доступа к элементам: с ее помощью извлекается элемент последовательности с индексом *start*. Вторая форма записи извлекает подстроку, начиная с элемента с индексом *start* и заканчивая элементом с индексом *end*, ***не включая*** его. Третью форму записи мы рассмотрим очень скоро.

При использовании второй формы записи (с одним двоеточием) мы можем опустить любой из индексов. Если опустить начальный индекс, по умолчанию будет использоваться значение 0. Если опустить конечный индекс, по умолчанию будет использоваться значение `len(seq)`. Это означает, что если опустить оба индекса, например, `s[:]`, это будет равносильно выражению `s[0:len(s)]`, и в результате будет извлечена, то есть скопирована, последовательность целиком.

На рис. 2.2 приводятся некоторые примеры извлечения срезов из строки `s`, которая получена в результате присваивания `s = "The waxwork man"`.

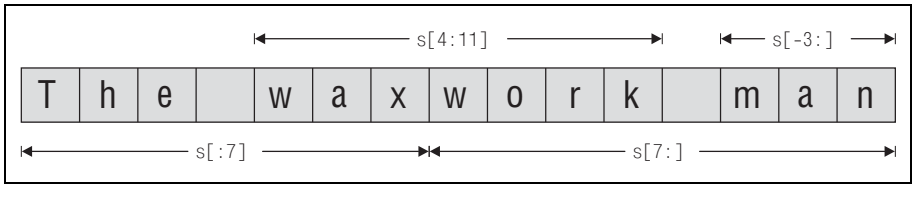

Рис. 2.2. *Извлечение срезов из последовательности*

Один из способов вставить подстроку в строку состоит в смешивании операторов извлечения среза и операторов конкатенации. Например:

```
>>> s = s[:12] + "wo" + s[12:]
>>> s
'The waxwork woman'
```

Кроме того, поскольку текст «wo» присутствует в оригинальной строке, тот же самый эффект можно было бы получить путем присваивания значения выражения `s[:12] + s[7:9] + s[12:]`.

Операторы и методы строк, стр. 92

Оператор конкатенации `+` и добавления подстроки `+=` не особенно эффективны, когда в операции участвует множество строк. Для объединения большого числа строк обычно лучше использовать метод `str.join()`, с которым мы познакомимся в следующем подразделе.

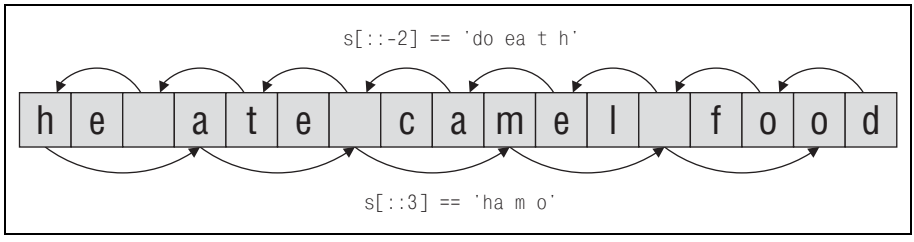

Рис. 2.3. *Извлечение разреженных срезов*

Третья форма записи (с двумя двоеточиями) напоминает вторую форму, но в отличие от нее значение *step* определяет, с каким шагом следует извлекать символы. Как и при использовании второй формы записи, мы можем опустить любой из индексов. Если опустить начальный индекс, по умолчанию будет использоваться значение 0, при условии, что задано неотрицательное значение *step*; в противном случае начальный индекс по умолчанию получит значение −1. Если опустить конечный индекс, по умолчанию будет использоваться значение `len(seq)`, при условии, что задано неотрицательное значение *step*; в противном случае конечный индекс по умолчанию получит значение индекса перед началом строки. Мы не можем опустить значение *step*, и оно не может быть равно нулю – если задание шага не требуется, то следует использовать вторую форму записи (с одним двоеточием), в которой шаг выбора элементов не указывается.

На рис. 2.3 приводится пара примеров извлечения разреженных срезов из строки `s`, которая получена в результате присваивания `s = "he ate camel food"`.

Здесь мы использовали значения по умолчанию для начального и конечного индексов, то есть извлечение среза `s[::-2]` начинается с последнего символа строки и извлекается каждый второй символ по направлению к началу строки. Аналогично извлечение среза `s[::3]` начинается с первого символа строки и извлекается каждый третий символ по направлению к концу строки.

Существует возможность комбинировать индексы с размером шага, как показано на рис. 2.4.

Операция извлечения элементов с определенным шагом часто применяется к последовательностям, отличным от строк, но один из ее вариантов часто применяется к строкам:

```
>>> s, s[::-1]
('The waxwork woman', 'namow krowxaw ehT')
```

Шаг −1 означает, что будет извлекаться каждый символ, от конца до начала, то есть будет получена строка, в которой символы следуют в обратном порядке.

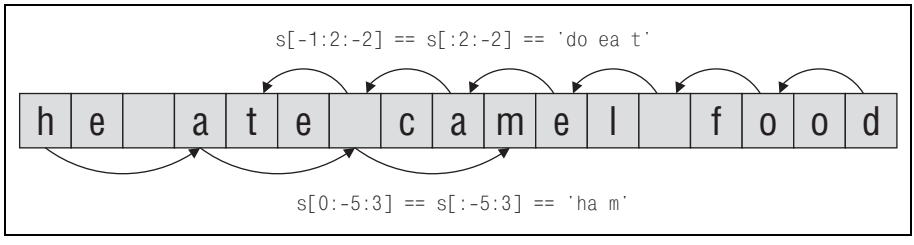

***Рис. 2.4.** Извлечение срезов из последовательности с определенным шагом*

Операторы и методы строк

Операторы и функции, применимые к итерируемым объектам, стр. 164

Поскольку строки относятся к категории неизменяемых последовательностей, все функциональные возможности, применимые к неизменяемым последовательностям, могут использоваться и со строками. Сюда входят оператор проверки на вхождение `in`, оператор конкатенации `+`, оператор добавления в конец `+=`, оператор дублирования `*` и комбинированный оператор присваивания с дублированием `*=`. Применение всех этих операторов в контексте строк мы обсудим в этом подразделе, а также обсудим большинство строковых методов. В табл. 2.7 приводится перечень всех строковых методов за исключением двух специализированных (`str.maketrans()` и `str.translate()`), которые будут обсуждаться немного позже.

Понятие «размер», стр. 443

Так как строки являются последовательностями, они являются объектами, имеющими «размер», и поэтому мы можем вызывать функцию `len()`, передавая ей строки в качестве аргумента. Возвращаемая функцией длина представляет собой количество символов в строке (ноль – для пустых строк).

Мы уже знаем, что перегруженная версия оператора `+` для строк выполняет операцию конкатенации. В случаях, когда требуется объединить множество строк, лучше использовать метод `str.join()`. Метод принимает в качестве аргумента последовательность (то есть список или кортеж строк) и объединяет их в единую строку, вставляя между ними строку, относительно которой был вызван метод. Например:

```
>>> treatises = ["Arithmetica", "Conics", "Elements"]
>>> " ".join(treatises)
'Arithmetica Conics Elements'
>>> "-<>-".join(treatises)
'Arithmetica-<>-Conics-<>-Elements'
>>> "".join(treatises)
'ArithmeticaConicsElements'
```

Первый пример является, пожалуй, наиболее типичным; он объединяет строки из списка, вставляя между ними единственный символ, в данном случае – пробел. Третий пример представляет собой операцию конкатенации в чистом виде – благодаря тому что метод вызывается относительно пустой строки, строки объединяются без добавления чего бы то ни было между ними.

Таблица 2.7. Строковые методы

Синтаксис	Описание
`s.capitalize()`	Возвращает копию строки `s` с первым символом в верхнем регистре; смотрите также метод `str.title`
`s.center(width, char)`	Возвращает копию строки `s`, отцентрированную в строке с длиной `width`. Недостающие символы по умолчанию заполняются пробелами или символами в соответствии с необязательным аргументом *char* (строка с длиной, равной 1); смотрите также методы `str.ljust()`, `str.rjust()` и `str.format()`
`s.count(t, start, end)`	Возвращает число вхождений строки `t` в строку `s` (или в срез строки `s[start:end]`)
`s.encode(encoding, err)`	Возвращает объект типа `bytes`, представляющий строку в кодировке по умолчанию или в кодировке, определяемой аргументом *encoding*, с обработкой ошибок, определяемой необязательным аргументом *err* Тип данных bytes, стр. 344 Кодировки символов, стр. 112
`s.endswith(x, start, end)`	Возвращает `True`, если строка `s` (или срез строки `s[start:end]`) оканчивается подстрокой `x` или любой из строк, если `x` – кортеж; в противном случае возвращает `False`. Смотрите также метод `str.startswith()`
`s.expandtabs(size)`	Возвращает копию строки `s`, в которой символы табуляции замещены пробелами с шагом 8 или в соответствии со значением необязательного аргумента *size*
`s.find(t, start, end)`	Возвращает позицию самого первого (крайнего слева) вхождения подстроки `t` в строку `s` (или в срез строки `s[start:end]`), если подстрока `t` не найдена, возвращается `-1`. Для поиска самого последнего (крайнего справа) вхождения следует использовать метод `str.rfind()`. Смотрите также метод `str.index()`
`s.format(...)`	Возвращает копию строки `s`, отформатированную в соответствии с заданными аргументами. Этот метод и его аргументы рассматриваются в следующем подразделе Метод str.format(), стр. 100
`s.index(t, start, end)`	Возвращает позицию самого первого (крайнего слева) вхождения подстроки `t` в строку `s` (или в срез строки `s[start:end]`); если подстрока `t` не найдена, возбуждается исключение `ValueError`. Для поиска самого последнего (крайнего справа) вхождения следует использовать метод `str.rfind()`

Таблица 2.7 (продолжение)

Синтаксис	Описание
`s.isalnum()`	Возвращает `True`, если строка `s` не пустая и содержит только алфавитно-цифровые символы
`s.isalpha()`	Возвращает `True`, если строка `s` не пустая и содержит только алфавитные символы
`s.isdecimal()`	Возвращает `True`, если строка `s` не пустая и содержит только символы Юникода, обозначающие цифры десятичной системы счисления
`s.isdigit()`	Возвращает `True`, если строка `s` не пустая и содержит только символы ASCII, обозначающие цифры десятичной системы счисления
`s.isidentifier()`	Возвращает `True`, если строка `s` не пустая и является допустимым идентификатором Идентификаторы и ключевые слова, стр. 68
`s.islower()`	Возвращает `True`, если строка `s` имеет хотя бы один символ, который может быть представлен в нижнем регистре, и все такие символы находятся в нижнем регистре; смотрите также метод `str.isupper()`
`s.isnumeric()`	Возвращает `True`, если строка `s` не пустая и содержит только символы Юникода, используемые для обозначения чисел
`s.isprintable()`	Возвращает `True`, если строка `s` пустая или содержит только печатаемые символы, включая пробел, но не символ перевода строки
`s.isspace()`	Возвращает `True`, если строка `s` не пустая и содержит только пробельные символы
`s.istitle()`	Возвращает `True`, если строка `s` не пустая и имеет формат заголовка; смотрите также метод `str.title()`
`s.isupper()`	Возвращает `True`, если строка `s` имеет хотя бы один символ, который может быть представлен в верхнем регистре, и все такие символы находятся в верхнем регистре; смотрите также метод `str.islower()`
`s.join(seq)`	Объединяет все элементы последовательности `seq`, вставляя между ними строку `s` (которая может быть пустой строкой)

`s.ljust(`
`  width,`
`  char)`
	Возвращает копию строки `s`, выровненной по левому краю, в строке длиной `width`. Недостающие символы по умолчанию заполняются пробелами или символами в соответствии с необязательным аргументом *`char`* (строка с длиной, равной 1). Для выравнивания по правому краю используйте метод `str.rjust()`, для выравнивания по центру – метод `str.center()`; смотрите также метод `str.format()`

`s.lower()`	Возвращает копию строки `s`, в которой все символы приведены к нижнему регистру; смотрите также метод `str.upper()`

Синтаксис	Описание
`s.maketrans()`	Парный метод для `str.translate()`; подробности приводятся в тексте
`s.partition (t)`	Возвращает кортеж из трех строк – часть строки `s` перед самым первым (крайним слева) вхождением подстроки `t`, `t` и часть строки `s` после подстроки `t`; если подстрока `t` в строке `s` отсутствует, возвращаются строка `s` и две пустые строки. Для деления строки по самому последнему (крайнему справа) вхождению подстроки `t`, используйте метод `str.rpartition()`
`s.replace (t, u, n)`	Возвращает копию строки `s`, в которой каждое (но не более *n*, если этот аргумент определен) вхождение подстроки `t` замещается подстрокой `u`
`s.split(t, n)`	Возвращает список строк, выполняя разбиение строки `s` не более чем *n* раз по подстроке *t*. Если число *n* не задано, разбиение выполняется по всем найденным подстрокам *t*. Если подстрока *t* не задана, разбиение выполняется по пробельным символам. Для выполнения разбиения строки, начиная с правого края, используйте метод `str.rsplit` – этот метод имеет смысл применять, когда задано число разбиений *n*, которое меньше максимального числа возможных разбиений
`s.splitlines (f)`	Возвращает список строк, выполняя разбиение строки `s` по символам перевода строки, удаляя их, если в аргументе *f* не задано значение `True`
`s.startswith (x, start, end)`	Возвращает `True`, если строка `s` (или срез строки `s[start:end]`) начинается подстрокой `x` или любой из строк, если `x` – кортеж; в противном случае возвращает `False`. Смотрите также метод `str.endswith()`
`s.strip(chars)`	Возвращает копию строки `s`, из которой удалены начальные и завершающие пробельные символы (или символы, входящие в строку *chars*). Метод `str.lstrip()` выполняет удаление только в начале строки, а метод `str.rstrip()` – только в конце
`s.swapcase()`	Возвращает копию строки `s`, в которой все символы верхнего регистра преобразованы в символы нижнего регистра, а все символы нижнего регистра – в символы верхнего регистра; смотрите также методы `str.lower()` и `str.upper()`
`s.title()`	Возвращает копию строки `s`, в которой первые символы каждого слова преобразованы в символы верхнего регистра, а все остальные символы – в символы нижнего регистра; смотрите также метод `str.istitle()`
`s.translate()`	Парный метод для `str.maketrans()`; подробности приводятся в тексте
`s.upper()`	Возвращает копию строки `s`, в которой все символы приведены к верхнему регистру; смотрите также метод `str.lower()`
`s.zfill(w)`	Возвращает копию строки `s`, которая, если ее длина меньше величины `w`, дополняется слева нулями до длины `w`

Метод `str.join()` может также использоваться в комбинации со встроенной функцией `reversed()`, которая переворачивает строку – например, `"".join(reversed(s))`, хотя тот же результат может быть получен более кратким оператором извлечения разреженного среза – например, `s[::-1]`.

Оператор `*` обеспечивает возможность дублирования строки:

```
>>> s = "=" * 5
>>> print(s)
=====
>>> s *= 10
>>> print(s)
==================================================
```

Как показано в примере, мы можем также использовать комбинированный оператор присваивания с дублированием.[1]

Когда оператор проверки на вхождение `in` применяется к строкам, он возвращает `True`, если операнд слева является подстрокой операнда справа или равен ему. Когда необходимо точно определить позицию подстроки в строке, можно использовать два метода. Первый метод `str.index()` возвращает позицию подстроки в строке или возбуждает исключение `ValueError`, если подстрока не будет найдена. Второй метод `str.find()` возвращает позицию подстроки в строке или `-1` в случае неудачи. Оба метода принимают искомую подстроку в качестве первого аргумента и могут принимать еще пару необязательных аргументов. Второй аргумент определяет позицию начала поиска, а третий – позицию окончания поиска.

Какой из двух методов использовать – это лишь вопрос вкуса, хотя, если выполняется поиск нескольких вхождений одной и той же подстроки, программный код, использующий метод `str.index()`, выглядит более понятным, как показано ниже:

```
def extract_from_tag(tag, line):
    opener = "<" + tag + ">"
    closer = "</" + tag + ">"
    try:
        i = line.index(opener)
        start = i + len(opener)
        j = line.index(closer, start)
        return line[start:j]
    except ValueError:
        return None
```

```
def extract_from_tag(tag, line):
    opener = "<" + tag + ">"
    closer = "</" + tag + ">"
    i = line.find(opener)
    if i != -1:
        start = i + len(opener)
        j = line.find(closer, start)
        if j != -1:
            return line[start:j]
    return None
```

[1] Строки также поддерживают оператор форматирования `%`. Этот оператор считается устаревшим и поддерживается только для облегчения перевода программ с версии Python 2 на версию Python 3. Он не используется ни в одном из тех примеров, которые приводятся в книге.

Обе версии функции `extract_from_tag()` обладают одинаковым поведением. Например, вызов `extract_from_tag("red", "what a <red>rose</red> this is")` возвращает строку «rose». В версии слева, основанной на обработке исключения, программный код, выполняющий поиск, отделен от программного кода, выполняющего обработку ошибок, а в версии справа программный код обработки строки смешивается с программным кодом обработки ошибок.

Все методы – `str.count()`, `str.endswith()`, `str.find()`, `str.rfind()`, `str.index()`, `str.rindex()` и `str.startswith()` – принимают до двух необязательных аргументов: начальную и конечную позиции. Ниже приводятся два примера эквивалентностей (предполагается, что `s` – строка):

```
s.count("m", 6) == s[6:].count("m")
s.count("m", 5, -3) == s[5:-3].count("m")
```

Как видите, строковые методы, принимающие начальную и конечную позиции, действуют со срезом строки, определяемым этими позициями.

Теперь взгляните на еще одну пару эквивалентных фрагментов, на этот раз поясняющих действие метода `str.partition()`:

```
result = s.rpartition("/")
```

```
i = s.rfind("/")
if i == -1:
    result = s, "", ""
else:
    result = s[:i], s[i], s[i + 1:]
```

Фрагменты программного кода слева и справа не совсем эквивалентны, потому что фрагмент справа создает новую переменную `i`. Обратите внимание, что имеется возможность выполнять присваивание кортежей без лишних формальностей и что в обоих случаях выполняется поиск самого последнего (крайнего справа) вхождения символа /. Если предполагать, что `s` – это строка `"/usr/local/bin/firefox"`, то оба фрагмента вернут один и тот же результат: `('/usr/local/bin', '/', 'firefox')`.

Метод `str.endswith()` (и `str.startswith()`) может использоваться с единственным строковым аргументом, например `s.startswith("From:")`, или с кортежем строк. Ниже приводится инструкция, в которой используются методы `str.endswith()` и `str.lower()` для вывода имени файла, если он является файлом изображения в формате JPEG:

```
if filename.lower().endswith((".jpg", ".jpeg")):
    print(filename, "is a JPEG image")
```

Методы семейства `is*`, такие как `isalpha()` и `isspace()`, возвращают `True`, если строка, в контексте которой они вызываются, имеет по меньшей мере один символ, и все символы в строке соответствуют определенному критерию. Например:

```
>>> "917.5".isdigit(), "".isdigit(), "-2".isdigit(), "203".isdigit()
(False, False, False, True)
```

Методы семейства `is*` работают с символами Юникода, поэтому вызов `str.isdigit()` для строк `"\N{circled digit two}03"` и `"②03"` в обоих случаях вернет `True`. По этой причине, когда метод `isdigit()` возвращает `True`, нельзя утверждать, что строка может быть преобразована в целое число.

Когда мы получаем строки из внешних источников (из других программ, из файлов, через сетевые соединения или в результате взаимодействия с пользователем), они могут содержать в начале или в конце нежелательные пробельные символы. Удалить пробельные символы, находящиеся в начале строки, можно с помощью метода `str.lstrip()`, в конце строки – с помощью метода `str.rstrip()`, а с обоих концов – с помощью метода `str.strip()`. Мы можем также передавать методам семейства `*strip` строки, в этом случае они удалят каждое вхождение каждого символа с соответствующего конца (или с обоих концов) строки. Например:

```
>>> s = "\t no parking "
>>> s.lstrip(), s.rstrip(), s.strip()
('no parking ', '\t no parking', 'no parking')
>>> "<[unbracketed]>".strip("[](){}<>")
'unbracketed'
```

Пример csv2html.py, стр. 119

Мы можем также замещать подстроки в строках, используя метод `str.replace()`. Этот метод принимает два строковых аргумента и возвращает копию строки, в контексте которой был вызван метод, где каждое вхождение строки в первом аргументе замещено строкой во втором аргументе. Если второй аргумент представляет собой пустую строку, это приведет к удалению всех вхождений строки в первом аргументе. Мы увидим примеры использования `str.replace()` и некоторых других строковых методов в примере *csv2html.py* в разделе «Примеры» в конце этой главы.

Часто бывает необходимо разбить строку на список строк. Например, у нас может иметься текстовый файл с данными, в котором одной строке соответствует одна запись, а поля внутри записи отделяются друг от друга звездочками. Реализовать такое разбиение можно с помощью метода `str.split()`, передав ему строку, по которой выполняется разбиение, в виде первого аргумента и максимальное число разбиений в виде второго, необязательного аргумента. Если второй аргумент не указан, выполняется столько разбиений, сколько потребуется. Ниже приводится пример использования метода:

```
>>> record = "Leo Tolstoy*1828-8-28*1910-11-20"
>>> fields = record.split("*")
>>> fields
['Leo Tolstoy', '1828-8-28', '1910-11-20']
```

Теперь мы можем с помощью метода `str.split()` выделить год рождения и год смерти и определить, сколько лет прожил Лев Толстой (плюс-минус один год):

```
>>> born = fields[1].split("-")
>>> born
['1828', '8', '28']
>>> died = fields[2].split("-")
>>> print("lived about", int(died[0]) - int(born[0]), "years")
lived about 82 years
```

Нам потребовалось использовать преобразование `int()`, чтобы преобразовать годы из строк в целые числа, но в остальном в этом фрагменте нет ничего сложного. Мы могли бы получить годы непосредственно из списка `fields` – например, `year_born = int(fields[1].split("-")[0])`.

В табл. 2.7 остались неописанными два метода – `str.maketrans()` и `str.translate()`. Метод `str.maketrans()` используется для создания таблицы преобразований, которая отображает одни символы в другие. Этот метод принимает один, два или три аргумента, но мы рассмотрим только простейший случай использования (с двумя аргументами), когда первый аргумент представляет строку, содержащую символы для преобразования, а второй – строку с символами, в которые нужно преобразовать. Оба аргумента должны иметь одинаковую длину. Метод `str.translate()` принимает таблицу преобразований и возвращает копию строки с символами, преобразованными в соответствии с таблицей преобразований. Ниже приводится пример преобразования бенгальских цифр в английские:

```
table = "".maketrans("\N{bengali digit zero}"
    "\N{bengali digit one}\N{bengali digit two}"
    "\N{bengali digit three}\N{bengali digit four}"
    "\N{bengali digit five}\N{bengali digit six}"
    "\N{bengali digit seven}\N{bengali digit eight}"
    "\N{bengali digit nine}", "0123456789")
print("20749".translate(table))                    # выведет: 20749
print("\N{bengali digit two}07\N{bengali digit four}"
    "\N{bengali digit nine}".translate(table))     # выведет: 20749
```

Обратите внимание на то, как в этом примере использовался характерный для Python прием конкатенации строк в вызове метода `str.maketrans()` и во втором вызове функции `print()`, что позволило нам расположить строки в нескольких строках программного кода, не используя экранирование символов перевода строки и явную операцию конкатенации.

Метод `str.maketrans()` был вызван в контексте пустой строки, потому что совершенно неважно, в контексте какой строки он вызывается – он просто обрабатывает свои аргументы и возвращает таблицу преоб-

разований.[1] Методы str.maketrans() и str.translate() могут также использоваться для удаления символов, если в третьем аргументе методу str.maketrans() передать строку с нежелательными символами. Для более сложных случаев преобразования можно было бы создать отдельные кодеки; за более подробной информацией о кодеках обращайтесь к описанию модуля codec.

В языке Python имеется еще ряд библиотечных модулей, обеспечивающих дополнительные функциональные возможности при работе со строками. Мы уже упоминали модуль unicodedata и в следующем подразделе покажем, как им пользоваться. Из других модулей, на которые следует обратить внимание, можно назвать difflib, который используется для поиска различий между двумя файлами или строками, класс io.StringIO в модуле io, который позволяет обращаться к строкам как к файлам, и модуль textwrap, который предоставляет средства обертывания и заполнения строк. Существует также модуль string, в котором имеется несколько полезных констант, таких как ascii_letters и ascii_lowercase. Примеры использования некоторых из этих модулей будут приводиться в главе 5. Кроме того, язык Python обеспечивает превосходную поддержку регулярных выражений с помощью модуля re – этой теме целиком посвящена глава 12.

Форматирование строк с помощью метода str.format()

Метод str.format() представляет собой очень мощное и гибкое средство создания строк. Использование метода str.format() в простых случаях не вызывает сложностей, но для более сложного форматирования нам необходимо изучить синтаксис форматирования.

Метод str.format() возвращает новую строку, *замещая поля* в контекстной строке соответствующими аргументами. Например:

```
>>> "The novel '{0}' was published in {1}".format("Hard Times", 1854)
"The novel 'Hard Times' was published in 1854"
```

Каждое замещаемое поле идентифицируется именем поля в фигурных скобках. Если в качестве имени поля используется целое число, оно определяет порядковый номер аргумента, переданного методу str.format(). Поэтому в данном случае поле с именем 0 было замещено первым аргументом, а поле с именем 1 – вторым аргументом.

Если бы нам потребовалось включить фигурные скобки в строку формата, мы могли бы сделать это, дублируя их, как показано ниже:

```
>>> "{{{0}}} {1} ;-}}".format("I'm in braces", "I'm not")
"{I'm in braces} I'm not ;-}"
```

[1] Примечание для программистов, использующих объектно-ориентированный стиль: str.maketrans() – это метод класса.

Если попытаться объединить строку и число, интерпретатор Python совершенно справедливо возбудит исключение `TypeError`. Но это легко можно сделать с помощью метода `str.format()`:

```
>>> "{0}{1}".format("The amount due is $", 200)
'The amount due is $200'
```

С помощью `str.format()` мы также легко можем объединять строки (хотя для этой цели лучше подходит метод `str.join()`):

```
>>> x = "three"
>>> s ="{0} {1} {2}"
>>> s = s.format("The", x, "tops")
>>> s
'The three tops'
```

Здесь мы использовали несколько строковых переменных, тем не менее в большинстве примеров с применением метода `str.format()` в этом разделе мы будем использовать строковые литералы – исключительно ради удобства; но вы должны помнить, что в любых примерах, где используются строковые литералы, точно также можно было бы использовать и строковые переменные.

Замещаемые поля могут определять одним из следующих способов:

```
{field_name}
{field_name!conversion}
{field_name:format_specification}
{field_name!conversion:format_specification}
```

Следует заметить, что замещаемые поля могут *содержать* другие замещаемые поля. Вложенные замещаемые поля не могут иметь какое-либо форматирование – их назначение состоит в том, чтобы позволить динамически определять параметры форматирования. Примеры использования вложенных полей будут представлены при подробном изучении спецификаторов формата. А теперь приступим к изучению каждой части замещаемого поля, начав с его имени.

Имена полей

Имя поля может быть либо целым числом, соответствующим одному из аргументов метода `str.format()`, либо именем одного из именованных аргументов метода. Именованные аргументы мы будем рассматривать в главе 4, но в них нет ничего сложного, поэтому для полноты картины ниже приводится пара примеров:

```
>>> "{who} turned {age} this year".format(who="She", age=88)
'She turned 88 this year'
>>> "The {who} was {0} last week".format(12, who="boy")
'The boy was 12 last week'
```

В первом примере используются два именованных аргумента, who и age, а во втором – один позиционный аргумент (единственный тип аргументов, который мы использовали до сих пор) и один именованный аргумент. Обратите внимание, что в списке аргументов именованные аргументы всегда следуют после позиционных, и, конечно же, мы можем использовать в строке формата любые аргументы и в любом порядке.

Имена полей могут ссылаться на коллекции, такие как списки. В подобных случаях для идентификации требуемого элемента можно использовать индексы (но не срезы!):

```
>>> stock = ["paper", "envelopes", "notepads", "pens", "paper clips"]
>>> "We have {0[1]} and {0[2]} in stock".format(stock)
'We have envelopes and notepads in stock'
```

Поле с именем 0 ссылается на позиционный аргумент, поэтому полю {0[1]} соответствует второй элемент в списке stock, а полю {0[2]} – третий элемент в списке stock.

Тип dict, стр. 151

Позднее мы познакомимся со словарями в языке Python. Они хранят данные в виде пар «ключ-значение», а поскольку они также могут использоваться в качестве аргументов метода str.format(), приведем короткий пример. Не волнуйтесь, если что-то покажется вам непонятным, – вы все поймете, как только прочтете главу 3.

```
>>> d = dict(animal="elephant", weight=12000)
>>> "The {0[animal]} weighs {0[weight]}kg".format(d)
'The elephant weighs 12000kg'
```

При обращении к элементам словаря используется ключ, точно так же как используется целочисленный индекс при обращении к элементам списков и кортежей.

Мы можем также обращаться к атрибутам объектов по их именам. Предположим, что мы импортировали модули math и sys, тогда можно будет выполнить такое форматирование:

```
>>> "math.pi={0.pi} sys.maxunicode={1.maxunicode}".format(math, sys)
'math.pi=3.14159265359 sys.maxunicode=65535'
```

Таким образом, синтаксис имен полей позволяет обращаться как к позиционным, так и к именованным аргументам, которые передаются методу str.format(). Если в качестве аргументов передаются коллекции, такие как списки или словари, или объекты, имеющие атрибуты, то имеется возможность обращаться к любой части коллекции, используя нотацию [] или ., как это показано на рис. 2.5.

Преобразования

Числа типа Decimal, стр. 82

Когда мы обсуждали числа типа decimal.Decimal, мы отмечали, что такие числа могут выводиться одним из двух способов, например:

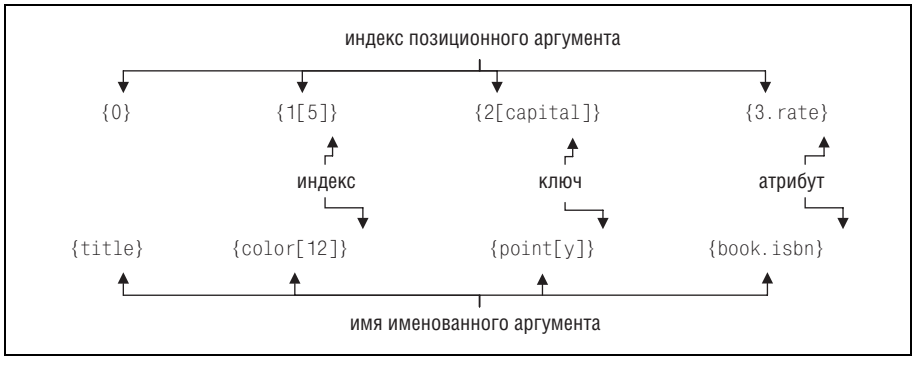

Рис. 2.5. *Примеры спецификаторов формата в именах полей с примечаниями*

```
>>> decimal.Decimal("3.4084")
Decimal('3.4084')
>>> print(decimal.Decimal("3.4084"))
3.4084
```

Первый способ отображения значения типа `decimal.Decimal` – это репрезентативная форма. Ее назначение состоит в том, чтобы предоставить строку, которая может быть воспринята интерпретатором Python для воссоздания объекта, который она представляет. Программы на языке Python могут прибегать к интерпретации отдельных фрагментов программного кода или целых программ, поэтому в некоторых ситуациях такая возможность может оказаться совсем нелишней. Не все объекты могут быть представлены в форме, позволяющей выполнить их воспроизведение; в таких случаях они предоставляются в виде строки, заключенной в угловые скобки. Например, репрезентативной формой модуля `sys` является строка `"<module 'sys' (built-in)>"`.

Функция `eval()`, стр. 400

Второй способ отображения значения типа `decimal.Decimal` – это строковая форма. Эта форма предназначена для представления человеку, поэтому основное ее назначение состоит в том, чтобы отображать информацию в том виде, в каком она будет иметь определенный смысл для человека. Если тип данных не имеет строковой формы представления, но необходима именно строка, Python будет использовать репрезентативную форму.

Встроенные типы данных языка Python знакомы с методом `str.format()` и при передаче их в качестве аргументов этому методу возвращают соответствующую строку для отображения. В том, чтобы добавить в метод `str.format()` поддержку собственных типов данных, нет ничего сложного, в этом вы сможете убедиться в главе 6. Кроме того, имеется

возможность переопределять привычное поведение типов данных и, по желанию, заставлять их возвращать строковую или репрезентативную форму представления. Этого можно добиться путем добавления в поля спецификаторов преобразования. В настоящее время существует три таких спецификатора: *s* – для принудительного вывода строковой формы, *r* – для принудительного вывода репрезентативной формы и *a* – для принудительного вывода репрезентативной формы, но с использованием только символов ASCII. Ниже приводится пример использования этих спецификаторов:

```
>>> "{0} {0!s} {0!r} {0!a}".format(decimal.Decimal("93.4"))
"93.4 93.4 Decimal('93.4') Decimal('93.4')"
```

В данном случае строковая форма объекта `decimal.Decimal` совпадает со строкой, предоставляемой для метода `str.format()`, что является вполне обычным делом. Кроме того, в данном конкретном примере нет никакой разницы между репрезентативной формой и репрезентативной формой ASCII, поскольку в обоих случаях используются только символы ASCII.

Ниже приводится еще один пример, но на сей раз в нем используется строка, содержащая заголовок фильма «翻訳で失われる» и хранящаяся в переменной `movie`. Если вывести строку как `"{0}".format(movie)`, она будет выведена без изменений, но если необходимо избежать вывода символов, не входящих в набор ASCII, можно использовать либо вызов `ascii(movie)`, либо выводить ее как `"{0!a}".format(movie)` – в обоих случаях будет получена строка `'\u7ffb\u8a33\u3067\u5931\u308f\u308c\u308b'`.

К настоящему моменту мы знаем, как помещать значения переменных в строку формата и как принудительно выбирать строковую или репрезентативную форму представления. Теперь мы готовы перейти к рассмотрению приемов форматирования самих значений.

Спецификаторы формата

Форматирование целых чисел, чисел с плавающей точкой и строк часто бывает вполне удовлетворительным. Но если нам требуется более тонкое управление форматированием, мы легко можем реализовать его с помощью спецификаторов формата. Мы будем отдельно рассматривать форматирование строк, целых чисел и чисел с плавающей точкой, чтобы было легче разобраться в деталях. Общий синтаксис, который в равной мере относится ко всем этим типам данных, показан на рис. 2.6.

В случае строк мы можем управлять символом-заполнителем, выравниванием внутри поля, а также минимальной и максимальной шириной поля. Спецификаторы формата для строк начинаются с символа двоеточия (`:`), за которым следует пара необязательных символов – символ-заполнитель (который не может быть правой фигурной скобкой `}`) и символ выравнивания (`<` – по левому краю, `^` – по центру и `>` –

заполнитель	*выравнивание*	*знак*	*#*	*0*	*ширина*	*.точность*	*тип*
Любой символ, кроме }	< по левому краю; > по правому краю; ^ по центру = заполнять нулями пространство между знаком числа и первой значащей цифрой	+ всегда выводить знак; – знак выводится, только когда необходимо; “ ” пробел или знак «–»	префикс для целых чисел 0b, 0o, or 0x	Дополнение чисел нулями	Минимальная ширина поля	Максимальная ширина поля для строк; количество знаков после запятой для чисел с плавающей точкой	int b, c, d, n, o, x, X; floats e, E, f, g, G, n, %

Рис. 2.6. *Спецификатор формата в общем виде*

по правому краю). Далее следует необязательное число, определяющее минимальную ширину поля вывода, и далее, при желании, через точку можно указать максимальную ширину поля вывода.

Обратите внимание, что если указан символ-заполнитель, то должно быть указано и направление выравнивания. Мы опустили части спецификатора формата, определяющие знак и тип, потому что на строки они не оказывают никакого влияния. Вполне безопасно (хотя и бессмысленно) использовать одно только двоеточие без указания дополнительных элементов спецификатора.

Рассмотрим несколько примеров:

```
>>> s = "The sword of truth"
>>> "{0}".format(s)     # форматирование по умолчанию
'The sword of truth'
>>> "{0:25}".format(s)  # минимальная ширина поля вывода 25
'The sword of truth '
>>> "{0:>25}".format(s) # выравнивание по правому краю, минимальная ширина 25
' The sword of truth'
>>> "{0:^25}".format(s) # выравнивание по центру, минимальная ширина 25
' The sword of truth '
>>> "{0:-^25}".format(s) # - заполнитель, по центру, минимальная ширина 25
'---The sword of truth----'
>>> "{0:.<25}".format(s) # . заполнитель, по левому краю, минимальная ширина 25
'The sword of truth.......'
>>> "{0:.10}".format(s) # максимальная ширина поля вывода 10
'The sword '
```

В предпоследнем примере нам пришлось определить выравнивание по левому краю (даже при том, что это – значение по умолчанию). В противном случае спецификатор формата приобрел бы вид `:.25` и просто означал бы максимальную ширину поля вывода 25 символов.

Как уже отмечалось ранее, внутри спецификатора формата можно использовать замещаемые поля. Это делает возможным динамически определять формат вывода. Ниже приводится пример, демонстрирующий два способа определить максимальную ширину строки с помощью переменной `maxwidth`:

```
>>> maxwidth = 12
>>> "{0}".format(s[:maxwidth])
'The sword of'
>>> "{0:.{1}}".format(s, maxwidth)
'The sword of'
```

В первом случае используется обычная операция извлечения среза, во втором – вложенное замещаемое поле.

Применительно к целым числам спецификаторы формата позволяют управлять символом-заполнителем, выравниванием внутри поля вывода, отображением знака числа, минимальной шириной поля и основанием системы счисления.

Спецификаторы формата для целых чисел начинаются с двоеточия, после которого может следовать пара необязательных символов – символ-заполнитель (который не может быть символом закрывающей фигурной скобки `}`) и символ выравнивания (`<` – по левому краю, `^` – по центру, `>` – по правому краю и `=` – указывающий на необходимость заполнять пространство между знаком числа и первой значащей цифрой). Далее следует необязательный символ знака числа: «`+`» – говорит об обязательной необходимости вывода знака числа, «`-`» – знак выводится только для отрицательных чисел, и пробел говорит о том, что для положительных чисел вместо знака числа должен выводиться пробел, а для отрицательных чисел – знак «`-`». Далее следует значение минимальной ширины поля, которому может предшествовать символ «`#`» с обозначением системы счисления (двоичная, восьмеричная или шестнадцатеричная) и символ «`0`» – в случае необходимости дополнения числа нулями слева. Если число должно выводиться в системе счисления, отличной от десятичной, необходимо указать символ типа системы счисления: «`b`» – для двоичной, «`o`» – для восьмеричной, «`x`» – для шестнадцатеричной с символами в нижнем регистре и «`X`» – для шестнадцатеричной с символами в верхнем регистре. Для полноты картины следует заметить, что допускается использовать символ «`d`», обозначающий десятичную систему счисления. Существует еще два символа типа: «`c`», который означает, что должен выводиться символ Юникода, соответствующий целому числу, и «`n`» – когда необходимо обеспечить вывод чисел с учетом региональных настроек.

Дополнение нулями слева можно реализовать двумя способами:

```
>>> "{0:0=12}".format(8749203) # 0 - символ-заполнитель, минимальная ширина 12
'000008749203'
>>> "{0:0=12}".format(-8749203)# 0 - символ-заполнитель, минимальная ширина 12
'-00008749203'
```

```
>>> "{0:012}".format(8749203)  # дополнение 0 и минимальная ширина 12
'000008749203'
>>> "{0:012}".format(-8749203) # дополнение 0 и минимальная ширина 12
'-00008749203'
```

В первых двух примерах 0 определяется как символ-заполнитель, которым заполняется пространство между знаком числа и первой значащей цифрой (=). Во вторых двух примерах определяется минимальная ширина поля 12 символов и признак необходимости дополнения нулями.

Ниже приводится несколько примеров управления выравниванием:

```
>>> "{0:*<15}".format(18340427)  # * символ-заполнитель, выравнивание
'18340427*******'                # по левому краю, минимальная ширина 15
>>> "{0:*>15}".format(18340427)  # * символ-заполнитель, выравнивание
'*******18340427'                # по правому краю, минимальная ширина 15
>>> "{0:*^15}".format(18340427)  # * символ-заполнитель, выравнивание
'***18340427****'                # по центру, минимальная ширина 15
>>> "{0:*^15}".format(-18340427) # * символ-заполнитель, выравнивание
'***-18340427***'                # по центру, минимальная ширина 15
```

Ниже приводится несколько примеров управления выводом знака числа:

```
>>> "[{0: }] [{1: }]".format(539802, -539802) # пробел или знак "-"
'[ 539802] [-539802]'
>>> "[{0:+}] [{1:+}]".format(539802, -539802) # знак выводится принудительно
'[+539802] [-539802]'
>>> "[{0:-}] [{1:-}]".format(539802, -539802) # знак "-" выводится только
'[539802] [-539802]'                          # при необходимости
```

Далее следуют два примера использования символов управления типом:

```
>>> "{0:b} {0:o} {0:x} {0:X}".format(14613198)
'110111101111101011001110 67575316 deface DEFACE'
>>> "{0:#b} {0:#o} {0:#x} {0:#X}".format(14613198)
'0b110111101111101011001110 0o67575316 0xdeface 0XDEFACE'
```

Для целых чисел невозможно определить максимальную ширину поля вывода, потому что в противном случае это может повлечь необходимость отсечения значащих цифр числа и вывод числа, не имеющего смысла.

Последний символ управления форматом вывода целых чисел (доступный также для чисел с плавающей точкой) – это символ «n». Он имеет то же действие, что и символ «d» в случае вывода целых чисел или символ «g» в случае вывода чисел с плавающей точкой. Отличительной особенностью символа «n» является то, что он учитывает региональные настройки, то есть использует характерный для текущего региона символ-разделитель целой и дробной части числа и разделитель разрядов. Регион, используемый по умолчанию, называется «C», и для этого региона в качестве разделителя целой и дробной части числа используется точка, а в качестве разделителя разрядов – пустая строка.

Чтобы иметь возможность принимать во внимание региональные настройки пользователя, в начале программы в качестве двух первых выполняемых инструкций можно добавить следующие две строки:[1]

```
import locale
locale.setlocale(locale.LC_ALL, "")
```

Передавая пустую строку в качестве названия региона, мы тем самым предлагаем интерпретатору попытаться автоматически определить регион пользователя (например, путем определения значения переменной окружения `LANG`) и перейти на использование региона «C» в случае неудачи. Ниже приводятся несколько примеров, демонстрирующих влияние различных региональных настроек на вывод целых и вещественных чисел:

```
x, y = (1234567890, 1234.56)
locale.setlocale(locale.LC_ALL, "C")
c = "{0:n} {1:n}".format(x, y)      # c == "1234567890 1234.56"
locale.setlocale(locale.LC_ALL, "en_US.UTF-8")
en = "{0:n} {1:n}".format(x, y)     # en == "1,234,567,890 1,234.56"
locale.setlocale(locale.LC_ALL, "de_DE.UTF-8")
de = "{0:n} {1:n}".format(x, y)     # de == "1.234.567.890 1.234,56"
```

Символ «`n`» очень удобно использовать с целыми числами, но при выводе чисел с плавающей точкой он имеет ограниченное применение, потому что большие вещественные числа выводятся в экспоненциальной форме.

При выводе чисел с плавающей точкой спецификаторы формата дают возможность управлять символом-заполнителем, выравниванием в пределах поля вывода, выводом знака числа, минимальной шириной поля, числом знаков после десятичной точки и формой представления – простая, экспоненциальная или в виде процентов.

Для форматирования чисел с плавающей точкой используются те же самые спецификаторы, что и для целых чисел, с двумя отличиями в конце. После необязательного значения минимальной ширины поля вывода можно указать число знаков после десятичной точки, добавив символ точки и целое число. В самом конце мы можем указать символ типа: «`e`» – для вывода числа в экспоненциальной форме, с символом «*e*» в нижнем регистре; «`E`» – для вывода числа в экспоненциальной форме, с символом «*E*» в верхнем регистре; «`f`» – для вывода числа в стандартной форме, «`g`» – для вывода числа в «общей» форме, то есть для небольших чисел действует как символ «`f`», а для очень больших – как символ «`e`», и «`G`» – то же самое, что символ «`g`», только использу-

[1] В программах, имеющих несколько потоков выполнения, функцию `locale.setlocale()` лучше вызывать всего один раз, на этапе запуска программы, и еще до того, как будут запущены дополнительные потоки, поскольку эту функцию обычно небезопасно вызывать в многопоточном окружении.

ется формат либо «f», либо «E». Кроме того, допускается использовать символ «%», при использовании которого выводимое число умножается на 100 и для вывода применяется формат «f», с добавлением символа «%» в конце числа.

Ниже приводятся несколько примеров вывода числа в экспоненциальной и стандартной форме:

```
>>> amount = (10 ** 3) * math.pi
>>> "[{0:12.2e}] [{0:12.2f}]".format(amount)
'[ 3.14e+03] [ 3141.59]'
>>> "[{0:*>12.2e}] [{0:*>12.2f}]".format(amount)
'[****3.14e+03] [*****3141.59]'
>>> "[{0:*>+12.2e}] [{0:*>+12.2f}]".format(amount)
'[***+3.14e+03] [****+3141.59]'
```

В первом примере установлена минимальная ширина поля вывода 12 символов и 2 знака после десятичной точки. Второй пример построен на основе первого и к нему добавлен вывод символа-заполнителя «*». При использовании символа-заполнителя необходимо указывать символ выравнивания, поэтому мы указали выравнивание по правому краю (даже при том, что этот способ выравнивания используется по умолчанию для чисел). Третий пример построен на основе двух предыдущих, в нем добавлен символ «+» управления принудительным выводом знака числа.

К моменту написания этих строк в языке Python отсутствовали средства прямого управления форматированием комплексных чисел. Однако мы легко можем решить эту проблему, форматируя действительную и мнимую части как отдельные числа с плавающей точкой. Например:

```
>>> "{0.real:.3f}{0.imag:+.3f}j".format(4.75917+1.2042j)
'4.759+1.204j'
>>> "{0.real:.3f}{0.imag:+.3f}j".format(4.75917-1.2042j)
'4.759-1.204j'
```

Мы обращаемся к каждому атрибуту комплексного числа по отдельности и форматируем их как числа с плавающей точкой с тремя знаками после запятой. Кроме того, мы принудительно выводим знак мнимой части, добавляя символ *j*.

Пример: print_unicode.py

В предыдущих подразделах мы детально исследовали спецификаторы формата для метода `str.format()` и видели достаточно много фрагментов программного кода, демонстрирующих аспекты их применения на практике. В этом подподразделе мы рассмотрим небольшой, но достаточно поучительный пример использования метода `str.format()`, в котором мы увидим применение спецификаторов формата в реальном контексте. В примере также используются некоторые строковые методы,

с которыми мы познакомились в предыдущем разделе, и вводится в использование функция из модуля unicodedata.[1]

Эта программа состоит всего из 25 строк выполняемого программного кода. Она импортирует два модуля, sys и unicodedata, и определяет одну функцию – print_unicode_table(). Рассмотрение примера мы начнем с запуска программы, чтобы увидеть, что она делает; затем мы рассмотрим программный код в конце программы, где выполняется вся фактическая работа; и в заключение рассмотрим функцию, определяемую в программе.

```
print_unicode.py spoked
decimal  hex  chr                name
------- ----- --- ------------------------------------
  10018 2722 ✢ Four Teardrop-Spoked Asterisk
  10019 2723 ✣ Four Balloon-Spoked Asterisk
  10020 2724 ✤ Heavy Four Balloon-Spoked Asterisk
  10021 2725 ✥ Four Club-Spoked Asterisk
  10035 2733 ✳ Eight Spoked Asterisk
  10043 273B ✻ Teardrop-Spoked Asterisk
  10044 273C ✼ Open Centre Teardrop-Spoked Asterisk
  10045 273D ✽ Heavy Teardrop-Spoked Asterisk
  10051 2743 ❃ Heavy Teardrop-Spoked Pinwheel Asterisk
  10057 2749 ❉ Balloon-Spoked Asterisk
  10058 274A ❊ Eight Teardrop-Spoked Propeller Asterisk
  10059 274B ❋ Heavy Eight Teardrop-Spoked Propeller Asterisk
```

При запуске без аргументов программа выводит таблицу всех символов Юникода, начиная с пробела и до символа с наибольшим возможным кодом. При запуске с аргументом, как показано в примере, выводятся только те строки таблицы, где в названии символов Юникода содержится значение строки-аргумента, переведенной в нижний регистр.

```
word = None
if len(sys.argv) > 1:
    if sys.argv[1] in ("-h", "--help"):
        print("usage: {0} [string]".format(sys.argv[0]))
        word = 0
    else:
        word = sys.argv[1].lower()
```

[1] Эта программа предполагает, что консоль настроена на работу в кодировке UTF-8. К сожалению, консоль в операционной системе Windows имеет весьма ограниченную поддержку UTF-8, а консоль в системе Mac OS X по умолчанию использует кодировку Apple Roman. Чтобы обойти эти ограничения, в состав примеров к книге включен файл *print_unicode_uni.py* – версия программы, выполняющая вывод в файл, который затем может быть открыт в редакторе, таком как IDLE, поддерживающем кодировку UTF-8.

Глава 7, работа с файлами, стр. 334

```
if word != 0:
    print_unicode_table(word)
```

После инструкций импортирования и определения функции `print_unicode_table()` выполнение достигает программного кода, показанного выше. Сначала предположим, что пользователь не указал в командной строке искомое слово. Если аргумент командной строки присутствует и это `-h` или `--help`, программа выводит информацию о порядке использования и устанавливает флаг `word` в значение `0`, указывая тем самым, что работа завершена. В противном случае в переменную `word` записывается копия аргумента, введенного пользователем, с преобразованием всех символов в нижний регистр. Если значение `word` не равно `0`, программа выводит таблицу.

При выводе информации о порядке использования применяется спецификатор формата, который представляет собой простое имя формата, в данном случае – порядковый номер позиционного аргумента. Мы могли бы записать эту строку, как показано ниже:

```
print("usage: {0[0]} [string]".format(sys.argv))
```

При таком подходе первый символ `0` соответствует порядковому номеру позиционного аргумента, а `[0]` – это индекс элемента *внутри* аргумента, и такой прием сработает, потому что `sys.argv` является списком.

```
def print_unicode_table(word):
    print("decimal  hex  chr {0:^40}".format("name"))
    print("------- ----- --- {0:-<40}".format(""))

    code = ord(" ")
    end = sys.maxunicode

    while code < end:
        c = chr(code)
        name = unicodedata.name(c, "*** unknown ***")
        if word is None or word in name.lower():
            print("{0:7} {0:5X} {0:^3c} {1}".format(
                code, name.title()))
        code += 1
```

Мы использовали пару пустых строк исключительно для улучшения удобочитаемости. Первые две строки функции выводят строки заголовка. Первый вызов `str.format()` выводит текст «name», отцентрированный в поле вывода, шириной 40 символов, а второй вызов выводит пустую строку в поле шириной 40 символов, используя символ «-» в качестве символа-заполнителя, с выравниванием по левому краю. (Мы вынуждены указывать символ выравнивания, когда задается символ-заполнитель.) Как вариант, вторую строку функции можно было записать, как показано ниже:

```
print("------- ----- --- {0}".format("-" * 40))
```

Здесь мы использовали оператор дублирования строки (*), чтобы создать необходимую строку, и просто вставили ее в строку формата. В третьем случае можно было бы просто ввести 40 символов «-» и использовать простой литерал строки.

Кодировки символов, стр. 112

Текущий код символа Юникода сохраняется в переменной `code`, которая инициализируется кодом пробела (`0x20`). В переменную `end` записывается максимально возможный код символа Юникода, который может принимать разные значения в зависимости от того, какая из кодировок (UCS-2 или UCS-4) использовалась при компиляции Python.

Внутри цикла `while` с помощью функции `chr()` мы получаем символ Юникода, соответствующий числовому коду. Функция `unicodedata.name()` возвращает название заданного символа Юникода, во втором необязательном аргументе передается имя, которое будет использовано в случае, когда имя символа не определено.

Если пользователь не указывает аргумент командной строки (`word is None`) или аргумент был указан и он входит в состав копии имени символа Юникода, в которой все символы приведены к нижнему регистру, то выводится соответствующая строка таблицы.

Мы передаем переменную `code` методу `str.format()` один раз, но в строке формата она используется трижды. Первый раз – при выводе значения `code` как целого числа в поле с шириной 7 символов (по умолчанию в качестве символа-заполнителя используется пробел, поэтому нет необходимости явно указывать его). Второй раз – при выводе значения `code` как целого числа в шестнадцатеричном формате символами верхнего регистра в поле шириной 5 символов. И третий раз – при выводе символа Юникода, соответствующего значению `code`, с помощью спецификатора формата «c», отцентрированного в поле с минимальной шириной 3 символа. Обратите внимание, что нам не потребовалось указывать тип «d» в первом спецификаторе формата, потому что он подразумевается по умолчанию для целых чисел. Второй аргумент – это имя символа Юникода, которое выводится с помощью метода `str.title()`, в результате которого первый символ каждого слова преобразуется к верхнему регистру, а остальные символы – к нижнему.

Теперь, когда мы познакомились с универсальным методом `str.format()`, мы можем с успехом использовать его на протяжении остальной части книги.

Кодировки символов

Так или иначе, но компьютеры могут хранить информацию только в виде байтов, то есть в виде 8-битовых значений в диапазоне от `0x00` до `0xFF`. Каждый символ должен быть представлен некоторым образом в терминах байтов. На заре развития вычислительной техники были

разработаны схемы кодирования символов, в которых каждому конкретному был поставлен в соответствие байт с конкретным значением. Например, в кодировке ASCII символ *A* представлен байтом со значением `0x41`, *B* — `0x42` и т. д. В Западной Европе часто использовалась кодировка Latin-1, ее первые 127 символов совпадали с 7-битовой кодировкой ASCII, а остальные значения представляли символы с умляутами и другие символы, необходимые европейцам. За долгие годы появилось множество кодировок, которые используются до сих пор.

К сожалению, наличие такого разнообразия кодировок оказалось очень неудобным, особенно при разработке интернационализируемого программного обеспечения. Одним из решений, которое было принято практически повсеместно, стало применение кодировки Юникод. В кодировке Юникод каждому символу ставится в соответствие целое число, то есть его код, как и в кодировках, разработанных ранее, но Юникод не ограничивается использованием одного байта на символ и потому способен обеспечить единую систему представления каждого символа любого языка. А для обеспечения обратной совместимости первые 127 символов Юникода совпадают со 127 символами 7-битовой кодировки ASCII.

Но как хранятся символы Юникода? В настоящее время определено немногим более 1 миллиона символов Юникода, поэтому даже 32-битовых целых чисел со знаком более чем достаточно для представления любого кода в кодировке Юникод. Таким образом, самый простой способ хранения символов Юникода заключается в использовании последовательностей 32-битовых целых чисел, по одному целому числу на символ. Такое представление очень удобно хранить в памяти, потому что мы получаем массив 32-битовых чисел, элементы которого имеют однозначное соответствие с символами. Но тогда если текст в файлах или передаваемый по сети в основном содержит символы 7-битовой кодировки ASCII, то три из четырех переданных байтов будут нулевыми (`0x00`). Чтобы избежать появления такого большого объема ненужной информации, сама кодировка Юникод имеет несколько представлений.

В памяти символы Юникода хранятся либо в формате UCS-2 (по сути, 16-битовые целые беззнаковые числа), способном представить первые 65 535 кодов символов, или в формате UCS-4 (32-битовые целые числа), способном представить все коды символов, которых к моменту написания этих строк было 1 114 111. Выбор того или иного формата производится на этапе компиляции РуП122

thon. (Если значение `sys.maxunicode` равно 65 535, значит Python компилировался с поддержкой формата UCS-2).

При сохранении данных в виде файлов или при передаче по сети используется более сложный формат представления. При использовании Юникода коды символов могут кодироваться с помощью кодировки UTF-8, в которой первые 127 символов кодируются однобайтовыми

значениями, а остальные – двумя или более байтами. Кодировка UTF-8 очень компактна для английского текста, и если в нем используются только символы из 7-битового набора ASCII, то файлы с текстом в кодировке UTF-8 ничем не отличаются от файлов в кодировке ASCII. Другая популярная кодировка – UTF-16. В ней для кодирования значительной части символов используются два байта и для остальных – четыре байта. Она более компактна для некоторых азиатских языков, чем UTF-8, но, в отличие от нее, текст в кодировке UTF-16 должен начинаться с признака, указывающего порядок следования байтов, чтобы при чтении кодов можно было определить, какой порядок следования пар байтов используется – прямой (big-endian) или обратный (little-endian). Кроме того, по-прежнему широко используются старые кодировки, такие как GB2312, ISO-8859-5, Latin-1.

Метод `str.encode()` возвращает последовательность байтов, фактически – объект типа `bytes`, о котором будет рассказываться в главе 7, закодированных в соответствии с кодировкой, заданной в качестве аргумента. С помощью этого метода можно глубже понять различия между кодировками и понять, почему неправильные предположения о кодировке могут приводить к появлению ошибок:

```
>>> artist = "Tage Åsefб"
>>> artist.encode("Latin1")
b'Tage \xc5s\xe9n'
>>> artist.encode("CP850")
b'Tage \x8fs\x82n'
>>> artist.encode("utf8")
b'Tage \xc3\x85s\xc3\xa9n'
>>> artist.encode("utf16")
b'\xff\xfeT\x00a\x00g\x00e\x00 \x00\xc5\x00s\x00\xe9\x00n\x00'
```

Символ «*b*» перед открывающей кавычкой указывает, что это не строковый литерал, а литерал типа `bytes`. Для удобства при создании литералов типа `bytes` мы можем смешивать печатаемые символы ASCII с экранированными шестнадцатеричными значениями.

Мы не можем представить имя «Tage Åsefб» с помощью символов ASCII, потому что в этом наборе отсутствует символ «*Å*», как и любые другие символы с умляутами, поэтому при попытке сделать это возбуждается исключение `UnicodeEncodeError`. Кодировка Latin-1 (известная так же, как ISO-8859-1) использует для представления символов 8-битовые значения, и в ней присутствуют все символы, необходимые для представления данного имени. С другой стороны, артисту Ernoв Bank повезло меньше, так как символ «*ő*» отсутствует в наборе символов Latin-1. Конечно, оба имени благополучно могут быть представлены в кодировке Юникод. Примечательно, что при использовании кодировки UTF-16 первые два байта являются признаком порядка следования байтов – они используются функцией декодирования, чтобы определить, какой порядок следования используется, прямой или обратный, и выполнить декодирование соответствующим образом.

Следует отметить пару важных особенностей, присущих методу `str.encode()`. Первый аргумент (имя кодировки) не чувствителен к регистру символов, а символы дефиса и подчеркивания в имени считаются эквивалентными, поэтому имена «us-ascii» и «US_ASCII» рассматриваются как одно и то же имя. Кроме того, для одной и той же кодировки может иметься множество альтернативных названий: например, названия «latin», «latin1», «latin_1», «ISO-8859-1», «CP819» и некоторые другие обозначают кодировку «Latin-1». Метод может также принимать второй необязательный аргумент, который сообщает, как следует обрабатывать ошибки. Например, мы можем закодировать любую строку в кодировке ASCII, передав во втором аргументе «ignore» или «replace» – ценой потери данных, или без потерь, передав строку «backslashreplace» – в этом случае символы, не входящие в набор ASCII, будут представлены последовательностями `\x`, `\u` и `\U`. Например, вызов `artist.encode("ascii", "ignore")` вернет `b'Tage sn'`, вызов `artist.encode("ascii", "replace")` вернет `b'Tage ?s?n'`, а вызов `artist.encode("ascii", "backslashreplace")` вернет `b'Tage \xc5s\xe9n'`. (Точно так же мы могли бы получить строку ASCII-символов с помощью вызова `"{0!a}".format(artist)`, который вернет `'Tage \xc5s\xe9n'`.)

В дополнение к методу `str.encode()` имеется метод `bytes.decode()` (а также `bytearray.decode()`), который возвращает строку с байтами, декодированными при помощи заданной кодировки. Например:

```
>>> print(b"Tage \xc3\x85s\xc3\xa9n".decode("utf8"))
Tage Åsen
>>> print(b"Tage \xc5s\xe9n".decode("latin1"))
Tage Åsen
```

Различия между 8-битовыми кодировками Latin-1, CP850 (кодировка IBM PC) и кодировкой UTF-8 очевидно доказывают, что выбор кодировки наугад едва ли может считаться успешной стратегией. К счастью, кодировка UTF-8 фактически уже стала стандартом для простых текстовых файлов, благодаря чему следующие поколения, вероятно, даже не узнают, что некогда существовали другие кодировки.

Для файлов с расширением *.py* используется кодировка UTF-8, поэтому Python всегда знает, какая кодировка используется для представления строковых литералов. Это означает, что мы можем использовать в своих строках любые символы Юникода, поддерживаемые нашим текстовым редактором.[1]

Когда Python читает данные из внешних источников, например из сетевых сокетов, он не может заранее знать, какая кодировка используется,

[1] Вполне возможно использовать и другие кодировки. Подробности можно найти в документе «Python Tutorial», в разделе «Source Code Encoding». (*http://docs.python.org/3.0/tutorial/interpreter.html#source-code-encoding. – Прим. перев.*)

поэтому он возвращает байты, которые мы можем декодировать нужным образом. В отношении текстовых файлов Python использует более дружественный подход, используя локальную кодировку, если мы не указываем ее явно.

К счастью, некоторые форматы файлов явно определяют свою кодировку. Например, мы можем предположить, что XML-файл использует кодировку UTF-8, если в нем отсутствует директива `<?xml?>`, явно указывающая другую кодировку. Поэтому при чтении XML-файлов мы можем извлекать, скажем первые 1000 байтов, отыскивать определение кодировки и, если оно присутствует, декодировать содержимое файла в соответствии с указанной кодировкой; в противном случае переходить на использование кодировки UTF-8. Такой прием должен срабатывать для любых XML-файлов или простых текстовых файлов, в которых используется любая из однобайтовых кодировок, поддерживаемых Python, за исключением кодировок, основанных на EBCDIC (CP424, CP500), и некоторых других (CP037, CP864, CP865, CP1026, CP1140, HZ, SHIFT-JIS-2004, SHIFT-JISX0213). К сожалению, такой подход неприменим в случае использования многобайтовых кодировок (таких как UTF-16 и UTF-32). В каталоге пакетов Python (Python Package Index), *pypi.python.org/pypi*, имеется по крайней мере два пакета, позволяющих автоматически определять кодировку файлов.

Примеры

В этом разделе мы будем использовать знания, полученные в этой и в предыдущей главах, чтобы представить две маленькие, но законченные программы, соединяющие в себе все, что мы узнали до сих пор. Первая программа имеет некоторое отношение к математике, но она очень короткая и занимает всего 35 строк. Вторая связана с обработкой текста и имеет более существенный объем – в ней определяется семь функций и содержит она около 80 строк программного кода.

quadratic.py

Квадратные уравнения – это уравнения вида $ax^2 + bx + c = 0$, где $a \neq 0$, описывающие параболу. Корни таких уравнений находятся по формуле

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Часть формулы $b^2 - 4ac$ называется *дискриминантом* – если это положительная величина, уравнение имеет два действительных корня, если дискриминант равен нулю – уравнение имеет один действительный корень, и в случае отрицательного значения уравнение имеет два комплексных корня. Мы напишем программу, которая будет принимать

от пользователя коэффициенты *a*, *b* и *c* (коэффициенты *b* и *c* могут быть равны нулю) и затем вычислять и выводить его корень или корни.[1]

Для начала посмотрим, как работает программа, а потом перейдем к изучению программного кода.

```
quadratic.py
ax² + bx + c = 0
enter a: 2.5
enter b: 0
enter c: -7.25
2.5x² + 0.0x + -7.25 = 0 → x = 1.70293863659 or x = -1.70293863659
```

С коэффициентами 1.5, -3 и 6 программа выведет (некоторые цифры обрезаны):

```
1.5x² + -3.0x + 6.0 = 0 → x = (1+1.7320508j) or x = (1-1.7320508j)
```

Вывод программы не так хорош, как хотелось бы – например, вместо `+ -3.0x` лучше было бы выводить `-3.0x`, а коэффициенты, равные нулю, – вообще не показывать. Вы получите шанс ликвидировать эти недостатки при выполнении упражнений.

Теперь обратимся к программному коду, который начинается тремя инструкциями `import`:

```
import cmath
import math
import sys
```

Нам необходимы обе математические библиотеки для работы с числами типа `float` и `complex`, так как функции, вычисляющие квадратный корень из вещественных и комплексных чисел, отличаются. Модуль `sys` нам необходим, так как в нем определена константа `sys.float_info.epsilon`, которая потребуется нам для сравнения вещественных чисел со значением 0.

Нам также необходима функция, которая будет получать от пользователя число с плавающей точкой:

```
def get_float(msg, allow_zero):
    x = None
    while x is None:
        try:
            x = float(input(msg))
```

[1] Поскольку консоль в операционной системе Windows имеет весьма ограниченную поддержку UTF-8, а консоль в системе Mac OS X по умолчанию использует кодировку Apple Roman, существует две проблемы с символами ² и →, которые используются программой *quadratic.py*. Мы включили в примеры файл *quadratic_uni.py*, который отображает корректные символы в консоли Linux и использует их заменители (`^2` и `->`) в других системах.

```
            if not allow_zero and abs(x) < sys.float_info.epsilon:
                print("zero is not allowed")
                x = None
        except ValueError as err:
            print(err)
    return x
```

Эта функция выполняет цикл, пока пользователь не введет допустимое число с плавающей точкой (например, 0.5, -9, 21, 4.92), и допускает ввод значения 0, только если аргумент `allow_zero` имеет значение `True`.

Вслед за определением функции `get_float()` выполняется оставшаяся часть программного кода. Мы разделим его на три части и начнем со взаимодействия с пользователем:

```
print("ax\N{SUPERSCRIPT TWO} + bx + c = 0")
a = get_float("enter a: ", False)
b = get_float("enter b: ", True)
c = get_float("enter c: ", True)
```

Благодаря функции `get_float()` получить значения коэффициентов *a*, *b* и *c* оказалось очень просто. Второй аргумент функции сообщает, когда значение 0 является допустимым.

```
x1 = None
x2 = None
discriminant = (b ** 2) - (4 * a * c)
if discriminant == 0:
    x1 = -(b / (2 * a))
else:
    if discriminant > 0:
        root = math.sqrt(discriminant)
    else: # discriminant < 0
        root = cmath.sqrt(discriminant)
    x1 = (-b + root) / (2 * a)
    x2 = (-b - root) / (2 * a)
```

Программный код выглядит несколько иначе, чем формула, потому что мы начали вычисления с определения значения дискриминанта. Если дискриминант равен 0, мы знаем, что уравнение имеет единственное действительное решение и можно сразу же вычислить его. В противном случае мы вычисляем действительный или комплексный квадратный корень из дискриминанта и находим два корня уравнения.

```
equation = ("{0}x\N{SUPERSCRIPT TWO} + {1}x + {2} = 0"
            " \N{RIGHTWARDS ARROW} x = {3}").format(a, b, c, x1)
if x2 is not None:
    equation += " or x = {0}".format(x2)
print(equation)
```

Мы не использовали сколько-нибудь сложного форматирования, поскольку форматирование, используемое по умолчанию для чисел с плавающей точкой в языке Python, прекрасно подходит для этого приме-

ра, но мы использовали некоторые имена Юникода для вывода пары специальных символов.

csv2html.py

Часто бывает необходимо представить данные в формате HTML. В этом подразделе мы разработаем программу, которая читает данные из файла в простом формате CSV (Comma Separated Value – значения, разделенные запятыми) и выводит таблицу HTML, содержащую эти данные. В составе Python присутствует мощный и сложный модуль для работы с форматом CSV и похожими на него – модуль `csv`, но здесь мы будем выполнять всю обработку вручную.

В формате CSV каждая запись располагается на одной строке, а поля внутри записи отделяются друг от друга запятыми. Каждое поле может быть либо строкой, либо числом. Строки должны окружаться апострофами или кавычками, а числа не должны окружаться кавычками, если они не содержат запятые. Внутри строк допускается присутствие запятых, и они не должны интерпретироваться как разделители полей. Мы будем исходить из предположения, что первая запись в файле содержит имена полей. На выходе будет воспроизводиться таблица в формате HTML с выравниванием текста по левому краю (по умолчанию для HTML) и с выравниванием чисел по правому краю, по одной строке на запись и по одной ячейке на поле.

Программа должна вывести открывающий тег таблицы HTML, затем прочитать каждую строку данных и для каждой строки вывести соответствующую строку таблицы HTML, а в завершение вывести закрывающий тег таблицы HTML. Мы будем использовать светло-зеленый цвет фона для первой строки таблицы (где будут выводиться названия полей), а при выводе строк с данными будем чередовать белый и светло-желтый цвет фона. Кроме того, нам необходимо правильно экранировать специальные символы HTML («&», «<» и «>»), а строки немного сократить.

Ниже приводится маленький фрагмент файла с данными:

```
"COUNTRY","2000","2001",2002,2003,2004
"ANTIGUA AND BARBUDA",0,0,0,0,0
"ARGENTINA",37,35,33,36,39
"BAHAMAS, THE",1,1,1,1,1
"BAHRAIN",5,6,6,6,6
```

Предположим, что данные находятся в файле *data/co2-sample.csv* и выполнена команда `csv2html.py < data/co2-sample.csv > co2-sample.html`, тогда файл *co2-sample.html* должен содержать примерно следующее:

```
<table border='1'><tr bgcolor='lightgreen'>
<td>Country</td><td align='right'>2000</td><td align='right'>2001</td>
<td align='right'>2002</td><td align='right'>2003</td>
<td align='right'>2004</td></tr>
```

```
...
<tr bgcolor='lightyellow'><td>Argentina</td>
<td align='right'>37</td><td align='right'>35</td>
<td align='right'>33</td><td align='right'>36</td>
<td align='right'>39</td></tr>
...
</table>
```

Мы немного привели в порядок результаты работы программы и опустили некоторые строки, подставив вместо них многоточия. Мы использовали очень простую версию HTML – HTML 4 transitional, без применения таблиц стилей. На рис. 2.7 показано, как выглядит полученная таблица в веб-броузере.

Country	2000	2001	2002	2003	2004
Antigua and Barbuda	0	0	0	0	0
Argentina	37	35	33	36	39
Bahamas, The	1	1	1	1	1
Bahrain	5	6	6	6	6

***Рис. 2.7.** Таблица, произведенная программой csv2html.py, в броузере*

Теперь, когда мы увидели, как используется программа и что она делает, можно приступать к изучению программного кода. Программа начинается с импортирования модуля `sys` – с этого момента мы не будем больше показывать строки, выполняющие импортирование, если в них не импортируется нечто необычное или они не требуют обсуждения. Последняя инструкция в программе – это простой вызов функции:

```
main()
```

Хотя в языке Python не требуется явно указывать точку входа в программу, как в некоторых других языках программирования, тем не менее является распространенной практикой создание в программе на языке Python функции с именем `main()`, которая вызывается для выполнения обработки. Поскольку функция не может вызываться до того, как она будет определена, мы должны вставлять вызов `main()` только после того, как данная функция будет определена. Порядок следования функций в файле (то есть порядок, в котором они создаются) не имеет значения.

В программе *csv2html.py* первой вызываемой функцией является функция `main()`, которая в свою очередь вызывает функции `print_start()` и `print_line()`. Функция `print_line()` вызывает функции `extract_fields()` и `escape_html()`. Структура программы показана на рис. 2.8.

Когда интерпретатор Python читает файл, он начинает делать это с самого начала. Поэтому сначала будет выполнен импорт, затем будет

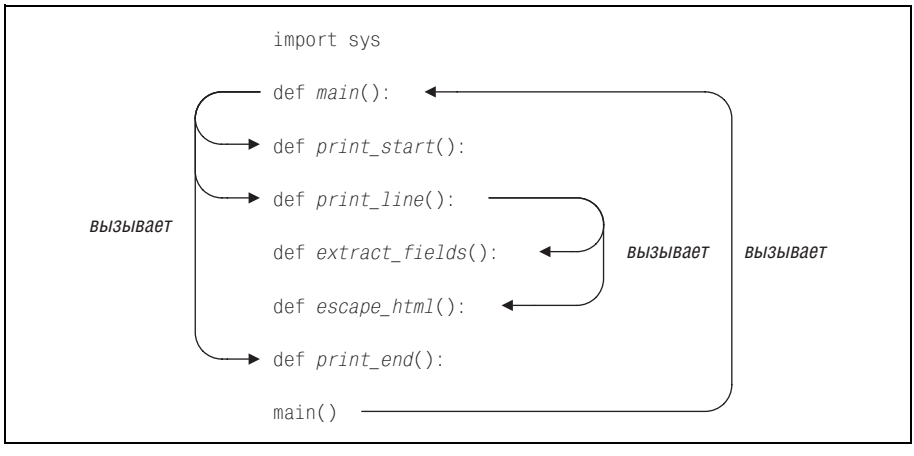

***Рис. 2.8.** Структура программы csv2html.py*

создана функция `main()`, а затем будут созданы остальные функции – в том порядке, в каком они следуют в файле. Когда интерпретатор, наконец, достигнет вызова `main()` в конце файла, все функции, которые вызываются функцией `main()` (и все функции, которые вызываются этими функциями), будут определены. Выполнение обработки, как и следовало ожидать, начинается в точке вызова функции `main()`.

Рассмотрим все функции по порядку, начиная с функции `main()`.

```
def main():
    maxwidth = 100
    print_start()
    count = 0
    while True:
        try:
            line = input()
            if count == 0:
                color = "lightgreen"
            elif count % 2:
                color = "white"
            else:
                color = "lightyellow"
            print_line(line, color, maxwidth)
            count += 1
        except EOFError:
            break
    print_end()
```

Переменная `maxwidth` используется для хранения числа символов в ячейке. Если поле больше, чем это число, часть строки отсекается и на место отброшенного текста добавляется многоточие. Программный код

функций print_start(), print_line() и print_end() будет приведен чуть ниже. Цикл while выполняет обход всех входных строк – это могут быть строки, вводимые пользователем с клавиатуры, но мы предполагаем, что данные будут перенаправлены из файла. Далее выбирается цвет фона и вызывается функция print_line(), которая выводит строку в виде строки таблицы в формате HTML.

```
def print_start():
    print("<table border='1'>")

def print_end():
    print("</table>")
```

Мы могли бы не создавать эти две функции и просто вставить соответствующие вызовы print() в функцию main(). Но мы предпочитаем выделять логику, так как это делает реализацию более гибкой, хотя в этом маленьком примере гибкость не имеет большого значения.

```
def print_line(line, color, maxwidth):
    print("<tr bgcolor='{0}'>".format(color))
    fields = extract_fields(line)
    for field in fields:
        if not field:
            print("<td></td>")
        else:
            number = field.replace(",", "")
            try:
                x = float(number)
                print("<td align='right'>{0:d}</td>".format(round(x)))
            except ValueError:
                field = field.title()
                field = field.replace(" And ", " and ")
                field = escape_html(field)
                if len(field) <= maxwidth:
                    print("<td>{0}</td>".format(field))
                else:
                    print("<td>{0:.{1}} ...</td>".format(field,
                                                         maxwidth))
    print("</tr>")
```

Мы не можем использовать метод str.split(",") для разбиения каждой строки на поля, потому что запятые могут находиться внутри строк в кавычках. Поэтому мы возложили эту обязанность на функцию extract_fields(). Получив список строк полей (в виде строк без окружающих их кавычек), мы выполняем обход списка и создаем для каждого поля ячейку таблицы.

Если поле пустое, мы выводим пустую ячейку. Если поле было заключено в кавычки, это может быть строка или число в кавычках, содержащее символы запятой, например "1,566". Учитывая такую возможность, мы создаем копию поля без запятых и пытаемся преобразовать ее в число типа float. Если преобразование удалось, мы определяем

выравнивание в ячейке по правому краю, а значение поля округляется до ближайшего целого, которое и выводится. Если преобразование не удалось, следовательно, поле содержит строку. В этом случае мы с помощью метода `str.title()` изменяем регистр символов и замещаем слово «*And*» на слово «*and*», устраняя побочный эффект действия метода `str.title()`. Затем выполняется экранирование специальных символов HTML и выводится либо поле целиком, либо первые `maxwidth` символов с добавлением многоточия. Простейшей альтернативой использованию вложенного поля замены в строке формата является получение среза строки, например:

```
print("<td>{0} ...</td>".format(field[:maxwidth]))
```

Еще одно преимущество такого подхода состоит в том, что он требует меньшего объема ввода с клавиатуры.

```
def extract_fields(line):
    fields = []
    field = ""
    quote = None
    for c in line:
        if c in "\"'":
            if quote is None: # начало строки в кавычках
                quote = c
            elif quote == c:  # конец строки в кавычках
                quote = None
            else:
                field += c    # другая кавычка внутри строки в кавычках
            continue
        if quote is None and c == ",": # end of a field
            fields.append(field)
            field = ""
        else:
            field += c        # добавить символ в поле
    if field:
            fields.append(field)  # добавить последнее поле в список
    return fields
```

Эта функция читает символы из строки один за другим и накапливает список полей, где каждое поле – это строка без окружающих ее кавычек. Функция способна обрабатывать поля, не заключенные в кавычки, и поля, заключенные в кавычки или в апострофы, корректно обрабатывая запятые и кавычки (апострофы в строках, заключенных в кавычки, и кавычки в строках, заключенных в апострофы).

```
def escape_html(text):
    text = text.replace("&", "&amp;")
    text = text.replace("<", "&lt;")
    text = text.replace(">", "&gt;")
    return text
```

Эта функция просто замещает каждый специальный символ HTML соответствующей ему сущностью языка HTML. В первую очередь, конечно, мы должны заменить символ амперсанда и угловые скобки, хотя порядок не имеет никакого значения. В стандартной библиотеке Python имеется более сложная версия этой функции – вы получите возможность использовать ее в упражнениях и еще раз встретитесь с ней в главе 7.

В заключение

Эта глава началась с демонстрации списка ключевых слов языка Python и описания правил, применяемых к идентификаторам в языке Python. Благодаря поддержке Юникода идентификаторы языка Python не ограничены поднабором символов такого небольшого множества, как ASCII или Latin-1.

Также был описан тип данных `int`, который отличается от аналогичных типов во многих других языках программирования тем, что не имеет ограничений на размер. Размер целых чисел в языке Python ограничивается лишь объемом машинной памяти, и интерпретатор вполне в состоянии работать с числами, состоящими из сотен цифр. Все основные типы данных в языке Python относятся к категории неизменяемых, но эта их особенность практически незаметна – за счет того, что комбинированные операторы присваивания (`+=`, `*=`, `-=`, `/=` и другие) позволяют использовать достаточно естественный синтаксис, хотя при этом интерпретатор Python создает новые объекты с результатами и выполняет повторную привязку к ним наших переменных. Литералы целых чисел обычно записываются в виде десятичных чисел, но также существует возможность записывать двоичные литералы, используя префикс `0b`, восьмеричные литералы, используя префикс `0o`, и шестнадцатеричные литералы, используя префикс `0x`.

Когда деление двух целых чисел выполняется с помощью оператора /, результатом всегда будет число типа `float`. Это отличает Python от многих других языков программирования, но позволяет избежать некоторых трудноуловимых ошибок, которые могут возникнуть из-за усечения дробной части в результате. (Если необходимо выполнить целочисленное деление, следует использовать оператор //.)

В языке Python имеется тип данных `bool`, который может иметь одно из двух значений – `True` или `False`. В языке Python имеется три логических оператора: `and`, `or` и `not`, два из которых (`and` и `or`) опираются на логику сокращенных вычислений.

Имеется три разновидности чисел с плавающей точкой: `float`, `complex` и `decimal.Decimal`. Наиболее часто используется тип `float` – он представляет числа с плавающей точкой двойной точности, чьи точные характеристики зависят от библиотек C, C# или Java, на основе которых была выполнена компиляция Python. Комплексные числа представле-

ны парой чисел типа `float`, одно из которых хранит действительную часть комплексного числа, а второе – мнимую. Тип `decimal.Decimal` реализован модулем `decimal`. По умолчанию эти числа имеют точность представления 28 десятичных знаков, однако точность может быть увеличена или уменьшена в зависимости от потребностей.

Все три типа чисел с плавающей точкой могут использоваться в комбинации с типичными арифметическими операторами и функциями. В дополнение к этому модуль `math` предоставляет разнообразные тригонометрические, гиперболические и логарифмические функции, которые могут использоваться с числами типа `float`, а модуль `cmath` предоставляет аналогичное множество функций для работы с числами типа `complex`.

Большая часть главы посвящена строкам. Литералы строк в языке Python могут создаваться с помощью апострофов или кавычек, а если возникает необходимость включить в строку символы перевода строки или кавычки, можно использовать тройные кавычки. Для вставки специальных символов могут использоваться различные экранированные последовательности, такие как табуляция (`\t`) и перевод строки (`\n`), и символы Юникода, как с использованием шестнадцатеричных экранированных последовательностей, так и с использованием названий символов Юникода. Несмотря на то, что строки поддерживают те же самые операторы сравнения, что и другие типы данных в языке Python, мы отметили, что сортировка строк, содержащих неанглийские символы, может вызывать сложности.

Поскольку строки являются последовательностями, к ним может применяться оператор получения среза (`[]`), имеющий простой, но мощный синтаксис. Строки могут также объединяться с помощью оператора `+` и дублироваться с помощью оператора `*`; кроме того, можно использовать комбинированные операторы присваивания (`+=` и `*=`), хотя для конкатенации строк предпочтительнее использовать метод `str.join()`. Строки имеют множество других методов, включая методы проверки их содержимого (такие как `str.isspace()` и `str.isalpha()`), методы изменения регистра символов (такие как `str.lower()` и `str.title()`), методы поиска (такие как `str.find()` и `str.index()`) и многие другие.

Поддержка строк в языке Python действительно находится на очень высоком уровне, позволяя нам легко отыскивать, извлекать или сравнивать как целые строки, так и их части, замещать символы или подстроки, разбивать строки на списки подстрок и объединять списки строк в единую строку.

Пожалуй, самым универсальным строковым методом является метод `str.format()`. Этот метод используется для создания строк путем замещения полей значениями переменных и посредством задания спецификаторов формата, точно определяющих характеристики каждого поля, замещаемого некоторым значением. Синтаксис имен замещаемых полей позволяет организовать доступ к позиционным или именo-

ванным аргументам метода и использовать индексы, ключи или имена атрибутов для доступа к элементам или атрибутам аргументов. Спецификаторы формата позволяют определять символ-заполнитель, направление выравнивания и минимальную ширину поля вывода. Кроме того, при форматировании чисел мы можем определять, как должен выводиться знак числа, а для чисел с плавающей точкой указывать число знаков после десятичной точки и выводить их в стандартном или экспоненциальном представлении.

Мы также обсудили сложную проблему кодировок символов. По умолчанию для файлов *.py* используется кодировка UTF-8, благодаря чему мы имеем возможность записывать комментарии, идентификаторы и данные на любом языке человеческого общения. С помощью метода `str.encode()` мы можем преобразовать строку в последовательность байтов, используя определенную кодировку, а с помощью метода `bytes.decode()` выполнить обратное преобразование последовательности байтов в строку, используя определенную кодировку. Широкое разнообразие кодировок, находящихся в использовании, может доставлять массу неудобств, но кодировка UTF-8 быстро превращается в фактический стандарт для простых текстовых файлов (и уже используется по умолчанию для XML-файлов), поэтому данная проблема должна потерять свою остроту в ближайшие годы.

В дополнение к типам данных, рассматривавшимся в этой главе, Python предоставляет еще два встроенных типа данных – `bytes` и `bytearray`, оба они будут рассматриваться в главе 7. В языке Python имеется также несколько типов коллекций, часть которых является встроенными типами, а часть реализована в стандартной библиотеке. Наиболее важные типы коллекций языка Python будут рассматриваться в следующей главе.

Упражнения

1. Измените программу *print_unicode.py* так, чтобы пользователь мог вводить в командной строке несколько разных слов и получать только те строки из таблицы символов Юникода, в которых содержатся все слова, указанные пользователем. Это означает, что мы сможем вводить такие команды:

   ```
   print_unicode_ans.py greek symbol
   ```

   Один из способов достижения поставленной цели состоит в том, чтобы заменить переменную `word` (которая может хранить `0`, `None` или строку) списком `words`. Не забудьте изменить информацию о порядке использования. В результате изменений не более десяти строк программного кода добавится и не более десяти строк изменится. Решение находится в файле *print_unicode_ans.py* (Пользователи Windows и кроссплатформенной версии программы должны

модифицировать файл *print_inicode_uni.py*, а решение находится в файле *print_inicode_uni_ans.py*.)

2. Измените программу *quadratic.py* так, чтобы она не выводила коэффициенты со значением 0.0, а отрицательные коэффициенты выводились бы как *-n*, а не *+ -n*. Для этого придется заменить последние пять строк программы примерно пятнадцатью строками. Решение находится в файле *quadratic_ans.py*. (Пользователи Windows и кроссплатформенной версии программы должны модифицировать файл *quadratic_uni.py*, а решение находится в файле *quadratic_uni_ans.py*.)
3. Удалите функцию `escape_html()` из программы *cvs2html.py* и используйте вместо нее функцию `xml.sax.saxutils.escape()` из модуля `xml.sax.saxutils`. Для этого потребуется добавить одну новую строку (с инструкцией `import`), удалить пять строк (с ненужной функцией) и изменить одну строку (задействовать функцию `xml.sax.saxutils.escape()` вместо `escape_html()`). Решение приводится в файле *csv2html1_ans.py*.
4. Измените программу *cvs2html.py* еще раз и добавьте в нее новую функцию с именем `process_options()`. Эта функция должна вызываться из функции `main()` и возвращать кортеж с двумя значениями: `maxwidth` (типа `int`) и `format` (типа `str`). При вызове функция `process_options()` должна устанавливать `maxwidth` в значение по умолчанию 100, а строку `format` – в значение по умолчанию `"%.0f"`, которое будет использоваться как спецификатор формата при выводе чисел.

   Если пользователь вводит в командной строке «-h» или «--help», должно выводиться сообщение о порядке использования и возвращаться кортеж `(None, None)`. (В этом случае функция `main()` ничего делать не должна.) В противном случае функция должна прочитать аргументы командной строки и выполнить соответствующие присваивания. Например, устанавливать значение переменной `maxwidth`, если задан аргумент «maxwidth=*n*», и точно так же устанавливать значение переменной `format`, если задан аргумент «format=*s*». Ниже приводится сеанс работы с программой, когда пользователь затребовал инструкцию о порядке работы:

```
csv2html2_ans.py -h
usage:
csv2html.py [maxwidth=int] [format=str] < infile.csv > outfile.html

maxwidth is an optional integer; if specified, it sets the maximum
number of characters that can be output for string fields,
otherwise a default of 100 characters is used.
(maxwidth – необязательное целое число. Если задано, определяет
максимальное число символов для строковых полей. В противном случае
используется значение по умолчанию 100.)

format is the format to use for numbers; if not specified it
defaults to "%.0f".
```

```
(format – формат вывода чисел. Если не задан, по умолчанию используется
формат ".0f".)
```

А ниже приводится пример командной строки, в которой установлены оба аргумента:

```
csv2html2_ans.py maxwidth=20 format=0.2f < mydata.csv > mydata.html
```

Не забудьте изменить функцию `print_line()` так, чтобы она использовала переменную `format` при выводе чисел – для этого вам придется передавать функции дополнительный аргумент, добавить одну строку и изменить еще одну строку. И это немного затронет функцию `main()`. Функция `process_options()` должна содержать порядка двадцати пяти строк (включая девять строк с текстом сообщения о порядке использования). Это упражнение может оказаться сложным для неопытных программистов.

В состав примеров входят два файла с тестовыми данными: *data/co2-sample.csv* и *data/co2-from-fossilfuels.csv*. Решение приводится в файле *csv2html2_ans.py*. В главе 5 мы увидим, как для обработки аргументов командной строки можно использовать модуль `optparse`.

- Последовательности
- Множества
- Отображения
- Обход в цикле и копирование коллекций

3

Типы коллекций

В предыдущей главе мы познакомились с наиболее важными фундаментальными типами данных языка Python. В этой главе мы расширим свои возможности, узнав, как объединять элементы данных вместе, используя типы коллекций языка Python. В этой главе мы рассмотрим кортежи и списки, а также познакомимся с новыми типами коллекций, включая словари и множества, и детально изучим их.[1]

В дополнение к коллекциям мы также узнаем, как создавать элементы данных, вмещающие в себя другие элементы данных (подобно структурам в языках C и C++ и записям в языке Pascal). Такие элементы в случае необходимости могут интерпретироваться как единое целое, и при этом сохраняется возможность прямого доступа к отдельным элементам, хранящимся в них. Естественно, ничто не мешает вставлять такие агрегатные элементы в коллекции, как любые другие элементы.

Наличие коллекций элементов данных существенно упрощает выполнение операций, которые должны применяться к элементам, а также упрощает обработку коллекций элементов при чтении их из файлов. В этой главе мы рассмотрим основные приемы работы с файлами лишь в том объеме, который нам потребуется, отложив описание основных подробностей (включая обработку ошибок) до главы 7.

После знакомства с отдельными типами коллекций мы посмотрим, как можно организовать обход коллекций в цикле, поскольку в языке Python для итераций через любые коллекции используются одни и те же синтаксические конструкции. Кроме этого, мы исследуем проблемы и приемы копирования коллекций.

---

[1] Определение того, что является последовательностью, множеством или отображением, в этой главе дается не с формальной, а с практической точки зрения. Более формальные определения даются в главе 8.

Последовательности

Последовательности – это один из типов данных, поддерживающих оператор проверки на вхождение (in), функцию определения размера (`len()`), оператор извлечения срезов (`[]`) и возможность выполнения итераций. В языке Python имеется пять встроенных типов последовательностей: `bytearray`, `bytes`, `list`, `str` и `tuple` – первые два будут описаны отдельно, в главе 7. Ряд дополнительных типов последовательностей реализован в стандартной библиотеке; наиболее примечательным из них является тип `collections.namedtuple`. При выполнении итераций все эти последовательности гарантируют строго определенный порядок следования элементов.

Строки, стр. 84

Строки мы уже рассматривали в предыдущей главе, а в этом разделе познакомимся с кортежами, именованными кортежами и списками.

Кортежи

Извлечение срезов из строк, стр. 89

Кортеж – это упорядоченная последовательность из нуля или более ссылок на объекты. Кортежи поддерживают тот же синтаксис получения срезов, что и строки. Это упрощает извлечение элементов из кортежа. Подобно строкам, кортежи относятся к категории неизменяемых объектов, поэтому мы не можем замещать или удалять какие-либо их элементы. Если нам необходимо иметь возможность изменять упорядоченную последовательность, то вместо кортежей можно просто использовать списки или, если в программе уже используется кортеж, который нежелательно модифицировать, можно преобразовать кортеж в список с помощью функции преобразования `list()` и затем изменять полученный список.

Поверхностное и глубокое копирование, стр. 173

Тип данных `tuple` может вызываться как функция `tuple()` – без аргументов она возвращает пустой кортеж, с аргументом типа `tuple` возвращает поверхностную копию аргумента; в случае, если аргумент имеет другой тип, выполняется попытка преобразовать его в объект типа `tuple`. Эта функция принимает не более одного аргумента. Кроме того, кортежи могут создаваться без использования функции `tuple()`. Пустой кортеж создается с помощью пары пустых круглых скобок `()`, а кортеж, состоящий из одного или более элементов, может быть создан с помощью запятых. Иногда кортежи приходится заключать в круглые скобки, чтобы избежать синтаксической неоднозначности. Например, чтобы передать кортеж `1, 2, 3` в функцию, необходимо использовать такую форму записи: `function((1, 2, 3))`.

Рис. 3.1. *Позиции элементов в кортеже*

На рис. 3.1 показан кортеж `t = "venus", -28, "green", "21", 19.74` и индексы элементов внутри кортежа. Строки индексируются точно так же, но, если в строках каждой позиции соответствует единственный символ, то в кортежах каждой позиции соответствует единственная ссылка на объект.

Кортежи предоставляют всего два метода: `t.count(x)`, который возвращает количество объектов `x` в кортеже `t`, и `t.index(x)`, который возвращает индекс самого первого (слева) вхождения объекта `x` в кортеж `t` или возбуждает исключение `ValueError`, если объект `x` отсутствует в кортеже. (Эти методы имеются также и у списков.)

Кроме того, кортежи могут использоваться с оператором `+` (конкатенации), `*` (дублирования) и `[]` (получения среза), а операторы `in` и `not in` могут применяться для проверки на вхождение. Можно использовать также комбинированные операторы присваивания `+=` и `*=`. Несмотря на то, что кортежи являются неизменяемыми объектами, при выполнении этих операторов интерпретатор Python создает за кулисами новый кортеж с результатом операции и присваивает ссылку на него объекту, расположенному слева от оператора, то есть используется тот же самый прием, что и со строками. Кортежи могут сравниваться с помощью стандартных операторов сравнения (`<`, `<=`, `==`, `!=`, `>=`, `>`), при этом сравнивание производится поэлементно (и рекурсивно, при наличии вложенных элементов, таких как кортежи в кортежах).

Рассмотрим несколько примеров получения срезов, начав с извлечения единственного элемента и группы элементов:

```
>>> hair = "black", "brown", "blonde", "red"
>>> hair[2]
'blonde'
>>> hair[-3:] # то же, что и hair[1:]
('brown', 'blonde', 'red')
```

Эта операция выполняется точно так же, как и в случае со строками, списками или любыми другими последовательностями.

```
>>> hair[:2], "gray", hair[2:]
(('black', 'brown'), 'gray', ('blonde', 'red'))
```

Здесь мы попытались создать новый кортеж из 5 элементов, но в результате получили кортеж с тремя элементами, содержащий два двух-

элементных кортежа. Это произошло потому, что мы применили оператор запятой к трем элементам (кортеж, строка и кортеж). Чтобы получить единый кортеж со всеми этими элементами, необходимо выполнить конкатенацию кортежей:

```
>>> hair[:2] + ("gray",) + hair[2:]
('black', 'brown', 'gray', 'blonde', 'red')
```

Чтобы создать кортеж из одного элемента, необходимо поставить запятую, но если запятую просто добавить, будет получено исключение `TypeError` (так как интерпретатор будет думать, что выполняется конкатенация строки и кортежа), поэтому необходимо использовать запятую *и* круглые скобки.

В этой книге (начиная с этого момента) мы будем использовать определенный стиль записи кортежей. Когда кортеж будет стоять слева от двухместного оператора или справа от одноместного, мы будем опускать круглые скобки. Во всех остальных случаях будут использоваться круглые скобки. Ниже приводятся несколько примеров:

```
a, b = (1, 2)                          # слева от двухместного оператора

del a, b                               # справа от одноместного оператора

def f(x):
    return x, x ** 2                   # справа от одноместного оператора

for x, y in ((1, 1), (2, 4), (3, 9)): # слева от двухместного оператора
    print(x, y)
```

Совершенно необязательно следовать этому стилю записи – некоторые программисты предпочитают всегда использовать круглые скобки, что соответствует репрезентативной форме представления кортежей, однако другие используют скобки, только когда это строго необходимо.

```
>>> eyes = ("brown", "hazel", "amber", "green", "blue", "gray")
>>> colors = (hair, eyes)
>>> colors[1][3:-1]
('green', 'blue')
```

В следующем примере мы вложили друг в друга два кортежа. Коллекции допускают возможность вложения с любой глубиной вложенности. Оператор извлечения срезов `[]` может применяться для доступа к вложенным коллекциям столько раз, сколько это будет необходимо. Например:

```
>>> things = (1, -7.5, ("pea", (5, "Xyz"), "queue"))
>>> things[2][1][1][2]
'z'
```

Рассмотрим этот пример по частям, начиная с выражения `things[2]`, которое дает нам третий элемент кортежа (не забывайте, что первый элемент имеет индекс 0), который сам является кортежем `("pea", (5, "Xyz"), "queue")`. Выражение `things[2][1]` дает нам второй элемент кор-

тежа `things[2]`, который тоже является кортежем `(5, "Xyz")`. А выражение `things[2][1][1]` дает нам второй элемент этого кортежа, который представляет строку `"Xyz"`. Наконец, выражение `things[2][1][1][2]` дает нам третий элемент (символ) строки, то есть символ `"z"`.

Кортежи могут хранить элементы любых типов, включая другие коллекции, такие как кортежи и списки, так как на самом деле кортежи хранят ссылки на объекты. Использование сложных, вложенных структур данных, таких, как показано ниже, легко может создавать путаницу. Одно из решений этой проблемы состоит в том, чтобы давать значениям индексов осмысленные имена. Например:

```
>>> MANUFACTURER, MODEL, SEATING = (0, 1, 2)
>>> MINIMUM, MAXIMUM = (0, 1)
>>> aircraft = ("Airbus", "A320-200", (100, 220))
>>> aircraft[SEATING][MAXIMUM]
220
```

Конечно, в таком виде программный код выглядит более осмысленным, чем простое выражение `aircraft[2][1]`, но при этом приходится создавать большое число переменных, да и выглядит он несколько уродливо. В следующем подразделе мы познакомимся с более привлекательной альтернативой.

В первых двух строках вышеприведенного фрагмента мы выполнили присваивание кортежам. Когда справа от оператора присваивания указывается последовательность (в данном случае – это кортежи), а слева указан кортеж, мы говорим, что последовательность справа *распаковывается*). Операция распаковывания последовательностей может использоваться для организации обмена значений между переменными, например:

```
a, b = (b, a)
```

Строго говоря, круглые скобки справа не являются обязательными, но, как уже отмечалось выше, в этой книге мы используем стиль записи, когда скобки опускаются только в левом операнде двухместного оператора и в правом операнде одноместного оператора и используются во всех остальных случаях.

Мы уже сталкивались с примерами распаковывания последовательностей в контексте оператора цикла `for ... in`. Следующий пример приводится только в качестве напоминания:

```
for x, y in ((-3, 4), (5, 12), (28, -45)):
    print(math.hypot(x, y))
```

Здесь выполняется обход кортежа, состоящего из двухэлементных кортежей, каждый из которых распаковывается в переменные `x` и `y`.

Именованные кортежи

Именованные кортежи ведут себя точно так же, как и обычные кортежи, и не уступают им в производительности. Отличаются они возможностью ссылаться на элементы кортежа не только по числовому индексу, но и по имени, что в свою очередь позволяет создавать сложные агрегаты из элементов данных.

В модуле `collections` имеется функция `namedtuple()`. Эта функция используется для создания собственных типов кортежей. Например:

```
Sale = collections.namedtuple("Sale",
            "productid customerid date quantity price")
```

Первый аргумент функции `collections.namedtuple()` – это имя создаваемого кортежа. Второй аргумент – это строка имен, разделенных пробелами, для каждого элемента, который будет присутствовать в этом кортеже. Первый аргумент и имена во втором аргументе должны быть допустимыми идентификаторами языка Python. Функция возвращает класс (тип данных), который может использоваться для создания именованных кортежей. Так, в примере выше мы можем интерпретировать имя `Sale` как имя любого другого класса (такого как `tuple`) в языке Python и создавать объекты типа `Sale`.[1] Например:

```
sales = []
sales.append(Sale(432, 921, "2008-09-14", 3, 7.99))
sales.append(Sale(419, 874, "2008-09-15", 1, 18.49))
```

В этом примере мы создали список из двух элементов типа `Sale`, то есть из двух именованных кортежей. Мы можем обращаться к элементам таких кортежей по их индексам – например, обратиться к элементу `price` в первом элементе списка `sales` можно с помощью выражения `sales[0][-1]` (вернет значение 7.99) – или по именам, которые делают программный код более удобочитаемым:

```
total = 0
for sale in sales:
    total += sale.quantity * sale.price
print("Total ${0:.2f}".format(total)) # выведет: Total $42.46
```

Очень часто простоту и удобство, которые предоставляют именованные кортежи, можно обратить на пользу делу. Например, ниже приводится версия примера «aircraft» из предыдущего подраздела (стр. 133), имеющая более аккуратный вид:

```
>>> Aircraft = collections.namedtuple("Aircraft",
...                                   "manufacturer model seating")
>>> Seating = collections.namedtuple("Seating", "minimum maximum")
```

[1] Примечание для программистов, использующих объектно-ориентированный стиль: каждый класс, созданный таким способом, будет являться подклассом класса `tuple`.

```
>>> aircraft = Aircraft("Airbus", "A320-200", Seating(100, 220))
>>> aircraft.seating.maximum
220
```

Уже видно, что именованные кортежи могут быть очень удобны; кроме того, в главе 6 мы перейдем к изучению объектно-ориентированного программирования, где выйдем за пределы простых именованных кортежей и узнаем, как создавать свои собственные типы данных, которые могут не только хранить элементы данных, но и иметь собственные методы.

Списки

Извлечение срезов из строк, стр. 89

Список – это упорядоченная последовательность из нуля или более ссылок на объекты. Списки поддерживают тот же синтаксис получения срезов, что и строки с кортежами. Это упрощает извлечение элементов из списка. В отличие от строк и кортежей списки относятся к категории изменяемых объектов, поэтому мы можем замещать или удалять любые их элементы. Кроме того, существует возможность вставлять, замещать и удалять целые срезы списков.

Поверхностное и глубокое копирование, стр. 173

Тип данных `list` может вызываться как функция `list()` – без аргументов она возвращает пустой список, с аргументом типа `list` возвращает поверхностную копию аргумента; в случае, если аргумент имеет другой тип, выполняется попытка преобразовать его в объект типа `list`. Эта функция принимает не более одного аргумента. Кроме того, списки могут создаваться без использования функции `list()`. Пустой список создается с помощью пары пустых квадратных скобок `[]`, а список, состоящий из одного или более элементов, может быть создан с помощью последовательности элементов, разделенных запятыми, заключенной в квадратные скобки. Другой способ создания списков заключается в использовании генераторов списков – эта тема будет рассматриваться ниже в этом подразделе.

Генераторы списков, стр. 142

Поскольку все элементы списка в действительности являются ссылками на объекты, списки, как и кортежи, могут хранить элементы любых типов данных, включая коллекции, такие как списки и кортежи. Списки могут сравниваться с помощью стандартных операторов сравнения (`<`, `<=`, `==`, `!=`, `>=`, `>`), при этом сравнивание производится поэлементно (и рекурсивно, при наличии вложенных элементов, таких как списки или кортежи в списках).

В результате выполнения операции присваивания `L = [-17.5, "kilo", 49, "V", ["ram", 5, "echo"], 7]` мы получим список, как показано на рис. 3.2.

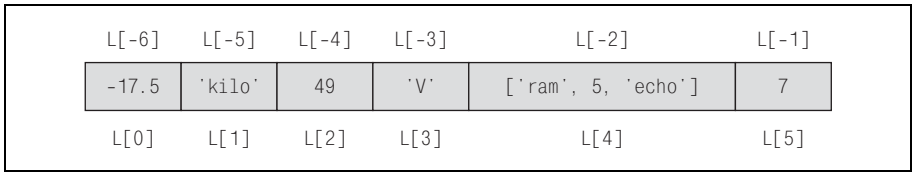

***Рис. 3.2.** Позиции элементов в списке*

К спискам, таким как `L`, мы можем применять оператор извлечения среза, повторяя его столько раз, сколько потребуется для доступа к элементам в списке, как показано ниже:

```
L[0] == L[-6] == -17.5
L[1] == L[-5] == 'kilo'
L[1][0] == L[-5][0] == 'k'
L[4][2] == L[4][-1] == L[-2][2] == L[-2][-1] == 'echo'
L[4][2][1] == L[4][2][-3] == L[-2][-1][1] == L[-2][-1][-3] == 'c'
```

Списки, как и кортежи, могут вкладываться друг в друга; допускают выполнение итераций по их элементам и извлечение срезов. Все примеры с кортежами, которые приводились в предыдущем подразделе, будут работать точно так же, если вместо кортежей в них будут использованы списки. Списки поддерживают операторы проверки на вхождение `in` и `not in`, оператор конкатенации `+`, оператор расширения `+=` (то есть добавляет операнд справа в конец списка) и операторы дублирования `*` и `*=`. Списки могут также использоваться в качестве аргументов функции `len()` и в инструкции `del`, которая будет рассматриваться в этом подразделе и которая описывается во врезке «Удаление элементов с помощью инструкции del» на стр. 139. Кроме того, списки предоставляют методы, перечисленные в табл. 3.1.

Таблица 3.1. Методы списков

Синтаксис	Описание
`L.append (x)`	Добавляет элемент `x` в конец списка `L`
`L.count(x)`	Возвращает число вхождений элемента `x` в список `L`
`L.extend(m)` `L += m`	Добавляет в конец списка `L` все элементы итерируемого объекта `m`; оператор `+=` делает то же самое

`L.index(x,`
`  start,`
`  end)`
	Возвращает индекс самого первого (слева) вхождения элемента `x` в список `L` (или в срез `start:end` списка `L`), в противном случае возбуждает исключение `ValueError`

`L.insert(i, x)`	Вставляет элемент `x` в список `L` в позицию `int i`
`L.pop()`	Удаляет самый последний элемент из списка `L` и возвращает его в качестве результата
`L.pop(i)`	Удаляет из списка `L` элемент с индексом `int i` и возвращает его в качестве результата

Синтаксис	Описание
`L.remove(x)`	Удаляет самый первый (слева) найденный элемент `x` из списка `L` или возбуждает исключение `ValueError`, если элемент `x` не будет найден
`L.reverse()`	Переставляет в памяти элементы списка в обратном порядке
`L.sort(...)`	Сортирует список в памяти. Этот метод принимает те же необязательные аргументы *key* и *reverse*, что и встроенная функция `sorted()`

Функция `sorted()`, стр. 164, 170

Несмотря на то, что для доступа к элементам списка можно использовать оператор извлечения среза, тем не менее в некоторых ситуациях бывает необходимо одновременно извлечь две или более частей списка. Сделать это можно с помощью операции распаковывания последовательности. Любой итерируемый объект (списки, кортежи и другие) может быть распакован с помощью оператора распаковывания «звездочка» (`*`). Когда слева от оператора присваивания указывается две или более переменных, одна из которых предваряется символом `*`, каждой переменной присваивается по одному элементу списка, а переменной со звездочкой присваивается оставшаяся часть списка. Ниже приводится несколько примеров выполнения распаковывания списков:

```
>>> first, *rest = [9, 2, -4, 8, 7]
>>> first, rest
(9, [2, -4, 8, 7])
>>> first, *mid, last = "Charles Philip Arthur George Windsor".split()
>>> first, mid, last
('Charles', ['Philip', 'Arthur', 'George'], 'Windsor')
>>> *directories, executable = "/usr/local/bin/gvim".split("/")
>>> directories, executable
(['', 'usr', 'local', 'bin'], 'gvim')
```

Когда используется оператор распаковывания последовательности, как в данном примере, выражение `*rest` и подобные ему называются *выражениями со звездочкой*.

В языке Python имеется также похожее понятие *аргументов со звездочкой*. Например, допустим, что имеется следующая функция, принимающая три аргумента:

```
def product(a, b, c):
    return a * b * c # здесь * - это оператор умножения
```

тогда мы можем вызывать эту функцию с тремя аргументами или использовать аргументы со звездочкой:

```
>>> product(2, 3, 5)
30
>>> L = [2, 3, 5]
>>> product(*L)
```

```
30
>>> product(2, *L[1:])
30
```

В первом примере функция вызывается, как обычно, с тремя аргументами. Во втором вызове использован аргумент со звездочкой; в этом случае список из трех элементов распаковывается оператором `*`, так что функция получает столько аргументов, сколько ей требуется. Того же эффекта можно было бы добиться при использовании кортежа с тремя элементами. В третьем вызове функции первый аргумент передается традиционным способом, а другие два – посредством применения операции распаковывания двухэлементного среза списка `L`. Функции и передача аргументов полностью будут описываться в главе 4.

В программах всегда однозначно известно, является оператор `*` оператором умножения или оператором распаковывания последовательности. Когда он появляется слева от оператора присваивания – это оператор распаковывания; когда он появляется где-то в другом месте (например, в вызове функции) – это оператор распаковывания, если он используется в одноместном операторе, и оператор умножения, если он используется в двухместном операторе.

Мы уже знаем, что имеется возможность выполнять итерации по элементам списка с помощью конструкции `for` *`item`* `in` *`L`*`:`. Если в цикле потребуется изменять элементы списка, то можно использовать следующий прием:

```
for i in range(len(L)):
    L[i] = process(L[i])
```

Функция range(), стр. 167

Встроенная функция `range()` возвращает целочисленный итератор. С одним целочисленным аргументом, n, итератор `range()` возвращает последовательность чисел $0, 1, \ldots, n-1$.

Этот прием можно использовать для увеличения всех элементов в списке целых чисел. Например:

```
for i in range(len(numbers)):
    numbers[i] += 1
```

Поскольку списки поддерживают возможность извлечения срезов, в определенных случаях один и тот же эффект может быть достигнут как с помощью оператора извлечения среза, так и с помощью одного из методов списков. Например, предположим, что имеется список `woods = ["Cedar", "Yew", "Fir"]`; дополнить такой список можно двумя способами:

```
woods += ["Kauri", "Larch"]        |        woods.extend(["Kauri", "Larch"])
```

В обоих случаях в результате будет получен список `['Cedar', 'Yew', 'Fir', 'Kauri', 'Larch']`.

Удаление элементов с помощью инструкции del

Несмотря на то, что название инструкции `del` вызывает ассоциации со словом *delete* (*удалить*), она не обязательно удаляет какие-либо данные. Когда инструкция `del` применяется к элементу данных, который не является коллекцией, она разрывает связь между ссылкой на объект и самим элементом данных и удаляет *ссылку на объект*. Например:

```
>>> x = 8143   # создается ссылка на объект 'x'
                 и целое число 8143
>>> x
8143
>>> del x      # удаляется ссылка на объект 'x',
                 число готово к утилизации
>>> x
Traceback (most recent call last):
...
NameError: name 'x' is not defined
(NameError: имя 'x' не определено)
```

Когда удаляется ссылка на объект, если не осталось других ссылок, указывающих на этот объект, то интерпретатор помечает элемент данных, на который указывала ссылка, как готовый к утилизации. Невозможно предсказать, когда произойдет утилизация и произойдет ли она вообще (это зависит от реализации Python), поэтому, когда необходимо явно освободить память, делать это придется вручную. Язык Python предоставляет два решения проблемы неопределенности. Одно из них состоит в использовании конструкции `try ... finally`, которая гарантирует освобождение памяти, а другое заключается в использовании инструкции `with`, с которой мы познакомимся в главе 8.

Когда инструкция `del` применяется к коллекциям, таким как кортежи или списки, удаляется только ссылка на эти коллекции. Коллекция и ее элементы (а также элементы, которые сами являются коллекциями для своих элементов, рекурсивно) помечаются как готовые к утилизации, если не осталось других ссылок, указывающих на эти коллекции.

Для изменяемых коллекций, таких как списки, инструкция `del` может применяться к отдельным элементам или срезам – в любом из этих случаев используется оператор извлечения среза `[]`. Если для удаления предназначен элемент или элементы коллекции и в программе не осталось ссылок, указывающих на эти элементы, они помечаются, как готовые к утилизации.

Отдельные элементы можно добавлять в конец списка с помощью метода `list.append()`. Элементы могут вставляться в любую позицию в списке с помощью метода `list.insert()` или посредством обращения к срезу с нулевой длиной. Например, допустим, что имеется список `woods = ["Cedar", "Yew", "Fir", "Spruce"]`; тогда вставить новый элемент в позицию с индексом 2 (то есть сделать этот элемент третьим элементом списка) можно одним из двух способов:

```
woods[2:2] = ["Pine"]          |       woods.insert(2, "Pine")
```

В обоих случаях в результате будет получен список `['Cedar', 'Yew', 'Pine', 'Fir', 'Spruce']`.

Отдельные элементы списка можно изменять, выполняя присваивание определенной позиции в списке, например, `woods = 'Redwood'`. Путем присваивания итерируемых объектов можно изменять целые срезы в списке, например, `woods[1:3] = ["Spruce", "Sugi", "Rimu"]`. Срез и итерируемый объект не обязательно должны иметь одинаковую длину. В любом случае элементы, попавшие в срез, будут удалены, а на их место будут вставлены элементы итерируемого объекта. Если длина итерируемого объекта короче замещаемого среза, список уменьшится, а если длина итерируемого объекта больше замещаемого среза, то список увеличится.

Чтобы прояснить, что именно происходит в результате присваивания итерируемого объекта срезу списка, рассмотрим еще один пример. Представим, что имеется список `L = ["A", "B", "C", "D", "E", "F"]` и что выполняется присваивание итерируемого объекта (в данном случае – списка) срезу: `L[2:5] = ["X", "Y"]`. В первую очередь производится удаление элементов среза, то есть за кулисами список принимает вид `['A', 'B', 'F']`. А затем все элементы итерируемого объекта вставляются в позицию первого элемента среза, и в результате получается список `['A', 'B', 'X', 'Y', 'F']`.

Существует еще ряд других способов удаления элементов списка. Чтобы удалить самый правый элемент списка, можно воспользоваться методом `list.pop()` без аргументов – удаленный элемент возвращается в качестве результата. Аналогично можно использовать метод `list.pop()` с целочисленным значением индекса – для удаления (и возвращения) элемента с определенным индексом. Еще один способ удаления элемента заключается в использовании метода `list.remove()`, которому передается удаляемый элемент. Также для удаления отдельных элементов или целых срезов можно использовать инструкцию `del` – например, `del woods[4]`. Кроме того, срезы могут удаляться путем присваивания пустого списка, так следующие два фрагмента являются эквивалентными:

```
woods[2:4] = []                |       del woods[2:4]
```

В левом фрагменте выполняется присваивание итерируемого объекта (пустого списка) срезу, то есть здесь сначала удаляются элементы сре-

за, а так как итерируемый объект, предназначенный для вставки, пуст, вставка не производится.

Извлечение срезов из строк, стр. 89

Когда мы впервые обсуждали операцию извлечения разреженного среза, мы делали это в контексте строк, где извлечение разреженных срезов выглядит не очень интересно. Но в случае списков операция извлечения разреженного среза позволяет получить доступ к каждому *n*-му элементу, что может оказаться очень удобно. Например, предположим, что имеется список `x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` и необходимо все элементы с нечетными индексами (то есть `x[1]`, `x[3]` и т. д.) установить в значение 0. Получить доступ к каждому второму элементу можно, используя оператор среза с шагом, например, `x[::2]`. Но такой оператор даст доступ к элементам с индексами 0, 2, 4 и т. д. Мы можем исправить положение, указав начальный индекс, использовав выражение `x[1::2]`, которое возвращает срез, содержащий нужные нам элементы. Чтобы установить все элементы среза в значение 0, нам необходим список нулей, и этот список должен содержать то же самое число нулей, сколько элементов содержится в срезе.

Полное решение задачи выглядит так: `x[1::2] = [0] * len(x[1::2])`. Теперь список x имеет следующий вид `[1, 0, 3, 0, 5, 0, 7, 0, 9, 0]`. Мы воспользовались оператором дублирования `*` для создания списка, содержащего необходимое число нулей, основываясь на длине (то есть количестве элементов) среза. Особенно интересно, что при присваивании списка `[0, 0, 0, 0, 0]` *разреженному* срезу, интерпретатор корректно замещает первым нулем значение `x[1]`, вторым нулем значение `x[3]` и т. д.

Функция sorted(), стр. 164, 170

Списки могут упорядочиваться в обратном порядке и сортироваться, так же как и другие итерируемые объекты, с помощью встроенных функций `reversed()` и `sorted()`, описываемых в подразделе «Итераторы, функции и операторы для работы с итерируемыми объектами» (стр. 163). Списки также имеют эквивалентные методы `list.reverse()` и `list.sort()`, которые работают непосредственно с самим списком (поэтому они ничего не возвращают); последний из них принимает те же необязательные аргументы, что и функция `sorted()`. Для сортировки списков строк без учета регистра символов используется распространенный прием – например, список `woods` можно было бы отсортировать так: `woods.sort(key=str.lower)`. Аргумент `key` используется, чтобы определить функцию, которая будет применяться к каждому элементу, и возвращать значение, участвующее в сравнении в процессе сортировки. Как уже отмечалось в предыдущей главе,

в разделе, где рассматривались вопросы сравнения строк (стр. 63), для языков, отличных от английского, сортировка строк в подразумеваемом людьми порядке может оказаться непростым делом.

Что касается вставки элементов, списки обеспечивают лучшую производительность при добавлении или удалении элементов в конце списка (`list.append()`, `list.pop()`). Падение производительности происходит, когда приходится отыскивать элементы в списке, например, с помощью методов `list.remove()` или `list.index()`, а также при использовании оператора `in` проверки на вхождение. Когда необходимо обеспечить высокую скорость поиска или проверки на вхождение, возможно, более удачным выбором будут множества и словари (оба типа коллекций описываются ниже, в этой же главе). Однако и для списков можно обеспечить высокую скорость поиска, если хранить их в отсортированном виде – в языке Python алгоритм сортировки особенно хорошо оптимизирован для случая сортировки частично отсортированных списков – и отыскивать элементы методом дихотомии (реализован в модуле `bisect`). (В главе 6 мы создадим свой класс списков, которые хранятся в отсортированном виде.)

Генераторы списков

Небольшие списки часто создаются как литералы, но длинные списки обычно создаются программным способом. Списки целых чисел могут создаваться с помощью выражения `list(range(`*n*`))`; когда необходим итератор целых чисел, достаточно функции `range()`; а для создания списков других типов часто используется оператор цикла `for ... in`. Предположим, например, что нам требуется получить список високосных годов в определенном диапазоне. Для начала мы могли бы использовать такой цикл:

```
leaps = []
for year in range(1900, 1940):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        leaps.append(year)
```

Функция range(), стр. 167

Когда функции `range()` передаются два целочисленных аргумента n и m, итератор возвращает последовательность целых чисел n, $n + 1$, ..., $m - 1$.

Конечно, если диапазон известен заранее, можно было бы использовать литерал списка, например, `leaps = [1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936]`.

Генератор списков – это выражение и цикл с дополнительным условием, заключенное в квадратные скобки, в котором цикл используется для создания элементов списка, а условие используется для исключения нежелательных элементов. В простейшем виде генератор списков записывается, как показано ниже:

```
[item for item in iterable]
```

Это выражение вернет список всех элементов объекта *iterable* и семантически ничем не отличается от выражения `list(iterable)`. Интересными генераторы списков делают две особенности – они могут использоваться как выражения и они допускают включение условной инструкции, вследствие чего мы получаем две типичные синтаксические конструкции использования генераторов списков:

```
[expression for item in iterable]
[expression for item in iterable if condition]
```

Вторая форма записи эквивалентна циклу:

```
temp = []
for item in iterable:
    if condition:
        temp.append(expression)
```

Обычно выражение *expression* является либо самим элементом *item*, либо некоторым выражением с его участием. Конечно, генератору списков не требуется временная переменная `temp[]`, которая необходима в версии с циклом `for ... in`.

Теперь можно переписать программный код создания списка високосных годов с использованием генератора списка. Мы сделаем это в три этапа. Сначала создадим список, содержащий все годы в указанном диапазоне:

```
leaps = [y for y in range(1900, 1940)]
```

То же самое можно было бы сделать с помощью выражения `leaps = list(range(1900, 1940))`. Теперь добавим простое условие, которое будет оставлять в списке только каждый четвертый год:

```
leaps = [y for y in range(1900, 1940) if y % 4 == 0]
```

И, наконец, получаем окончательную версию:

```
leaps = [y for y in range(1900, 1940)
         if (y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)]
```

Использование генератора списков в данном случае позволило уменьшить объем программного кода с четырех строк до двух – не так много, но в крупных проектах суммарная экономия может оказаться весьма существенной.

Так как генераторы списков воспроизводят списки, то есть итерируемые объекты, и сами генераторы списков используют итерируемые объекты, имеется возможность вкладывать генераторы списков друг в друга. Это эквивалентно вложению циклов `for ... in`. Например, если бы нам потребовалось сгенерировать список всех возможных кодов одежды для разных полов, разных размеров и расцветок, но исключая одежду для полных женщин, нужды и чаянья которых индустрия моды нередко игнорирует, мы могли бы использовать вложенные циклы `for ... in`, как показано ниже:

```
codes = []
for sex in "MF":             # мужская (Male), женская (Female)
    for size in "SMLX":      # маленький, средний, большой, очень большой
        if sex == "F" and size == "X":
            continue
        for color in "BGW":  # черный (Black), серый (Gray), белый (White)
            codes.append(sex + size + color)
```

Этот фрагмент воспроизводит список, содержащий 21 элемент – `['MSB', 'MSG', ..., 'FLW']`. Тот же самый список можно создать парой строк, если воспользоваться генераторами списков:

```
codes = [s + z + c for s in "MF" for z in "SMLX" for c in "BGW"
         if not (s == "F" and z == "X")]
```

Здесь каждый элемент списка воспроизводится выражением `s + z + c`. Кроме того, в генераторе списков несколько иначе построена логика обхода нежелательной комбинации пол/размер – проверка выполняется в самом внутреннем цикле, тогда как в версии с циклами `for ... in` эта проверка выполняется в среднем цикле. Любой генератор списков можно переписать, используя один или более циклов `for ... in`.

Выражения-генераторы, стр. 397

Если сгенерированный список получается очень большим, то, возможно, более эффективным было бы создавать очередные элементы списка по мере необходимости, вместо того чтобы создавать сразу весь список. Эту задачу можно решить с помощью выражений-генераторов, которые будут обсуждаться в главе 8.

Множества

Тип `set` – это разновидность коллекций, которая поддерживает оператор проверки на вхождение `in`, функцию `len()` и относится к разряду итерируемых объектов. Кроме того, множества предоставляют метод `set.isdisjoint()` и поддерживают операторы сравнения и битовые операторы (которые в контексте множеств используются для получения объединения, пересечения и т. д.). В языке Python имеется два встроенных типа множеств: изменяемый тип `set` и неизменяемый `frozenset`. При переборе элементов множества элементы могут следовать в произвольном порядке.

В состав множеств могут включаться только *хешируемые* объекты. Хешируемые объекты – это объекты, имеющие специальный метод `__hash__()`, на протяжении всего жизненного цикла объекта всегда возвращающий одно и то же значение, которые могут участвовать в операциях сравнения на равенство посредством специального метода `__eq__()`. (Специальные методы – это методы, имена которых начинаются и оканчиваются двумя символами подчеркивания; они описываются в главе 6.)

Все встроенные неизменяемые типы данных, такие как `float`, `frozenset`, `int`, `str` и `tuple`, являются хешируемыми объектами и могут добавляться во множества. Встроенные изменяемые типы данных, такие как `dict`, `list` и `set`, не являются хешируемыми объектами, так как значение хеша в каждом конкретном случае зависит от содержащихся в объекте элементов, поэтому они не могут добавляться в множества.

Множества могут сравниваться между собой с использованием стандартных операторов сравнения (`<`, `<=`, `==`, `!=`, `>=`, `>`). Обратите внимание: операторы `==` и `!=` имеют обычный смысл, и сравнение выполняется путем поэлементного сравнения (или рекурсивно при наличии таких вложенных элементов, как кортежи и фиксированные множества (`frozenset`)), но остальные операторы сравнения выполняют сравнение подмножеств и надмножеств, как вскоре будет показано.

Тип set

Тип `set` – это неупорядоченная коллекция из нуля или более ссылок на объекты, указывающих на хешируемые объекты. Множества относятся к категории изменяемых типов, поэтому легко можно добавлять и удалять их элементы, но, так как они являются неупорядоченными коллекциями, к ним не применимо понятие индекса и не применима операция извлечения среза. На рис. 3.3 иллюстрируется множество, созданное следующим фрагментом программного кода:

```
S = {7, "veil", 0, -29, ("x", 11), "sun", frozenset({8, 4, 7}), 913}
```

Тип данных `set` может вызываться как функция `set()` – без аргументов она возвращает пустое множество; с аргументом типа `set` возвращает поверхностную копию аргумента; в случае, если аргумент имеет другой тип, выполняется попытка преобразовать его в объект типа `set`. Эта функция принимает не более одного аргумента. Кроме того, непустые множества могут создаваться без использования функции `set()`, а пустые множества могут создаваться только с помощью функции `set()` – их нельзя создать с помощью пары пустых скобок.[1] Множество, состоящее из одного или более элементов, может быть создано с помощью последовательности элементов, разделенных запятыми, заключенной в фигурные скобки. Другой способ создания множеств заключается в использовании генераторов множеств – эта тема будет рассматриваться ниже в соответствующем подразделе. Множества всегда содержат уникальные элементы – добавление повторяющихся элементов возможно, но не имеет

Поверхностное и глубокое копирование, стр. 173

Генераторы множеств, стр. 149

[1] Пустые фигурные скобки `{}` используются для создания пустого словаря, как будет показано в следующем разделе.

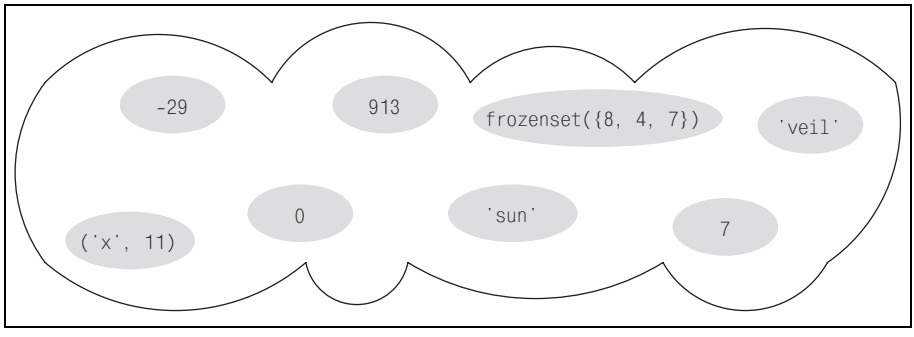

Рис. 3.3. *Множество – это неупорядоченная коллекция уникальных элементов*

смысла. Например, следующие три множества являются эквивалентными: `set("apple")`, `set("aple")` и `{'e', 'p', 'a', 'l'}`. Благодаря этой их особенности множества часто используются для устранения повторяющихся значений. Например, если предположить, что `x` – это список строк, то после выполнения инструкции `x = list(set(x))` в списке останутся только уникальные строки, причем располагаться они могут в произвольном порядке.

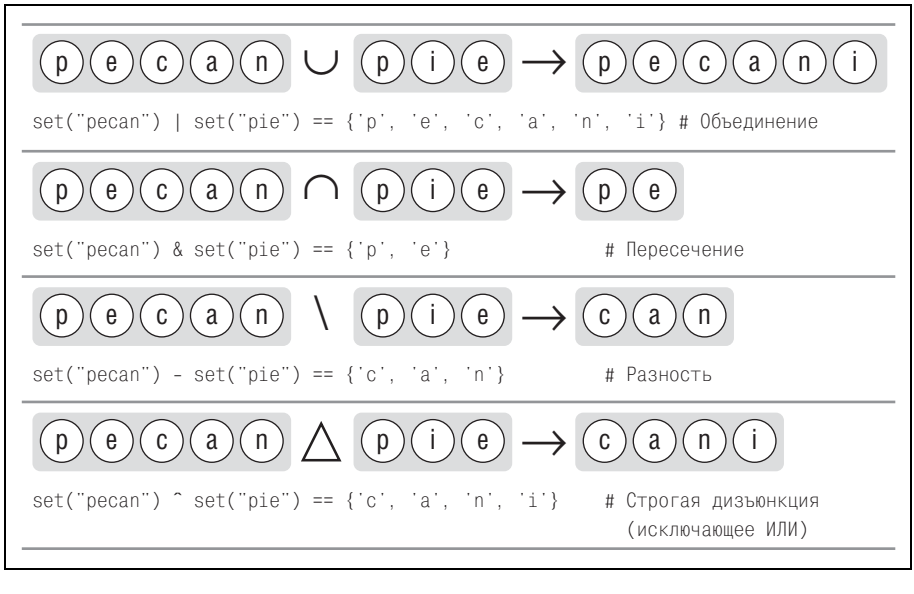

Рис. 3.4. *Стандартные операторы множеств*

Множества поддерживают встроенную функцию `len()` и быструю проверку на вхождение с помощью операторов `in` и `not in`. Они также предоставляют типичный набор операторов, как показано на рис. 3.4. Полный перечень методов и операторов, применимых к множествам, приводится в табл. 3.2. Все методы семейства «update» (`set.update()`, `set.intersection_update()` и т. д.) могут принимать в качестве аргумента любые итерируемые объекты, но эквивалентные им комбинированные операторы присваивания (`|=`, `&=` и т. д.) требуют, чтобы оба операнда были множествами.

Таблица 3.2. Методы и операторы множеств

Синтаксис	Описание
`s.add(x)`	Добавляет элементы `x` во множество `s`, если они отсутствуют в `s`
`s.clear()`	Удаляет все элементы из множества `s`
`s.copy()`	Возвращает поверхностную копию множества `s`[a]
`s.difference(t)` `s - t`	Возвращает новое множество, включающее элементы множества `s`, которые отсутствуют в множестве `t`[a]
`s.difference_update(t)` `s -= t`	Удаляет из множества `s` все элементы, присутствующие в множестве `t`
`s.discard(x)`	Удаляет элемент `x` из множества `s`, если он присутствует в множестве `s`; смотрите также метод `set.remove()`
`s.intersection(t)` `s & t`	Возвращает новое множество, включающее элементы, присутствующие одновременно в множествах `s` и `t`[a]
`s.intersection_update(t)` `s &= t`	Оставляет во множестве `s` пересечение множеств `s` и `t`
`s.isdisjoint(t)`	Возвращает `True`, если множества `s` и `t` не имеют общих элементов[a]
`s.issubset(t)` `s <= t`	Возвращает `True`, если множество `s` эквивалентно множеству `t` или является его подмножеством; чтобы проверить, является ли множество `s` только подмножеством множества `t`, следует использовать проверку `s < t`[a]
`s.issuperset(t)` `s >= t`	Возвращает `True`, если множество `s` эквивалентно множеству `t` или является его надмножеством; чтобы проверить, является ли множество `s` только надмножеством множества `t`, следует использовать проверку `s > t`[a]

Поверхностное и глубокое копирование, стр. 173

[a] Этот метод и соответствующий ему оператор (если таковой имеется) могут также применять к фиксированным множествам.

Таблица 3.2 (продолжение)

Синтаксис	Описание
`s.pop()`	Возвращает и удаляет случайный элемент множества `s` или возбуждает исключение `KeyError`, если `s` — это пустое множество
`s.remove(x)`	Удаляет элемент `x` из множества `s` или возбуждает исключение `KeyError`, если элемент `x` отсутствует в множестве `s`; смотрите также метод `set.discard()`
`s.symmetric_difference(t)` `s ^ t`	Возвращает новое множество, включающее все элементы, присутствующие в множествах `s` и `t`, за исключением элементов, присутствующих в обоих множествах одновременно[a]
`s.symmetric_difference_update(t)` `s ^= t`	Возвращает в множестве `s` результат строгой дизъюнкции множеств `s` и `t`[a]
`s.union(t)` `s | t`	Возвращает новое множество, включающее все элементы множества `s` и все элементы множества `t`, отсутствующие в множестве `s`[a]
`s.update(t)` `s |= t`	Добавляет во множество `s` все элементы множества `t`, отсутствующие в множестве `s`

Типичный случай использования множеств – когда необходимо организовать быструю проверку на вхождение. Например, нам может потребоваться выводить на экран инструкцию о порядке использования программы, когда пользователь вводит аргументы «-h» или «--help»:

```
if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
```

Другой типичный случай использования множеств – когда необходимо избежать обработки повторяющихся элементов данных. Например, предположим, что имеется итерируемый объект (такой как список), содержащий IP-адреса, извлеченные из файлов журнала веб-сервера, и необходимо выполнить некоторые действия, причем не более одного раза для каждого отдельного IP-адреса. Допустим, что IP-адреса сохраняются в хешируемом и итерируемом объекте `ips` и что для каждого адреса должна вызываться функция `process_ip()`, которая уже определена. Тогда следующие фрагменты программного кода сделают то, что нам требуется, хотя и немного по-разному:

```
seen = set()
for ip in ips:
    if ip not in seen:
        seen.add(ip)
        process_ip(ip)
```

```
for ip in set(ips):
    process_ip(ip)
```

Во фрагменте слева, если очередной IP-адрес еще не обрабатывался, он добавляется во множество `seen` и обрабатывается, в противном случае адрес пропускается. В фрагменте справа мы изначально имеем дело только с уникальными IP-адресами. Различие между этими фрагментами заключается, во-первых, в том, что в левом фрагменте создается множество `seen`, необходимость в котором отсутствует в правом фрагменте, и, во-вторых, в левом фрагменте IP-адреса обрабатываются в порядке, в каком они присутствуют в объекте `ips`, тогда как в правом фрагменте они обрабатываются в произвольном порядке.

Фрагмент справа выглядит проще, но, если порядок обработки элементов объекта `ips` имеет значение, придется использовать фрагмент слева или изменить первую строку во фрагменте справа – например, так: `for ip in sorted(set(ips)):`, если этого будет достаточно, чтобы получить желаемый порядок следования. Теоретически фрагмент справа может оказаться более медленным при большом количестве элементов в объекте `ips`, поскольку в нем множество создается целиком, а не с определенным шагом.

Множества также могут использоваться для удаления требуемых элементов. Например, если представить, что у нас имеется список имен файлов и нам необходимо исключить из него файлы с инструкциями по сборке (возможно, по той простой причине, что они генерируются автоматически, а не создаются вручную), мы могли бы использовать следующий прием:

```
filenames = set(filenames)
for makefile in {"MAKEFILE", "Makefile", "makefile"}:
    filenames.discard(makefile)
```

Этот фрагмент удалит все makefile, присутствующие в списке, имена которых следуют стандарту использования заглавных символов. Этот фрагмент ничего не будет делать, если в списке отсутствуют искомые файлы. То же самое может быть реализовано в одной строке программного кода при помощи оператора получения разности множеств (`-`):

```
filenames = set(filenames) - {"MAKEFILE", "Makefile", "makefile"}
```

Кроме того, мы могли бы удалить элементы с помощью метода `set.remove()`, хотя этот метод возбуждает исключение `KeyError`, если удаляемый элемент отсутствует во множестве.

Генераторы множеств

В дополнение к возможности создавать множества с помощью функции `set()` или литералов, существует возможность создавать множества с помощью *генераторов множеств*. Генератор множества – это выражение и цикл с необязательным условием, заключенные в фигурные скобки. Подобно генераторам списков, генераторы множеств поддерживают две формы записи:

```
{expression for item in iterable}
{expression for item in iterable if condition}
```

Мы могли бы использовать генераторы множеств для фильтрации нежелательных элементов (когда порядок следования элементов не имеет значения), как показано ниже:

```
html = {x for x in files if x.lower().endswith((".htm", ".html"))}
```

Если предположить, что `files` – это список имен файлов, то данный генератор множества создает множество `html`, в котором хранятся только имена файлов с расширениями *.htm* и *.html*, независимо от регистра символов.

Как и в случае с генераторами списков, в генераторах множеств используются итерируемые объеты, которые в свою очередь могут быть генераторами множеств (или генераторами любого другого типа), что позволяет создавать весьма замысловатые генераторы множеств.

Тип frozenset

Поверхностное и глубокое копирование, стр. 173

Фиксированное множество (`frozenset`) – это множество, которое после создания невозможно изменить. Хотя при этом мы, конечно, можем повторно связать переменную, которая ссылалась на фиксированное множество, с чем-то другим. Фиксированные множества могут создаваться только в результате обращения к имени типа `frozenset` как к функции. При вызове `frozenset()` без аргументов возвращается пустое фиксированное множество; с аргументом типа `frozenset` возвращается поверхностная копия аргумента; если аргумент имеет другой тип, выполняется попытка преобразовать его в объект типа `frozenset`. Эта функция принимает не более одного аргумента.

Поскольку фиксированные множества относятся к категории неизменяемых объектов, они поддерживают только те методы и операторы, которые воспроизводят результат, не оказывая воздействия на фиксированное множество или на множества, к которым они применяются. В табл. 3.2 (на стр. 147) перечислены все методы множеств из которых фиксированными множествами поддерживаются: `frozenset.copy()`, `frozenset.difference()` (`-`), `frozenset.intersection()` (`&`), `frozenset.isdisjoint()`, `frozenset.issubset()` (`<=` и `<` для выявления подмножеств), `frozenset.issuperset()` (`>=` и `>` для выявления надмножеств), `frozenset.union()` (`|`) и `frozenset.symmetric_difference()` (`^`), – то есть все те, что помечены в таблице знаком сноски [a].

Если двухместный оператор применяется ко множеству и фиксированному множеству, тип результата будет совпадать с типом операнда, стоящего слева от оператора. То есть если предположить, что `f` – это фиксированное множество, а `s` – это обычное множество, то выражение `f & s` вернет объект типа `frozenset`, а выражение `s & f` – объект ти-

па set. В случае операторов == и != порядок операндов не имеет значения, и выражение f == s вернет True, только если оба множества содержат одни и те же элементы.

Другое следствие неизменности фиксированных множеств заключается в том, что они соответствуют критерию хеширования, предъявляемому к элементам множеств, и потому множества и фиксированные множества могут содержать другие фиксированные множества.

В следующем разделе, а также в упражнениях в конце главы мы встретим множество примеров использования множеств.

Отображения

Отображениями называются типы данных, поддерживающие оператор проверки на вхождение (in), функцию len() и возможность обхода элементов в цикле. Отображения – это коллекции пар элементов «ключ-значение», которые предоставляют методы доступа к элементам и их ключам и значениям. При выполнении итераций порядок следования элементов отображений может быть произвольным. В языке Python имеется два типа отображений: встроенный тип dict и тип collections.defaultdict, определяемый в стандартной библиотеке. Мы будем использовать термин *словарь* для ссылки на любой из этих типов, когда различия между ними не будут иметь никакого значения.

В качестве ключей словарей могут использоваться только хешируемые объекты, поэтому в качестве ключей словаря такие неизменяемые типы, как float, frozenset, int, str и tuple, использовать допускается, а изменяемые типы, такие как dict, list и set, – нет. С другой стороны, каждому ключу соответствует некоторое значение, которое может быть ссылкой на объект любого типа, включая числа, строки, списки, множества, словари, функции и т. д.

Хешируемые объекты, стр. 145

Словари могут сравниваться с помощью стандартных операторов сравнения (<, <=, ==, !=, >=, >), при этом сравнивание производится поэлементно (и рекурсивно, при наличии вложенных элементов, таких как кортежи или словари в словарях). Пожалуй, единственными операторами сравнения, применение которых к словарям имеет смысл, являются операторы == и !=.

Словари

Тип dict – это неупорядоченная коллекция из нуля или более пар «ключ-значение», в которых в качестве ключей могут использоваться ссылки на хешируемые объекты, а в качестве значений – ссылки на объекты любого типа. Словари относятся к категории изменяемых типов, поэтому легко можно добавлять и удалять их элементы, но так

как они являются неупорядоченными коллекциями, к ним не применимо понятие индекса и не применима операция извлечения среза.

Поверхностное и глубокое копирование, стр. 173

Тип данных `dict` может вызываться как функция `dict()` – без аргументов она возвращает пустой словарь; если в качестве аргумента передается отображение, возвращается словарь, основанный на этом отображении: например, с аргументом типа `dict` возвращается поверхностная копия словаря. Существует возможность передавать в качестве аргумента последовательности, если каждый элемент последовательности в свою очередь является последовательностью из двух объектов, первый из которых используется в качестве ключа, а второй – в качестве значения. Как вариант, для создания словарей, в которых ключи являются допустимыми идентификаторами языка Python, можно использовать именованные аргументы; тогда имена аргументов будут играть роль ключей, а значения аргументов – роль значений ключей. Кроме того, словари могут создаваться с помощью фигурных скобок – пустые скобки `{}` создадут пустой словарь. Непустые фигурные скобки должны содержать один или более элементов, разделенных запятыми, каждый из которых состоит из ключа, символа двоеточия и значения. Еще один способ создания словарей заключается в использовании генераторов словарей – эта тема будет рассматриваться ниже, в соответствующем подразделе.

Именованные аргументы, стр. 206

Генераторы словарей, стр. 160

Ниже приводятся несколько способов создания словарей – все они создают один и тот же словарь:

```
d1 = dict({"id": 1948, "name": "Washer", "size": 3})
d2 = dict(id=1948, name="Washer", size=3)
d3 = dict([("id", 1948), ("name", "Washer"), ("size", 3)])
d4 = dict(zip(("id", "name", "size"), (1948, "Washer", 3)))
d5 = {"id": 1948, "name": "Washer", "size": 3}
```

Функция `zip()`, стр. 169

Словарь `d1` создается с помощью литерала словаря. Словарь `d2` создается с помощью именованных аргументов. Словари `d3` и `d4` создаются из последовательностей, а словарь `d5` создается из литерала словаря. Встроенная функция `zip()`, использованная при создании словаря `d4`, возвращает список кортежей, первый из которых содержит первые элементы всех итерируемых аргументов функции `zip()`, второй – вторые элементы и т. д. Синтаксис, основанный на применении именованных аргументов (использованный при создании словаря `d2`), обычно является наиболее компактным и удобным, но при этом ключами могут быть только допустимые идентификаторы.

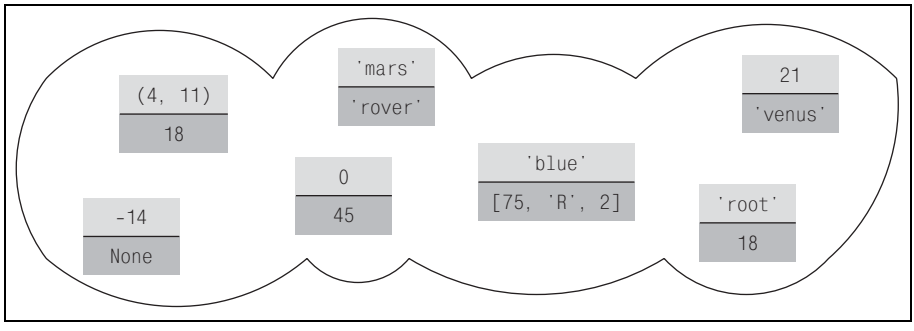

***Рис. 3.5.** Словарь – это неупорядоченная коллекция элементов (ключ, значение) с уникальными ключами*

На рис. 3.5 демонстрируется словарь, созданный следующим фрагментом программного кода:

```
d = {"root": 18, "blue": [75, "R", 2], 21: "venus", -14: None,
     "mars": "rover", (4, 11): 18, 0: 45}
```

Ключи словарей являются уникальными, поэтому если в словарь добавляется пара «ключ-значение» с ключом, который уже присутствует в словаре, в результате происходит замена значения существующего ключа новым значением. Для доступа к отдельным элементам используются квадратные скобки: например, выражение `d["root"]` вернет `18`, выражение `d[21]` вернет строку `"venus"`, а выражение `d[91]` применительно к словарю, изображенному на рис. 3.5, возбудит исключение `KeyError`.

Квадратные скобки могут также использоваться для добавления и удаления элементов словаря. Чтобы добавить новый элемент, используется оператор `=`, например, `d["X"] = 59`. Для удаления элементов используется инструкция `del`, например, инструкция `del d["mars"]` удалит из словаря элемент с ключом `"mars"` или возбудит исключение `KeyError`, если элемент с указанным ключом отсутствует в словаре. Кроме того, элементы могут удаляться (и возвращаться вызывающей программе) методом `dict.pop()`.

Словари поддерживают встроенную функцию `len()` и для ключей поддерживают возможность быстрой проверки на вхождение с помощью операторов `in` и `not in`. В табл. 3.3 перечислены все методы словарей.

Так как словари содержат пары «ключ-значение», у нас может возникнуть потребность обойти в цикле элементы словаря (ключ, значение) по значениям или по ключам. Например, ниже приводятся два эквивалентных способа обхода пар «ключ-значение»:

```
for item in d.items():
    print(item[0], item[1])
```

```
for key, value in d.items():
    print(key, value)
```

Таблица 3.3. Методы словарей

Синтаксис	Описание
`d.clear()`	Удаляет все элементы из словаря `d`
`d.copy()`	Возвращает поверхностную копию словаря `d`
`d.fromkeys(s, v)`	Возвращает словарь типа `dict`, ключами которого являются элементы последовательности `s`, а значениями либо `None`, либо `v`, если аргумент `v` определен
`d.get(k)`	Возвращает значение ключа `k` или `None`, если ключ `k` отсутствует в словаре
`d.get(k, v)`	Возвращает значение ключа `k` или `v`, если ключ `k` отсутствует в словаре
`d.items()`	Возвращает представление[a] всех пар (ключ, значение) в словаре `d`
`d.keys()`	Возвращает представление[a] всех ключей словаря `d`
`d.pop(k)`	Возвращает значение ключа `k` и удаляет из словаря элемент с ключом `k` или возбуждает исключение `KeyError`, если ключ `k` отсутствует в словаре
`d.pop(k, v)`	Возвращает значение ключа `k` и удаляет из словаря элемент с ключом `k` или возвращает значение `v`, если ключ `k` отсутствует в словаре
`d.popitem()`	Возвращает и удаляет произвольную пару (ключ, значение) из словаря `d` или возбуждает исключение `KeyError`, если словарь `d` пуст
`d.setdefault(k, v)`	То же, что и `dict.get()` за исключением того, что, если ключ `k` в словаре отсутствует, в словарь вставляется новый элемент с ключом `k` и со значением `None` или `v`, если аргумент `v` задан
`d.update(a)`	Добавляет в словарь `d` пары (ключ, значение) из `a`, которые отсутствуют в словаре `d`, а для каждого ключа, который уже присутствует в словаре `d`, выполняется замена соответствующим значением из `a`; `a` может быть словарем, итерируемым объектом с парами (ключ, значение) или именованными аргументами
`d.values()`	Возвращает представление[a] всех значений в словаре `d`

Поверхностное и глубокое копирование, стр. 173

[a] Представления словарей можно трактовать и использовать как итерируемые объекты. Они обсуждаются ниже, в этом же разделе.

Обход значений в словаре выполняется похожим способом:

```
for value in d.values():
    print(value)
```

Для обхода ключей в словаре можно использовать метод `dict.keys()` или просто интерпретировать словарь как итерируемый объект и выполнить итерации по его ключам, как показано в следующих двух фрагментах:

```
for key in d:
    print(key)
```

```
for key in d.keys():
    print(key)
```

Если необходимо изменить значения в словаре, то можно выполнить обход ключей словаря в цикле и изменить значения, используя оператор квадратных скобок. Например, ниже показано, как можно было бы увеличить все значения в словаре `d`, если предполагать, что все значения являются числами:

```
for key in d:
    d[key] += 1
```

Методы `dict.items()`, `dict.keys()` и `dict.values()` возвращают *представления словарей*. Представление словаря – это в действительности итерируемый объект, доступный только для чтения и хранящий элементы, ключи или значения словаря в зависимости от того, какое представление было запрошено.

Вообще, мы легко можем интерпретировать представления как итерируемые объекты. Однако между представлениями и обычными итерируемыми объектами есть два различия. Одно из них заключается в том, что если словарь, для которого было получено представление, изменяется, то представление будет отражать эти изменения. Другое отличие состоит в том, что представления ключей и элементов поддерживают некоторые операции, свойственные множествам. Допустим, у нас имеются представление словаря `v` и множество или представление словаря `x`; для этой пары поддерживаются следующие операции:

```
v & x    # Пересечение
v | x    # Объединение
v - x    # Разность
v ^ x    # Строгая дизъюнкция
```

Для проверки наличия некоторого определенного ключа в словаре можно использовать оператор проверки на вхождение `in`, например, *`x in d`*. Чтобы выяснить, какие ключи из заданного множества присутствуют в словаре, можно использовать оператор пересечения. Например:

```
d = {}.fromkeys ("ABCD", 3) # d == {'A': 3, 'B': 3, 'C': 3, 'D': 3}
s = set("ACX")              # s == {'A', 'C', 'X'}
matches = d.keys() & s      # matches == {'A', 'C'}
```

Обратите внимание, что в комментариях в этом фрагменте программного кода мы указали ключи в алфавитном порядке – это сделано лишь для простоты восприятия, поскольку словари и множества являются неупорядоченными коллекциями.

Словари часто используются для хранения счетчиков уникальных элементов. Один такой пример – подсчет числа вхождений каждого отдельного слова в файле. Ниже приводится программный код законченной программы (*uniqwords1.py*)), которая выводит в алфавитном

порядке список слов с количеством вхождений каждого из них во всех файлах, перечисленных в командной строке:

```
import string
import sys

words = {}
strip = string.whitespace + string.punctuation + string.digits + "\"'"
for filename in sys.argv[1:]:
    for line in open(filename):
        for word in line.lower().split():
            word = word.strip(strip)
            if len(word) > 2:
                words[word] = words.get (word, 0) + 1

for word in sorted(words):
    print("'{0}' occurs {1} times".format(word, words[word]))
```

Программа начинается с создания пустого словаря с именем `words`, затем путем конкатенации некоторых полезных строк, объявленных в модуле `string`, создается строка, содержащая все символы, которые мы будем игнорировать. Мы выполняем итерации по всем именам файлов, полученным в виде аргументов командной строки, и по всем строкам в каждом файле. Описание функции `open()` смотрите во врезке «Чтение и запись текстовых файлов» (стр. 157). Мы не указываем кодировку символов (потому что не знаем, какой она будет в каждом файле) и позволяем интерпретатору открывать каждый файл, используя по умолчанию локальную кодировку. После приведения всех символов строки к нижнему регистру она разбивается на слова, после этого с обоих концов каждого слова удаляются нежелательные символы. Если в получившемся слове осталось хотя бы три символа, мы выполняем обновление словаря.

Мы не можем использовать синтаксис `words[word] += 1`, потому что в самый первый раз, когда слово `word` еще отсутствует в словаре, это выражение будет возбуждать исключение `KeyError` — мы же не можем увеличивать значение элемента, отсутствующего в словаре. Поэтому мы используем иной подход. Мы вызываем метод `dict.get()` со значением по умолчанию 0. Если слово уже присутствует в словаре, метод `dict.get()` вернет существующее значение и это значение, увеличенное на 1, будет записано как новое значение элемента. Если слово отсутствует в словаре, метод `dict.get()` вернет значение по умолчанию 0 и это значение, увеличенное на 1 (то есть 1), будет записано как значение нового элемента, ключом которого является строка `word`. Чтобы пояснить ситуацию, ниже приводятся два фрагмента программного кода, выполняющего одно и то же, хотя код, использующий метод `dict.get()`, имеет более высокую эффективность:

```
words[word] = words.get(word, 0) + 1
```

```
if word not in words:
    words[word] = 0
words[word] += 1
```

В следующем подразделе мы познакомимся со словарями, имеющими значения по умолчанию, и рассмотрим альтернативное решение.

После того, как все встретившиеся слова будут накоплены в словаре, выполняются итерации по его ключам (по словам) в алфавитном порядке и выводятся слова и число раз, которое они встречаются.

Использование метода `dict.get()` позволяет легко обновлять значения в словаре, возвращая значения, если это отдельные элементы данных, такие как числа или строки. Но как быть, если каждое значение само по себе является коллекцией? Чтобы продемонстрировать, как обрабатывать такие значения, ниже приводится программа, которая читает

Чтение и запись текстовых файлов

Глава 7, работа с файлами, стр. 334

Файлы открываются с помощью встроенной функции `open()`, которая возвращает «объект файла» (типа `io.TextIOWrapper` для текстовых файлов). Функция `open()` принимает один обязательный аргумент – имя файла, которое может включать путь к файлу, и до шести необязательных аргументов, два из которых коротко описываются здесь. Второй аргумент определяет *режим* работы с файлом, то есть он указывает, будет ли файл интерпретироваться как текстовый или как двоичный, и для выполнения каких действий будет открыт файл – для чтения, для записи, для дополнения в конец или комбинации этих действий.

Кодировки символов, стр. 112

Для работы с текстовыми файлами Python использует кодировку символов, зависящую от платформы. Поэтому, возможно, лучше будет указывать кодировку с помощью аргумента `encoding` функции `open()`, то есть обычный синтаксис, используемый для открытия файлов, имеет следующий вид:

```
fin = open(filename, encoding="utf8")       # для чтения текста
fout = open(filename, "w", encoding="utf8") # для записи текста
```

Поскольку по умолчанию функция `open()` использует режим «для чтения», и для указания кодировки используется именованный аргумент `encoding`, а не позиционный, то при открытии файла для чтения можно опустить другие необязательные позиционные аргументы. Аналогично, при открытии файла для записи мы можем указывать только те аргументы, которые действительно необходимы. (Вопросы передачи аргументов подробно рассматриваются в главе 4.)

Как только файл будет открыт для чтения в текстовом режиме, можно будет прочитать его целиком в одну строку, используя метод объекта файла `read()`, или в список строк, используя метод объекта файла `readlines()`. Типичный прием построчного чтения содержимого файла основан на интерпретации объекта файла как итератора:

```
for line in open(filename, encoding="utf8"):
    process(line)
```

Этот прием работает, потому что объект файла допускает выполнение итераций по нему, как по последовательности, каждый элемент которой представляет собой строку, содержащую отдельную строку из файла. Строки, которые в этом случае получает программа, содержат символы перевода строки `\n`.

Если в качестве режима указать «w», файл будет открыт в режиме «записи текста». Запись в файл может производиться с помощью метода объекта файла `write()`, который в качестве аргумента принимает единственную строку. Каждая записываемая строка уже должна содержать символ перевода строки `\n`. При выполнении чтения и записи Python автоматически преобразует символы `\n` в последовательность символов завершения строки, характерную для той или иной платформы.

После окончания работы с объектом файла можно вызвать его метод `close()` – это приведет к выталкиванию буферов вывода на диск. В небольших программах на языке Python обычно не принято беспокоиться о вызове функции `close()`, поскольку Python делает это автоматически, когда объект файла выходит из текущей области видимости. Если возникают какие-либо проблемы, объект тут же сообщает о них возбуждением исключений.

содержимое файлов HTML, имена которых указываются в командной строке, и выводит список различных веб-сайтов, ссылки на которые присутствуют в файлах, и список файлов, в которых эти ссылки встречаются, ниже каждого веб-сайта. По своей структуре программа (*external_sites.py*) очень похожа на программу подсчета числа вхождений отдельных слов, которую мы только что рассмотрели. Ниже приводится основная часть программы:

```
sites = {}
for filename in sys.argv[1:]:
    for line in open(filename):
        i = 0
        while True:
            site = None
```

```
i = line.find("http://", i)
if i > -1:
    i += len("http://")
    for j in range(i, len(line)):
        if not (line[j].isalnum() or line[j] in ".-"):
            site = line[i:j].lower()
            break
    if site and "." in site:
        sites.setdefault(site, set()).add(filename)
    i = j
else:
    break
```

Программа начинается с создания пустого словаря. Затем выполняются итерации по списку файлов, перечисленных в командной строке, и по строкам в каждом файле. Нам необходимо учесть тот факт, что в каждой строке может содержаться произвольное число ссылок на веб-сайты, поэтому мы продолжаем вызывать метод `str.find()` в цикле, пока поиск не завершится неудачей. Если обнаруживается строка «http://», мы увеличиваем значение переменной `i` (начальная позиция в строке) на длину строки «http://» и затем просматриваем все последующие символы, пока не будет найден символ, недопустимый в именах веб-сайтов. Если обнаружена ссылка на сайт (в качестве простой проверки мы убеждаемся, что она содержит символ точки), мы добавляем ее в словарь.

Мы не можем использовать выражение `sites[site].add(filename)`, потому что в самый первый раз, когда сайт еще отсутствует в словаре, это выражение будет возбуждать исключение `KeyError` — т. к. нельзя добавить новое значение к множеству, которое пока отсутствует в словаре. Поэтому мы используем иной подход. Метод `dict.setdefault()` возвращает ссылку на элемент словаря с заданным ключом (первый аргумент). Если такой элемент отсутствует, метод создает новый элемент с указанным ключом и устанавливает в качестве значения либо `None`, либо указанное значение по умолчанию (второй аргумент). В данном случае в качестве значения по умолчанию передается результат вызова функции `set()`, то есть пустое множество. Поэтому вызов метода `dict.setdefault()` всегда будет возвращать ссылку на значение, либо существовавшее ранее, либо на вновь созданное. (Безусловно, если указанный ключ не является хешируемым значением, будет возбуждено исключение `TypeError`.)

В данном примере возвращаемая ссылка всегда будет указывать на множество (пустое множество при первом упоминании каждого конкретного ключа, то есть сайта), после чего мы добавляем имя файла, где встречена ссылка на сайт, ко множеству имен файлов для данного сайта. Использование множества гарантирует, что даже при наличии в файле нескольких ссылок на один и тот же сайт имя файла будет записано всего один раз.

Чтобы прояснить, как функционирует метод `dict.setdefault()`, ниже приводятся два эквивалентных фрагмента программного кода:

```
sites.setdefault(site, set()).add(fname)
```

```
if site not in sites:
    sites[site] = set()
sites[site].add(fname)
```

Для полноты картины ниже приводится остальная часть программы:

```
for site in sorted(sites):
    print("{0} is referred to in:".format(site))
    for filename in sorted(sites[site], key=str.lower):
        print("    {0}".format(filename))
```

Функция sorted(), стр. 164, 170

Под каждым веб-сайтом выводится с отступом список файлов, в которых встречается ссылка на этот веб-сайт. Вызов функции `sorted()` во внешнем цикле `for ... in` выполняет сортировку ключей словаря – всякий раз, когда словарь используется в контексте, где требуется итерируемый объект, используются его ключи. Если необходимо выполнить итерации по элементам (ключ, значение) или по значениям, можно использовать методы `dict.items()` или `dict.values()`. Внутренний цикл `for ... in` выполняет итерации по отсортированному списку имен файлов, присутствующих во множестве имен файлов для данного сайта.

Генераторы словарей

Генератор словарей – это выражение и цикл с необязательным условием, заключенное в фигурные скобки, очень напоминающее генератор множеств. Подобно генераторам списков и множеств, генераторы словарей поддерживают две формы записи:

```
{keyexpression: valueexpression for key, value in iterable}
{keyexpression: valueexpression for key, value in iterable if condition}
```

Ниже показано, как можно использовать генератор словарей для создания словаря, в котором каждый ключ является именем файла в текущем каталоге, а каждое значение – это размер файла в байтах:

```
file_sizes = {name: os.path.getsize(name) for name in os.listdir(".")}
```

Модули os и os.path, стр. 261

Функция `os.listdir()` из модуля `os` («operating system» – операционная система) возвращает список файлов и каталогов в указанном каталоге, но при этом в список никогда не включаются специальные имена каталогов «`.`» или «`..`». Функция `os.path.getsize()` возвращает размер заданного файла в байтах. Чтобы отфильтровать каталоги и другие элементы списка, не являющиеся файлами, можно добавить дополнительное условие:

```
file_sizes = {name: os.path.getsize(name) for name in os.listdir(".")
              if os.path.isfile(name)}
```

Функция `os.path.isfile()` из модуля `os.path` возвращает `True`, если указанный путь соответствует файлу, и `False` – в противном случае, то есть для каталогов, ссылок и тому подобного.

Генераторы словарей могут также использоваться для создания инвертированных словарей. Например, пусть имеется словарь `d`, тогда мы можем создать новый словарь, ключами которого будут значения словаря `d`, а значениями – ключи словаря `d`:

```
inverted_d = {v: k for k, v in d.items()}
```

Полученный словарь можно инвертировать обратно и получить первоначальный словарь – при условии, что все значения в первоначальном словаре были уникальными, однако инверсия будет терпеть неудачу, с возбуждением исключения `TypeError`, если какое-либо значение окажется не хешируемым.

Точно так же, как и в случае с генераторами списков и множеств, в качестве итерируемого объекта в генераторах словарей могут использоваться другие генераторы, то есть это могут быть вложенные генераторы любого типа.

Словари со значениями по умолчанию

Словари со значениями по умолчанию – это обычные словари, они поддерживают те же самые методы и операторы, что и обычные словари.[1] Единственное, что отличает такие словари от обычных словарей, – это способ обработки отсутствующих ключей, но во всех остальных отношениях они ничем не отличаются друг от друга.

При обращении к несуществующему («отсутствующему») ключу словаря возбуждается исключение `KeyError`. Это очень удобно, так как нередко для нас бывает желательно знать об отсутствии ключа, который, согласно нашим предположениям, может присутствовать. Но в некоторых случаях бывает необходимо, чтобы в словаре присутствовали все ключи, которые мы используем, даже если это означает, что элемент с заданным ключом добавляется в словарь в момент первого обращения к нему.

Например, допустим, что имеется словарь `d`, который *не* имеет элемента с ключом `m`, тогда выражение `x = d[m]` возбудит исключение `KeyError`. Если `d` – это словарь со значениями по умолчанию, созданный соответствующим способом, а элемент с ключом `m` принадлежит такому словарю, то при обращении к нему будет возвращено соответствующее значение, как и в случае с обычным словарем. Но если в словаре со значениями по умолчанию отсутствует ключ `m`, то будет создан новый

[1] Примечание для программистов, использующих объектно-ориентированный стиль: `defaultdict` – это подкласс класса `dict`.

элемент словаря с ключом `m` и со значением по умолчанию, и будет возвращено значение этого, вновь созданного элемента.

Пример unique-words1.py, стр. 155

Ранее мы написали небольшую программу, которая подсчитывала количество вхождений каждого отдельного слова в файлы, имена которых передавались в виде аргументов командной строки. В этой программе создавался словарь слов, как показано ниже:

```
words = {}
```

Каждый ключ в словаре `words` является словом, а значение – целым числом, в котором хранится количество вхождений данного слова во всех файлах. Ниже показано, как увеличивается значение счетчика, соответствующего некоторому слову:

```
words[word] = words.get(word, 0) + 1
```

Мы вынуждены были использовать метод `dict.get()`, чтобы учесть случай, когда слово встречается впервые (когда необходимо создать новый элемент со значением счетчика, равным 1), а также случаи, когда слово встречается повторно (когда необходимо прибавить 1 к значению счетчика для уже существующего слова).

При создании словаря со значениями по умолчанию мы можем определять *фабричную функцию*. Фабричная функция – это функция, которая вызывается, чтобы получить объект определенного типа. Все встроенные типы данных языка Python могут использоваться как фабричные функции, например, тип данных `str` может вызываться как функция `str()`, которая при вызове без аргументов возвращает пустой строковый объект. Фабричная функция, передаваемая словарю со значениями по умолчанию, используется для создания значений по умолчанию для отсутствующих ключей.

Обратите внимание, что *имя* функции – это ссылка на объект функции, поэтому, когда функция передается в качестве аргумента, передается одно только имя функции. Когда вслед за именем функции записываются круглые скобки, они сообщают интерпретатору, что он должен вызвать эту функцию.

Программа *uniquewords2.py* на одну строку длиннее, чем исходная программа *uniquewords1.py* (`import collections`), а, кроме того, изменились строки создания и обновления словаря. Ниже показано, как создается словарь со значениями по умолчанию:

```
words = collections.defaultdict (int)
```

Словарь со значениями по умолчанию `words` никогда не возбудит исключение `KeyError`. Если будет необходимо выполнить выражение `x = words["xyz"]` и в словаре будет отсутствовать элемент с ключом `"xyz"`, то при обращении к несуществующему ключу словарь со значениями по умолчанию немедленно создаст новый элемент с ключом `"xyz"` и значе-

нием 0 (вызовом функции `int()`), и это значение будет присвоено переменной `x`.

```
words[word] += 1
```

Теперь мы можем отказаться от использования метода `dict.get()` и просто увеличивать значение элемента. Когда будет обнаружено самое первое вхождение слова, будет создан новый элемент со значением 0 (к которому тут же будет прибавлено число 1), а при обнаружении каждого последующего вхождения число 1 будет добавляться к текущему значению.

Мы закончили полный обзор всех встроенных типов коллекций языка Python и пары типов коллекций из стандартной библиотеки. В следующем разделе мы рассмотрим некоторые проблемы, общие для всех типов коллекций.

Обход в цикле и копирование коллекций

После того как будет создана коллекция элементов данных, вполне естественно возникает желание обойти все элементы, содержащиеся в ней. В первом подразделе этого раздела мы познакомимся с итераторами языка Python, а также с операторами и функциями, применяемыми для работы с итераторами.

Еще одна часто выполняемая операция – копирование коллекций. Из-за того, что в языке Python повсеместно используются ссылки на объекты (ради повышения эффективности), существуют некоторые особенности, связанные с копированием, поэтому во втором подразделе этого раздела мы изучим принципы копирования коллекций и узнаем, как добиться именно того, что нам нужно.

Итераторы, функции и операторы для работы с итерируемыми объектами

Итерируемый тип данных – это такой тип, который может возвращать свои элементы по одному. Любой объект, имеющий метод `__iter__()`, или любая последовательность (то есть объект, имеющий метод `__getitem__()`, принимающий целочисленный аргумент со значением от 0 и выше), является итерируемым и может предоставлять *итератор*. Итератор – это объект, имеющий метод `__next__()`, который при каждом вызове возвращает очередной элемент и возбуждает исключение `StopIteration` после исчерпания всех элементов. В табл. 3.4 перечислены операторы и функции, которые могут применяться к итерируемым объектам.

Специальный метод `__iter__()`, стр. 319

Таблица 3.4. Общие функции и операторы для работы с итерируемыми объектами

Синтаксис	Описание
`s + t`	Возвращает конкатенацию последовательностей `s` и `t`
`s * n`	Возвращает конкатенацию из `int n` последовательностей `s`
`x in i`	Возвращает `True`, если элемент `x` присутствует в итерируемом объекте `i`, обратная проверка выполняется с помощью оператора `not in`
`all(i)`	Возвращает `True`, если все элементы итерируемого объекта `i` в логическом контексте оцениваются как значение `True`
`any(i)`	Возвращает `True`, если хотя бы один элемент итерируемого объекта `i` в логическом контексте оценивается как значение `True`
`enumerate (i,start)`	Обычно используется в циклах `for ... in`, чтобы получить последовательность кортежей (*index*, *item*), где значения индексов начинают отсчитывать от 0 или от значения *start*; подробности в тексте
`len(x)`	Возвращает «длину» объекта `x`. Если `x` – коллекция, то возвращаемое число представляет количество элементов. Если `x` – строка, то возвращаемое число представляет количество символов
`max(i, key)`	Возвращает наибольший элемент в итерируемом объекте `i` или элемент с наибольшим значением *key*(*item*), если функция *key* определена
`min(i, key)`	Возвращает наименьший элемент в итерируемом объекте `i` или элемент с наименьшим значением *key*(*item*), если функция *key* определена
`range (start, stop, step)`	Возвращает целочисленный итератор. С одним аргументом (`stop`) итератор представляет последовательность целых чисел от 0 до `stop - 1`, с двумя аргументами (*start*, `stop`) – последовательность целых чисел от *start* до `stop - 1`, с тремя аргументами – последовательность целых чисел от *start* до `stop - 1` с шагом *step*
`reversed(i)`	Возвращает итератор, который будет возвращать элементы итератора `i` в обратном порядке
`sorted (i, key, reverse)`	Возвращает список элементов итератора `i` в отсортированном порядке; аргумент *key* используется для выполнения сортировки DSU (Decorate, Sort, Undecorate – декорирование, сортировка, обратное декорирование). Если аргумент *reverse* имеет значение `True`, сортировка выполняется в обратном порядке
`sum (i, start)`	Возвращает сумму элементов итерируемого объекта `i`, плюс аргумент *start* (значение которого по умолчанию равно 0); объект `i` не должен содержать строк
`zip(i1, ..., iN)`	Возвращает итератор кортежей, используя итераторы от `i1` до `i`*N*; подробности в тексте

Порядок, в котором возвращаются элементы, зависит от итерируемого объекта. В случае списков и кортежей элементы обычно возвращаются в предопределенном порядке, начиная с первого элемента (находящегося в позиции с индексом 0), но другие итераторы возвращают элементы в произвольном порядке – например, итераторы словарей и множеств.

Встроенная функция `iter()` используется двумя совершенно различными способами. Применяемая к коллекции или к последовательности, она возвращает итератор для заданного объекта или возбуждает исключение `TypeError`, если объект не является итерируемым. Такой способ часто используется при работе с нестандартными типами коллекций и крайне редко – в других контекстах. Во втором варианте использования функции `iter()` ей передается вызываемый объект (функция или метод) и специальное значение. В этом случае полученная функция или метод вызывается на каждой итерации, а значение этой функции, если оно не равно специальному значению, возвращается вызывающей программе; в противном случае возбуждается исключение `StopIteration`.

Когда в программе используется цикл `for` *`item`* `in` *`iterable`*, интерпретатор Python вызывает функцию `iter(`*`iterable`*`)`, чтобы получить итератор. После этого на каждой итерации вызывается метод `__next__()` итератора, чтобы получить очередной элемент, а когда возбуждается исключение `StopIteration`, оно перехватывается и цикл завершается. Другой способ получить очередной элемент итератора состоит в том, чтобы вызвать встроенную функцию `next()`. Ниже приводятся два эквивалентных фрагмента программного кода (оба они вычисляют произведение элементов списка), в одном из них используется цикл `for ... in`, а во втором явно используется итератор:

```
product = 1
for i in [1, 2, 4, 8]:
    product *= i
print(product) # выведет: 64
```

```
product = 1
i = iter([1, 2, 4, 8])
while True:
    try:
        product *= next(i)
    except StopIteration:
        break
print(product) # выведет: 64
```

Любой (конечный) итерируемый объект *`i`* может быть преобразован в кортеж вызовом функции `tuple(`*`i`*`)` или в список – вызовом функции `list(`*`i`*`)`.

К итераторам могут применяться функции `all()` и `any()`, и они часто используются в функциональном программировании. Ниже приводится пара примеров, демонстрирующих использование функций `all()`, `any()`, `len()`, `min()`, `max()` и `sum()`:

Функциональное программирование, стр. 397

```
>>> x = [-2, 9, 7, -4, 3]
>>> all(x), any(x), len(x), min(x), max(x), sum(x)
```

```
(True, True, 5, -4, 9, 13)
>>> x.append(0)
>>> all(x), any(x), len(x), min(x), max(x), sum(x)
(False, True, 6, -4, 9, 13)
```

Из всех этих маленьких функций наиболее часто, пожалуй, используется функция `len()`.

Функция `enumerate()` принимает итератор и возвращает объект перечисления. Этот объект может рассматриваться как своего рода итератор. На каждой итерации он возвращает кортеж из двух элементов, первый из которых – это номер итерации (по умолчанию нумерация начинается с 0), а второй – следующий элемент итератора, который был передан функции `enumerate()`. Давайте рассмотрим порядок использования функции `enumerate()` в контексте небольшой, но законченной программы.

Программа *grepword.py* принимает в виде аргументов командной строки слово и одно или более имен файлов. Она выводит имя файла, номер строки и саму строку, содержащую искомое слово.[1] Ниже приводится пример сеанса работы с программой:

```
grepword.py Dom data/forenames.txt
data/forenames.txt:615:Dominykas
data/forenames.txt:1435:Dominik
data/forenames.txt:1611:Domhnall
data/forenames.txt:3314:Dominic
```

Файлы с данными *data/forenames.txt* и *data/surnames.txt* содержат несортированные списки имен, по одному имени в каждой строке.

Не считая инструкции импортирования модуля `sys`, программа занимает всего десять строк:

```
if len(sys.argv) < 3:
    print("usage: grepword.py word infile1 [infile2 [... infileN]]")
    sys.exit()

word = sys.argv[1]
for filename in sys.argv[2:]:
    for lino, line in enumerate(open(filename), start=1):
        if word in line:
            print("{0}:{1}:{2:.40}".format(filename, lino,
                                           line.rstrip()))
```

Программа начинается с проверки наличия хотя бы двух аргументов командной строки. Если число аргументов меньше двух, программа выводит сообщение с инструкцией о порядке использования и завершает работу. Функция `sys.exit()` немедленно завершает работу программы,

[1] В главе 9 будут представлены еще две реализации этой программы – *grepword-p.py* и *grepword-t.py*, которые распределяют работу по нескольким процессам и потокам выполнения.

закрывая любые открытые файлы. Она принимает необязательный аргумент типа `int`, который передается вызывающей командной оболочке.

Предполагается, что первым аргументом является слово, которое требуется отыскать, а другие аргументы – это имена файлов, в которых требуется произвести поиск. Мы преднамеренно вызываем функцию `open()`, не указывая кодировку символов – пользователь может использовать в именах файлов шаблонные символы для выбора группы файлов, каждый из которых может иметь собственную кодировку символов, поэтому в данном случае мы оставляем за интерпретатором право использовать платформозависимую кодировку.

Врезка «Чтение и запись текстовых файлов», стр. 157

Функция `open()` возвращает объект файла, открытого в текстовом режиме, в котором этот объект может использоваться как итератор, возвращая по одной строке из файла в каждой итерации. Передав итератор функции `enumerate()`, мы получаем итератор-перечисление, который в каждой итерации возвращает номер итерации (в переменной `lino`, «line number» – «номер строки») и строку из файла. Если слово, указанное пользователем, присутствует в строке, программа выводит имя файла, номер строки и первые 40 символов этой строки, из которой удаляются завершающие пробельные символы (такие как `\n`). Функция `enumerate()` принимает необязательный именованный аргумент `start`, который по умолчанию имеет значение 0. Мы передаем в этом аргументе значение 1, так как, в соответствии с общепринятыми соглашениями, нумерация строк в текстовых файлах начинается с 1.

Как правило, на практике используется не итератор-перечисление, а итератор, возвращающий последовательные целые числа. Это именно то, что делает функция `range()`. Если нам необходим кортеж или список целых чисел, мы можем преобразовать итератор, возвращаемый функцией `range()`, воспользовавшись соответствующей функцией преобразования, как показано ниже:

```
>>> list(range(5)), list(range(9, 14)), tuple(range(10, -11, -5))
([0, 1, 2, 3, 4], [9, 10, 11, 12, 13], (10, 5, 0, -5, -10))
```

Функция `range()` обычно используется в двух случаях: для создания списков или кортежей целых чисел и в качестве счетчика в циклах `for ... in`. Например, следующие два эквивалентных примера преобразуют значения в списке в положительные числа:

```
for i in range(len(x)):
    x[i] = abs(x[i])
```

```
i = 0
while i < len(x):
    x[i] = abs(x[i])
    i += 1
```

В обоих случаях, если предположить, что `x` – это список значений `[11, -3, -12, 8, -1]`, то после выполнения фрагментов он превратится в список `[11, 3, 12, 8, 1]`.

Благодаря тому, что существует возможность распаковывать итерируемые объекты с помощью оператора *, мы можем распаковать итератор, возвращаемый функцией range(). Например, если представить, что у нас имеется функция calculate(), которая принимает четыре аргумента, ниже приводятся несколько способов вызова этой функции с аргументами 1, 2, 3 и 4:

```
calculate(1, 2, 3, 4)
t = (1, 2, 3, 4)
calculate(*t)
calculate(*range(1, 5))
```

Во всех трех случаях функции передается четыре аргумента. Во втором случае распаковывается кортеж из четырех элементов, а в третьем – распаковывается итератор, возвращаемый функцией range().

Теперь рассмотрим небольшую, но законченную программу, которая соединяет в себе все, что мы узнали к настоящему моменту, и впервые явно использует функцию записи в файл. Программа *generate_test_names1.py* читает данные из файла с фамилиями и из файла именами, создает два списка, а затем записывает 100 случайно образованных имен в файл *test-names1.txt*.

В программе используется функция random.choice(), которая извлекает случайный элемент из последовательности, поэтому вполне возможно, что в окончательном списке одно и то же имя может появиться несколько раз. Для начала рассмотрим функцию, возвращающую список имен, а затем перейдем к остальной части программы:

```
def get_forenames_and_surnames():
    forenames = []
    surnames = []
    for names, filename in ((forenames, "data/forenames.txt"),
                            (surnames, "data/surnames.txt")):
        for name in open(filename, encoding="utf8"):
            names.append(name.rstrip())
    return forenames, surnames
```

Распаковывание кортежей, стр. 133

Во внешнем цикле for ... in выполняется обход двухэлементных кортежей, каждый из которых распаковывается в две переменные. Хотя списки могут быть чрезвычайно длинными, возврат их из функции выполняется очень эффективно, так как в языке Python используются ссылки на объекты, поэтому фактически функция возвращает всего лишь две ссылки на объекты.

Внутри программ на языке Python всегда следует использовать запись путей к файлам в стиле операционной системы UNIX, поскольку в этом случае можно не применять экранирование служебных символов и такой прием одинаково хорошо работает на всех поддерживаемых платформах (включая и Windows). Если у нас имеется строка пути, например в переменной path, и нам необходимо вывести ее перед

пользователем, мы всегда можем импортировать модуль `os` и вызвать метод `path.replace("\", os.sep)` для замены прямых слешей на символ-разделитель каталогов, используемый в текущей платформе.

```
forenames, surnames = get_forenames_and_surnames()
fh = open("test-names1.txt", "w", encoding="utf8")
for i in range(100):
    line = "{0} {1}\n".format(random.choice(forenames),
                              random.choice(surnames))
    fh.write(line)
```

Получив два списка, программа открывает выходной файл для записи и сохраняет объект файла в переменной `fh` («file handle» – дескриптор файла). После этого выполняется 100 циклов и на каждой итерации создается строка, в конец которой добавляется символ перевода строки, и эта строка записывается в файл. Мы не используем переменную цикла `i` – она нужна исключительно для того, чтобы удовлетворить требования синтаксиса цикла `for ... in`. Предыдущий фрагмент программного кода, функция `get_forenames_and_surnames()` и инструкция `import` образуют полную программу.

Врезка «Чтение и запись текстовых файлов», стр. 157

В программе *generate_test_names1.py* мы объединяли элементы из двух отдельных списков в единую строку. Другой способ объединения элементов двух или более списков (или других итерируемых объектов) заключается в использовании функции `zip()`. Функция `zip()` принимает один или более итерируемых объектов и возвращает итератор, который в свою очередь возвращает кортежи. Первый кортеж включает в себя первые элементы всех итерируемых объектов, второй кортеж – вторые элементы и т. д., итерации прекращаются, как только содержимое любого из итерируемых объектов будет исчерпано. Например:

```
>>> for t in zip(range(4), range(0, 10, 2), range(1, 10, 2)):
...     print(t)
(0, 0, 1)
(1, 2, 3)
(2, 4, 5)
(3, 6, 7)
```

Несмотря на то, что итераторы, возвращаемые вторым и третьим вызовами функции `range()`, могут вернуть по пять элементов, тем не менее первый итератор может воспроизвести всего четыре элемента, тем самым ограничивая количество элементов, которые может вернуть функция `zip()`.

Ниже приводится модифицированная версия программы, генерирующей имена, но на этот раз под имя отводится 25 символов, а вслед за каждым именем выводится случайный год. Программа называется *generate_test_names2.py* и выводит результаты в файл *test-names2.txt*. Мы не приводим здесь программный код функции `get_forenames_and_`

surnames() и вызов функции open(), так как они не изменились, за исключением имени выходного файла.

```
limit = 100
years = list(range(1970, 2013)) * 3
for year, forename, surname in zip(
        random.sample(years, limit),
        random.sample(forenames, limit),
        random.sample(surnames, limit)):
    name = "{0} {1}".format(forename, surname)
    fh.write("{0:.<25}.{1}\n".format(name, year))
```

Программа начинается с определения значения максимального числа имен, которые могут быть сгенерированы. Затем создается список лет – путем создания списка со значениями в диапазоне от 1970 до 2012 включительно, после чего этот список дублируется трижды, поэтому в окончательном списке каждый год встречается три раза. Это необходимо потому, что функция random.sample() (используемая вместо random.choice()) принимает итерируемый объект и число элементов, которые требуется воспроизвести – это число не может быть меньше, чем число элементов, которые может вернуть итерируемый объект. Функция random.sample() возвращает итератор, который воспроизводит указанное число элементов без повторений. Поэтому данная версия программы всегда будет воспроизводить уникальные имена.

Распаковывание кортежей, стр. 133

В цикле for ... in распаковывается каждый кортеж, возвращаемый функцией zip(). Нам требуется ограничить длину каждого имени 25 символами, а для этого сначала нужно создать строку с полным именем, а затем вторым вызовом метода str.format() ограничить ее длину. Каждое имя выравнивается по левому краю, а для имен короче 25 символов производится дополнение строки точками. Дополнительная точка гарантирует, что имена, полностью занимающие поле вывода, все же будут отделяться от года хотя бы одной точкой.

Метод str.format(), стр. 100

В завершение этого подраздела мы упомянем еще две функции, имеющие отношение к итерируемым объектам – sorted() и reversed(). Функция sorted() возвращает отсортированный список элементов, а функция reversed() просто возвращает итератор, который позволяет выполнить обход элементов заданного итератора в обратном порядке. Ниже приводится пример использования функции reversed():

```
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
>>> list(reversed(range(6)))
[5, 4, 3, 2, 1, 0]
```

Функция sorted() – более сложная, как показано в примере ниже:

```
>>> x = []
>>> for t in zip(range(-10, 0, 1), range(0, 10, 2), range(1, 10, 2)):
```

```
...     x += t
>>> x
[-10, 0, 1, -9, 2, 3, -8, 4, 5, -7, 6, 7, -6, 8, 9]
>>> sorted(x)
[-10, -9, -8, -7, -6, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sorted(x, reverse=True)
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -6, -7, -8, -9, -10]
>>> sorted(x, key=abs)
[0, 1, 2, 3, 4, 5, 6, -6, -7, 7, -8, 8, -9, 9, -10]
```

В предыдущем фрагменте функция `zip()` возвращает кортежи, состоящие из трех элементов: `(-10, 0, 1)`, `(-9, 2, 3)` и т. д. Оператор `+=` дополняет список, то есть добавляет каждый элемент заданной последовательности в конец списка.

Первый вызов функции `sorted()` возвращает копию списка, отсортированную в привычном порядке. Второй вызов возвращает копию списка, отсортированную в обратном порядке. В последнем вызове функции `sorted()` определена функция «key», к которой мы вернемся через несколько мгновений.

Обратите внимание, что в языке Python функции являются самыми обычными объектами, поэтому они могут передаваться другим функциям в виде аргументов и сохраняться в коллекциях без излишних формальностей. Не забывайте, что имя функции – это ссылка на объект функции, а круглые скобки, следующие за именем, сообщают интерпретатору Python о необходимости вызова этой функции.

Когда функции `sorted()` передается функция `key` (в данном случае – функция `abs()`), она будет вызываться для каждого элемента списка (каждый элемент будет передаваться функции в виде единственного аргумента), чтобы создать «декорированный» список. Затем выполняется сортировка декорированного списка, после чего в качестве результата возвращается недекорированный список. Мы легко можем использовать собственные функции в качестве аргумента `key`, в чем вы вскоре убедитесь.

Например, мы можем выполнить сортировку без учета регистра символов, передав в аргументе `key` метод `str.lower()`. Если представить, что у нас имеется список `["Sloop", "Yawl", "Cutter", "schooner", "ketch"]`, мы можем отсортировать его без учета регистра символов, используя DSU-сортировку (Decorate, Sort, Undecorate – декорирование, сортировка, обратное декорирование), всего одной строкой программного кода, передав нужную функцию в аргументе `key` или выполнив сортировку явно, как показано в следующих двух эквивалентных фрагментах программного кода:

```
temp = []
for item in x:
    temp.append((item.lower(), item))
```

```
x = sorted(x, key=str.lower)
```

```
x = []
for key, value in sorted(temp):
    x.append(value)
```

Оба фрагмента воспроизводят новый список: `["Cutter", "ketch", "schooner", "Sloop", "Yawl"]`, хотя действия, которые они выполняют, не идентичны, потому что во фрагменте справа создается промежуточный список `temp`.

В языке Python реализован адаптивный алгоритм устойчивой сортировки со слиянием, который отличается высокой скоростью и интеллектуальностью и особенно хорошо подходит для сортировки частично отсортированных списков, что встречается достаточно часто.[1] Слово «адаптивный» в названии алгоритма означает, что алгоритм сортировки адаптируется под определенные условия, например, учитывает наличие частичной сортировки данных. Слово «устойчивый» означает, что одинаковые элементы не перемещаются относительно друг друга (в конце концов, в этом нет никакой необходимости), и слова «сортировка со слиянием» – это общее название используемых алгоритмов сортировки. Когда выполняется сортировка списка целых чисел, строк или данных других простых типов, используется оператор «меньше чем» (`<`). Интерпретатор Python может сортировать коллекции, содержащие другие коллекции, выполняя рекурсивный спуск на произвольную глубину. Например:

```
>>> x = list(zip((1, 3, 1, 3), ("pram", "dorie", "kayak", "canoe")))
>>> x
[(1, 'pram'), (3, 'dorie'), (1, 'kayak'), (3, 'canoe')]
>>> sorted(x)
[(1, 'kayak'), (1, 'pram'), (3, 'canoe'), (3, 'dorie')]
```

Python отсортировал список кортежей, сравнив сначала первые элементы каждого кортежа, а если они были равны – вторые элементы. В результате элементы были отсортированы на основе целых чисел, с использованием строк для дополнительной сортировки. Мы можем принудительно сначала отсортировать список по строкам, а дополнительную сортировку выполнить по целым числам, определив простую функцию, которая будет использоваться в качестве аргумента `key`:

```
def swap(t):
    return t[1], t[0]
```

Функция `swap()` принимает кортеж из двух элементов и возвращает новый кортеж из двух элементов, в котором элементы переставлены местами. Представим, что функцию `swap()` мы ввели в среде IDLE, тогда можно выполнить следующее:

---

[1] Алгоритм был создан Тимом Петерсом (Tim Peters). Интересное описание и обсуждение алгоритма можно найти в файле *listsort.txt*, который поставляется в составе исходных программных кодов Python.

```
>>> sorted(x, key=swap)
[(3, 'canoe'), (3, 'dorie'), (1, 'kayak'), (1, 'pram')]
```

Кроме того, списки могут быть отсортированы непосредственно, без создания копии, с помощью метода `list.sort()`, который принимает те же необязательные аргументы, что и функция `sorted()`.

Сортировка может применяться только к коллекциям, все элементы которых могут сравниваться друг с другом:

```
sorted([3, 8, -7.5, 0, 1.3])          # вернет: [-7.5, 0, 1.3, 3, 8]
sorted([3, "spanner", -7.5, 0, 1.3]) # возбудит исключение TypeError
```

Хотя первый список содержит числа разных типов (`int` и `float`), тем не менее эти типы могут сравниваться друг с другом, поэтому сортировка к такому списку вполне применима. Но во втором списке содержится строка, которую не имеет смысла сравнивать с числами, поэтому возбуждается исключение `TypeError`. Если необходимо отсортировать список, содержащий целые числа, числа с плавающей точкой и строки, представляющие числа, можно попробовать передать в аргументе `key` функцию `float()`:

```
sorted(["1.3", -7.5, "5", 4, "-2.4", 1], key=float)
```

Это выражение вернет список `[-7.3, '-2.4', 1, '1.3', 4, '5']`. Обратите внимание, что значения в списке не изменились, то есть строки так и остались строками. Если какая-либо строка не сможет быть преобразована в число (например, `"spanner"`), будет возбуждено исключение `TypeError`.

Копирование коллекций

Ссылки на объекты, стр. 29

Поскольку в языке Python повсюду используются ссылки на объекты, когда выполняется оператор присваивания (`=`), никакого копирования данных на самом деле не происходит. Если справа от оператора находится литерал, например, строка или число, в операнд слева записывается ссылка, которая указывает на объект в памяти, хранящий значение литерала. Если справа находится ссылка на объект, в левый операнд записывается ссылка, указывающая на тот же самый объект, на который ссылается правый операнд. Вследствие этого операция присваивания обладает чрезвычайно высокой скоростью выполнения.

Когда выполняется присваивание крупной коллекции, такой как длинный список, экономия времени становится более чем очевидной. Например:

```
>>> songs = ["Because", "Boys", "Carol"]
>>> beatles = songs
```

```
>>> beatles, songs
(['Because', 'Boys', 'Carol'], ['Because', 'Boys', 'Carol'])
```

Здесь была создана новая ссылка на объект (`beatles`), и обе ссылки указывают на один и тот же список – никакого копирования данных не производилось.

Поскольку списки относятся к категории изменяемых объектов, мы можем вносить в них изменения. Например:

```
>>> beatles[2] = "Cayenne"
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Cayenne'])
```

Изменения были внесены с использованием переменной `beatles`, но это всего лишь ссылка, указывающая на тот же самый объект, что и ссылка `songs`. Поэтому любые изменения, произведенные с использованием одной ссылки, можно наблюдать с использованием другой ссылки. Часто это именно то, что нам требуется, поскольку копирование крупных коллекций может оказаться дорогостоящей операцией. Кроме того, это также означает, что имеется возможность передавать списки или другие изменяемые коллекции в виде аргументов функций, изменять эти коллекции в функциях и пребывать в уверенности, что изменения будут доступны после того, как функция вернет управление вызывающей программе.

Однако в некоторых ситуациях действительно бывает необходимо создать отдельную копию коллекции (то есть создать другой изменяемый объект). В случае последовательностей, когда выполняется оператор извлечения среза, например, `songs[:2]`, полученный срез – это всегда независимая копия элементов. Поэтому скопировать последовательность целиком можно следующим способом:

```
>>> songs = ["Because", "Boys", "Carol"]
>>> beatles = songs[:]
>>> beatles[2] = "Cayenne"
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Carol'])
```

В случае словарей и множеств копирование можно выполнить с помощью методов `dict.copy()` и `set.copy()`. Кроме того, в модуле `copy` имеется функция `copy.copy()`, которая возвращает копию заданного объекта. Другой способ копирования встроенных типов коллекций заключается в использовании имени типа как функции, которой в качестве аргумента передается копируемая коллекция. Например:

```
copy_of_dict_d = dict(d)
copy_of_list_L = list(L)
copy_of_set_s = set(s)
```

Обратите внимание, что все эти приемы копирования создают *поверхностные* копии, то есть копируются только ссылки на объекты, но не

сами объекты. Для неизменяемых типов данных, таких как числа и строки, это равносильно копированию (за исключением более высокой эффективности), но для изменяемых типов данных, таких как вложенные коллекции, это означает, что ссылки в оригинальной коллекции и в копии будут указывать на одни и те же объекты. Эту особенность иллюстрирует следующий пример:

```
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = x[:] # поверхностное копирование
>>> x, y
([53, 68, ['A', 'B', 'C']], [53, 68, ['A', 'B', 'C']])
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['Q', 'B', 'C']])
```

Когда выполняется поверхностное копирование списка `x`, копируется ссылка на вложенный список `["A", "B", "C"]`. Это означает, что третий элемент в обоих списках, `x` и `y`, ссылается на один и тот же список, поэтому любые изменения, произведенные во вложенном списке, можно наблюдать с помощью любой из ссылок, `x` или `y`. Если действительно необходимо создать абсолютно независимую копию коллекции с произвольной глубиной вложенности, необходимо выполнить глубокое копирование:

```
>>> import copy
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = copy.deepcopy(x)
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['A', 'B', 'C']])
```

Здесь списки `x` и `y`, а также элементы, которые они содержат, полностью независимы.

Обратите внимание: с этого момента мы будем использовать термины *копия* и *поверхностная копия* как взаимозаменяемые, а когда будет подразумеваться *глубокое копирование*, об этом будет упоминаться явно.

Примеры

Мы завершили обзор встроенных типов коллекций в языке Python, а также двух типов, реализованных в стандартной библиотеке (`collections.namedtuple` и `collections.defaultdict`). В языке Python имеется также тип коллекций `collections.deque`, двухсторонней очереди, и многие другие типы коллекций, реализованные сторонними разработчиками и доступные в каталоге пакетов Python Package Index, *pypi.python.org/pypi*. А сейчас мы рассмотрим пару немного более длинных примеров,

в которых используется многое из того, о чем рассказывалось в этой и в предыдущей главах.

Первая программа насчитывает примерно семьдесят строк и имеет отношение к обработке текстовой информации. Вторая программа содержит примерно девяносто строк и предназначена для выполнения математических вычислений. Обе программы используют словари, списки, именованные кортежи и множества и обе широко используют метод `str.format()`, который был описан в предыдущей главе.

generate_usernames.py

Представьте, что мы выполняем настройку новой компьютерной системы и нам необходимо сгенерировать имена пользователей для всех служащих нашей компании. У нас имеется простой текстовый файл (в кодировке UTF-8), где каждая строка представляет собой запись из полей, разделенных двоеточиями. Каждая запись соответствует одному сотруднику компании и содержит следующие поля: уникальный идентификатор служащего, имя, отчество (это поле может быть пустым), фамилию и название отдела. Ниже в качестве примера приводятся несколько строк из файла *data/users.txt*:

```
1601:Albert:Lukas:Montgomery:Legal
3702:Albert:Lukas:Montgomery:Sales
4730:Nadelle::Landale:Warehousing
```

Программа должна читать данные из файла, который указан в командной строке, извлекать отдельные поля из каждой строки (записи) и возвращать подходящие имена пользователей. Каждое имя пользователя должно быть уникальным и должно создаваться на основе имени сотрудника. Результаты должны выводиться на консоль в текстовом виде, отсортированными в алфавитном порядке по фамилии и имени, например:

```
Name                              ID   Username
------------------------------- ------ ---------
Landale, Nadelle................ (4730) nlandale
Montgomery, Albert L............ (1601) almontgo
Montgomery, Albert L............ (3702) almontgo1
```

Каждая запись имеет точно пять полей, и хотя можно обращаться к ним по числовым индексам, тем не менее мы будем использовать осмысленные имена, чтобы сделать программный код более понятным:

```
ID, FORENAME, MIDDLENAME, SURNAME, DEPARTMENT = range(5)
```

В языке Python общепринято использовать только символы верхнего регистра для идентификаторов, которые будут играть роль констант.

Нам также необходимо создать тип именованного кортежа, где будут храниться данные о текущем пользователе:

```
User = collections.namedtuple("User",
            "username forename middlename surname id")
```

Позднее, когда мы будем рассматривать оставшуюся часть программы, мы увидим, как используется именованный кортеж `User` и константы.

Основная логика программы сосредоточена в функции `main()`:

```
def main():
    if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
        print("usage: {0} file1 [file2 [... fileN]]".format(
              sys.argv[0]))
        sys.exit()

    usernames = set()
    users = {}
    for filename in sys.argv[1:]:
        for line in open(filename, encoding="utf8"):
            line = line.rstrip()
            if line:
                user = process_line(line, usernames)
                users[(user.surname.lower(), user.forename.lower(),
                       user.id)] = user
    print_users(users)
```

Если пользователь не ввел в командной строке имя какого-нибудь файла или ввел параметр «-h» или «--help», то программа просто выводит текст сообщения с инструкцией о порядке использования и завершает работу.

Из каждой прочитанной строки удаляются любые завершающие пробельные символы (такие как `\n`), и обработка строки продолжается, только если она не пустая. Это означает, что если в данных содержатся пустые строки, они будут просто проигнорированы.

Все сгенерированные имена пользователей сохраняются в множестве `usernames`, чтобы гарантировать отсутствие повторяющихся имен пользователей. Сами данные сохраняются в словаре `users`. Информация о каждом пользователе сохраняется в виде элемента словаря, ключом которого является кортеж, содержащий фамилию сотрудника, его имя и идентификатор, а значением – именованный кортеж типа `User`. Использование кортежа, содержащего фамилию сотрудника, его имя и идентификатор, в качестве ключа обеспечивает возможность вызывать функцию `sorted()` для словаря и получать итерируемый объект, в котором элементы будут упорядочены в требуемом нам порядке (то есть фамилия, имя, идентификатор), избежав необходимости создавать функцию, которую пришлось бы передавать в качестве аргумента `key`.

```
def process_line(line, usernames):
    fields = line.split(":")
    username = generate_username(fields, usernames)
    user = User(username, fields[FORENAME], fields[MIDDLENAME],
```

```
                fields[SURNAME], fields[ID])
    return user
```

Поскольку все записи имеют очень простой формат, и мы уже удалили из строки завершающие пробельные символы, извлечь отдельные поля можно простой разбивкой строки по двоеточиям. Мы передаем список полей и множество `usernames` в функцию `generate_username()` и затем создаем экземпляр именованного кортежа `User`, который возвращается вызывающей программе (функции `main()`), которая в свою очередь вставляет информацию о пользователе в словарь `users`, готовый для вывода на экран.

Если бы мы не создали соответствующие константы для хранения индексов, мы могли бы использовать числовые индексы, как показано ниже:

```
user = User(username, fields[1], fields[2], fields[3], fields[0])
```

Хотя такой программный код занимает меньше места, тем не менее это не самое лучшее решение. Во-первых, человеку, который будет сопровождать такой программный код, непонятно, какое поле какую информацию содержит, а, во-вторых, такой программный код чувствителен к изменениям в формате файла с данными – если изменится порядок или число полей в записи, этот программный код окажется неработоспособен. При использовании констант в случае изменения структуры записи нам достаточно будет изменить только значения констант, и программа сохранит свою работоспособность.

```
def generate_username(fields, usernames):
    username = ((fields[FORENAME][0] + fields[MIDDLENAME][:1] +
                 fields[SURNAME]).replace("-", "").replace("'", ""))
    username = original_name = username[:8].lower()
    count = 1
    while username in usernames:
        username = "{0}{1}".format(original_name, count)
        count += 1
    usernames.add(username)
    return username
```

При первой попытке имя пользователя создается путем конкатенации первого символа имени, первого символа отчества и фамилии целиком, после чего из полученной строки удаляются дефисы и апострофы. Выражение, извлекающее первый символ отчества, таит в себе одну хитрость. Если просто использовать обращение `fields[MIDDLENAME][0]`, то в случае отсутствия отчества будет возбуждено исключение `IndexError`. Но при использовании операции извлечения среза мы получаем либо первый символ отчества, либо пустую строку.

Затем мы переводим все символы полученного имени пользователя в нижний регистр и ограничиваем его длину восемью символами. Если имя пользователя уже занято (то есть оно уже присутствует в множе-

стве usernames), предпринимается попытка добавить в конец имени пользователя символ «1», если это имя пользователя тоже занято, тогда предпринимается попытка добавить символ «2» и т. д., пока не будет получено незанятое имя пользователя. После этого имя пользователя добавляется в множество usernames и возвращается вызывающей программе.

```
def print_users(users):
    namewidth = 32
    usernamewidth = 9

    print("{0:<{nw}} {1:^6} {2:{uw}}".format(
          "Name", "ID", "Username", nw=namewidth, uw=usernamewidth))
    print("{0:-<{nw}} {0:-<6} {0:-<{uw}}".format(
          "", nw=namewidth, uw=usernamewidth))

    for key in sorted(users):
        user = users[key]
        initial = ""
        if user.middlename:
            initial = " " + user.middlename[0]
        name = "{0.surname}, {0.forename}{1}".format(user, initial)
        print("{0:.<{nw}} ({1.id:4}) {1.username:{uw}}".format(
              name, user, nw=namewidth, uw=usernamewidth))
```

После обработки всех записей вызывается функция print_users(), которой в качестве параметра передается словарь users.

Первая инструкция print() выводит заголовки столбцов. Вторая инструкция print() выводит дефисы под каждым из заголовков. В этой второй инструкции метод str.format() используется довольно оригинальным образом. Для вывода ему определяется строка "", то есть пустая строка; в результате при выводе пустой строки мы получаем строку из дефисов заданной ширины поля вывода.

Метод str.format(), стр. 100

Затем мы используем цикл for ... in для вывода информации о каждом пользователе, извлекая ключи из отсортированного словаря. Для удобства мы создаем переменную user, чтобы не вводить каждый раз users[key] в оставшейся части функции. В цикле сначала вызывается метод str.format(), чтобы записать в переменную name фамилию сотрудника, имя и необязательный первый символ отчества. Обращение к элементам в именованном кортеже user производится по их именам. Собрав строку с именем пользователя, мы выводим информацию о пользователе, ограничивая ширину каждого столбца (имя сотрудника, идентификатор и имя пользователя) желаемыми значениями.

Полный программный код программы (который несколько отличается от того, что мы только что рассмотрели, несколькими начальными строками с комментариями и инструкциями импорта) находится в файле *generate_usernames.py*. Структура программы – чтение данных из

файла, обработка каждой записи, вывод результата – одна из наиболее часто встречающихся, и она повторяется в следующем примере.

statistics.py

Предположим, что у нас имеется пакет файлов с данными, содержащих числовые результаты некоторой обработки, выполненной нами, и нам необходимо вычислить некоторые основные статистические характеристики, которые дадут нам возможность составить общую картину о полученных данных. В каждом файле находится обычный текст (в кодировке ASCII), с одним или более числами в каждой строке (разделенными пробельными символами).

Ниже приводится пример информации, которую нам необходимо получить:

```
count     =    183
mean      =    130.56
median    =     43.00
mode      = [5.00, 7.00, 50.00]
std. dev. =    235.01
```

Здесь видно, что было прочитано 183 числа, из которых наиболее часто встречаются числа 5, 7 и 50, и со стандартным отклонением по выборке 235.01.

Сами статистические характеристики хранятся в именованном кортеже `Statistics`:

```
Statistics = collections.namedtuple("Statistics",
                                    "mean mode median std_dev")
```

Функция `main()` может служить схематическим отображением структуры программы:

```
def main():
    if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
        print("usage: {0} file1 [file2 [... fileN]]".format(
              sys.argv[0]))
        sys.exit()

    numbers = []
    frequencies = collections.defaultdict(int)
    for filename in sys.argv[1:]:
        read_data(filename, numbers, frequencies)
    if numbers:
        statistics = calculate_statistics(numbers, frequencies)
        print_results(len(numbers), statistics)
    else:
        print("no numbers found")
```

Все числа из всех файлов сохраняются в списке `numbers`. Для нахождения модальных («наиболее часто встречающихся») значений нам необ-

ходимо знать, сколько раз встречается каждое число, поэтому мы создаем словарь со значениями по умолчанию, используя функцию int() в качестве фабричной функции, где будут накапливаться счетчики.

Затем выполняется обход списка файлов и производится чтение данных из них. В качестве дополнительных аргументов мы передаем функции read_data() список и словарь со значениями по умолчанию, чтобы она могла обновлять их. После чтения всех данных мы исходим из предположения, что некоторые числа были благополучно прочитаны, и вызываем функцию calculate_statistics(). Она возвращает именованный кортеж типа Statistics, который затем используется для вывода результатов.

```
def read_data(filename, numbers, frequencies):
    for lino, line in enumerate(open(filename, encoding="ascii"),
                                start=1):
        for x in line.split():
            try:
                number = float(x)
                numbers.append(number)
                frequencies[number] += 1
            except ValueError as err:
                print("{0}:{1}: skipping {2}: {3}".format(
                      filename, lino, x, err))
```

Каждая строка разбивается по пробельным символам, после чего производится попытка преобразовать каждый элемент в число типа float. Если преобразование удалось, следовательно, это либо целое число, либо число с плавающей точкой в десятичной или в экспоненциальной форме. Полученное число добавляется в список numbers и выполняется обновление словаря frequencies со значениями по умолчанию. (Если бы здесь использовался обычный словарь dict, программный код, выполняющий обновление словаря, мог бы выглядеть так: frequencies[number] = frequencies.get(number, 0) + 1.) Если преобразование потерпело неудачу, выводится номер строки (счет строк в текстовых файлах по традиции начинается с 1), текст, который программа пыталась преобразовать в число, и сообщение об ошибке, соответствующее исключению ValueError.

```
def calculate_statistics(numbers, frequencies):
    mean = sum(numbers) / len(numbers)
    mode = calculate_mode(frequencies, 3)
    median = calculate_median(numbers)
    std_dev = calculate_std_dev(numbers, mean)
    return Statistics(mean, mode, median, std_dev)
```

Эта функция используется для сбора всех статистических характеристик воедино. Поскольку среднее значение вычисляется очень просто, мы делаем это прямо здесь. Вычислением других статистических характеристик занимаются отдельные функции, и в заключение данная

функция возвращает экземпляр именованного кортежа Statistics, содержащий четыре вычисленные статистические характеристики.

```
def calculate_mode(frequencies, maximum_modes):
    highest_frequency = max(frequencies.values())
    mode = [number for number, frequency in frequencies.items()
            if math.fabs(frequency - highest_frequency) <=
                sys.float_info.epsilon]
    if not (1 <= len(mode) <= maximum_modes):
        mode = None
    else:
        mode.sort()
    return mode
```

В выборке может существовать сразу несколько значений, встречающихся наиболее часто, поэтому, помимо словаря frequencies, функции передается максимально допустимое число модальных значений. (Эта функция вызывается из calculate_statistics(), и при вызове задается максимальное число модальных значений, равное трем.)

Функция max() используется для поиска наибольшего значения в словаре frequencies. Затем с помощью генератора списков создается список из значений, которые равны наивысшему значению. Поскольку числа могут быть с плавающей точкой, мы сравниваем абсолютное значение разницы (используя функцию math.fabs(), поскольку она лучше подходит для случаев сравнения малых величин, близких к порогу точности представления числовых значений в компьютере, чем abs()) с наименьшим значением, которое может быть представлено компьютером.

Если число модальных значений равно 0 или больше максимального, то в качестве модального значения возвращается None; в противном случае возвращается сортированный список модальных значений.

```
def calculate_median(numbers):
    numbers = sorted(numbers)
    middle = len(numbers) // 2
    median = numbers[middle]
    if len(numbers) % 2 == 0:
        median = (median + numbers[middle - 1]) / 2
    return median
```

Медиана («среднее значение») – это значение, находящееся в середине упорядоченной выборки чисел, за исключением случая, когда в выборке присутствует четное число чисел, – тогда значение медианы определяется как среднее арифметическое значение двух чисел, находящихся в середине.

Функция вычисления медианы сначала выполняет сортировку чисел по возрастанию. Затем посредством целочисленного деления определяется позиция середины выборки, откуда извлекается число и сохраняется как значение медианы. Если выборка содержит четное число

значений, то значение медианы определяется как среднее арифметическое двух чисел, находящихся в середине.

```
def calculate_std_dev(numbers, mean):
    total = 0
    for number in numbers:
        total += ((number - mean) ** 2)
    variance = total / (len(numbers) - 1)
    return math.sqrt(variance)
```

Стандартное отклонение – это мера дисперсии; оно определяет, как сильно отклоняются значения в выборке от среднего значения. Вычисление стандартного отклонения в этой функции выполняется по формуле

$$\frac{\sqrt{\Sigma(x+\bar{x})^2}}{n-1},$$

где x – очередное число, $\bar{x}$ – среднее значение, а n – количество чисел.

```
def print_results(count, statistics):
    real = "9.2f"

    if statistics.mode is None:
        modeline = ""
    elif len(statistics.mode) == 1:
        modeline = "mode = {0:{fmt}}\n".format(
                statistics.mode[0], fmt=real)
    else:
        modeline = ("mode = [" +
                    ", ".join(["{0:.2f}".format(m)
                    for m in statistics.mode]) + "]\n")
    print("""\
count = {0:6}
mean = {1.mean:{fmt}}
median = {1.median:{fmt}}
{2}\
std. dev. = {1.std_dev:{fmt}}""".format(
    count, statistics, modeline, fmt=real))
```

Большая часть этой функции связана с форматированием списка модальных значений в строку `modeline`. Если модальные значения отсутствуют, то строка `modeline` вообще не выводится. Если модальное значение единственное, список модальных значений содержит единственный элемент (`mode[0]`), который и выводится с той же строкой форматирования, что используется при выводе других статистических значений. Если имеется несколько модальных значений, они выводятся как список, в котором каждое значение форматируется отдельно. Делается это с помощью генератора списков, который позво-

Метод `str.format()`, стр. 100

ляет получить список строк с модальными значениями; строки затем объединяются в единую строку, где отделяются друг от друга запятой с пробелом (", "). Последняя инструкция `print()` в самом конце получилась очень простой благодаря использованию именованного кортежа. Он позволяет обращаться к статистическим значениям в объекте `statistics`, используя не числовые индексы, а их имена, а благодаря строкам в тройных кавычках мы смогли отформатировать выводимый текст наглядным способом.

В этой функции имеется одна особенность, о которой следует упомянуть отдельно. Строка с модальными значениями выводится с помощью элемента строки формата `{2}`, за которым следует символ обратного слеша. Символ обратного слеша экранирует символ перевода строки, поэтому если строка с модальными значениями пустая, то пустая строка выводиться не будет. Именно по этой причине мы вынуждены были добавить символ `\n` в конец строки `modeline`, если она не пустая.

В заключение

В этой главе мы рассмотрели все встроенные типы коллекций в языке Python, а также пару типов коллекций из стандартной библиотеки. Мы рассмотрели коллекции-последовательности, `tuple`, `collections.namedtuple` и `list`, поддерживающие, как и строки, возможность извлечения срезов. Также было рассмотрено использование оператора распаковывания последовательностей (`*`) и коротко было упомянуто использование аргументов со звездочками в вызовах функций. Мы также рассмотрели типы множеств `set` и `frozenset` и типы отображений `dict` и `collections.defaultdict`.

Мы узнали, как использовать именованные кортежи из стандартной библиотеки языка Python для создания своих собственных типов кортежей, доступ к элементам которых выполняется не только с помощью числовых индексов, но и более удобным способом – с помощью имен. Мы также увидели, как создавать «константы», используя для этого переменные, идентификаторы которых состоят исключительно из символов верхнего регистра.

При изучении списков мы увидели, что все, что применимо к кортежам, в равной степени применимо и к спискам. А благодаря тому, что списки относятся к категории изменяемых объектов, они обладают гораздо более широкими функциональными возможностями, чем кортежи. В число этих возможностей входят методы, изменяющие содержимое списка (например, `list.pop()`), а поддержка операций со срезами обеспечивает возможность вставки, замены и удаления срезов. Списки идеально подходят для хранения последовательностей элементов, особенно, когда необходим быстрый доступ к элементам по их индексам.

При обсуждении типов `set` и `frozenset` мы отметили, что они могут содержать только элементы хешируемых типов данных. Множества обеспечивают быструю работу оператора проверки на вхождение и удобны для фильтрации повторяющихся данных.

Словари отчасти напоминают множества, например, ключами словарей могут быть только уникальные значения хешируемых типов данных, как и элементы множеств. Но, в отличие от множеств, словари хранят пары ключ-значение, в которых значениями могут быть данные любых типов. При изучении словарей были охвачены методы `dict.get()` и `dict.setdefault()`, а при описании словарей со значениями по умолчанию были продемонстрированы альтернативы этим методам. Подобно множествам, словари предоставляют очень эффективный оператор проверки на вхождение и обеспечивают быстрый доступ к элементам по ключу.

Списки, множества, словари – все они имеют собственные реализации генераторов, которые могут использоваться для создания коллекций этих типов из итерируемых объектов (которые в свою очередь также могут быть генераторами), с наложением дополнительных условий, если это необходимо. Функции `range()` и `zip()` часто используются для создания коллекций; обе эти функции удобно использовать в циклах `for ... in` и в генераторах.

Элементы изменяемых коллекций могут удаляться с помощью соответствующих методов, таких как `list.pop()` и `set.discard()`, или с помощью инструкции `del` – например, инструкция `del d[k]` удалит из словаря `d` элемент с ключом `k`.

В языке Python используются ссылки на объекты, что делает операцию присваивания чрезвычайно эффективной, но это также означает, что при использовании оператора присваивания (`=`) сами объекты не копируются. Мы рассмотрели различия между поверхностным и глубоким копированием, а позднее увидели, как с помощью операции извлечения среза `L[:]` можно создать поверхностную копию всего списка, а с помощью метода `dict.copy()` создать поверхностную копию словаря. Любой объект, допускающий возможность копирования, может быть скопирован с помощью функций из модуля `copy`, например, функция `copy.copy()` выполняет поверхностное копирование, а функция `copy.deepcopy()` выполняет глубокое копирование.

Мы познакомились с высокооптимизированной встроенной функцией `sorted()`. Эта функция широко используется при программировании на языке Python. В языке Python отсутствуют типы упорядоченных коллекций, поэтому, когда необходимо выполнить итерации через коллекции в определенном порядке, это можно реализовать с помощью функции `sorted()`.

Встроенных типов коллекций – кортежей, списков, множеств, фиксированных множеств и словарей – вполне достаточно для решения лю-

бого круга задач. Тем не менее в стандартной библиотеке имеется несколько дополнительных типов коллекций и значительное количество типов, созданных сторонними разработчиками.

Часто возникает необходимость читать коллекции данных из файлов или записывать содержимое коллекций в файлы. В этой главе, в ходе очень краткого рассмотрения принципов работы с текстовыми файлами, основное наше внимание мы уделили чтению и записи текстовых строк. Полное описание работы с файлами приводится в главе 7, а дополнительные средства сохранения данных – в главе 11.

В следующей главе мы поближе познакомимся с управляющими конструкциями языка Python, среди которых будет представлена одна конструкция, с которой мы еще не сталкивались. Кроме того, мы более подробно изучим тему обработки исключений и некоторые дополнительные инструкции, такие как `assert`, с которыми мы еще не знакомы. Помимо этого, мы рассмотрим порядок создания собственных функций и в частности изучим чрезвычайно гибкий механизм работы с аргументами, используемый в языке Python.

Упражнения

1. Модифицируйте программу *external_sites.py* и задействуйте в ней словарь со значениями по умолчанию. Это легко сделать, добавив одну дополнительную инструкцию `import` и изменив всего две строки. Решение приводится в файле *external_sites_ans.py*.
2. Модифицируйте программу *uniquewords2.py* так, чтобы она выводила слова не в алфавитном порядке, а по частоте встречаемости. Вам потребуется обойти элементы словаря и создать маленькую функцию из двух строк, которая будет извлекать значение каждого элемента, и передать ее в виде аргумента `key` функции `sorted()`. Кроме того, потребуется соответствующим образом изменить инструкцию `print()`. Это несложно, но тут есть некоторый подвох. Решение приводится в файле *uniquewords_ans.py*.
3. Модифицируйте программу *generate_usernames.py* так, чтобы в каждой строке она выводила информацию о двух пользователях, ограничив длину имени 17 символами; через каждые 64 строки программа должна выводить символ перевода формата и в начале каждой страницы она должна выводить заголовки столбцов. Ниже приводится пример того, как должен выглядеть вывод программы:

```
Name                ID   Username  Name                ID   Username
----------------- ------ --------- ----------------- ------ ---------
Aitkin, Shatha... (2370) saitkin   Alderson, Nicole. (8429) nalderso
Allison, Karma... (8621) kallison  Alwood, Kole E... (2095) kealwood
Annie, Neervana.. (2633) nannie    Apperson, Lucyann (7282) leappers
```

Это достаточно сложно. Вам потребуется сохранить заголовки столбцов в переменных, чтобы потом их можно было использовать по мере необходимости, и изменить спецификаторы формата, чтобы обеспечить вывод более коротких имен. Один из способов обеспечить постраничный вывод заключается в том, чтобы сохранить все выводимые строки в списке, а затем выполнить обход списка, используя оператор извлечения среза с шагом для получения элементов слева и справа и применяя функцию `zip()` для их объединения. Решение приводится в файле *generate_usernames_ans.py*, а достаточно большой объем исходных данных вы найдете в файле *data/users2.txt*.

4

- Управляющие структуры
- Обработка исключений
- Собственные функции

Управляющие структуры и функции

В первых двух разделах этой главы будут рассматриваться управляющие структуры языка Python, причем в первом разделе будут рассматриваться условные инструкции и циклы, а во втором – инструкции обработки исключительных ситуаций. Большая часть управляющих структур и основы обработки исключений уже рассматривались в главе 1, но в этой главе они будут изучены более полно, включая дополнительный синтаксис управляющих структур, а также порядок возбуждения исключительных ситуаций и создание собственных исключений.

Третий и самый большой раздел посвящен созданию собственных функций, и здесь будет подробно рассматриваться чрезвычайно гибкий механизм работы с аргументами функций. Собственные функции позволяют нам упаковывать параметризуемую функциональность и уменьшать объем программного кода за счет оформления повторяющихся фрагментов в виде функций многократного использования. (В следующей главе мы узнаем, как создавать собственные модули, чтобы одни и те же функции можно было использовать в разных программах.)

Управляющие структуры

В языке Python условное ветвление реализуется с помощью инструкции `if`, а циклическая обработка – с помощью инструкций `while` и `for ... in`. В языке Python имеется также такая конструкция, как *условное выражение* – вариант инструкции `if`, аналог трехместного оператора (`?:`), имеющегося в C-подобных языках.

Условное ветвление

Как мы видели в главе 1, общий синтаксис инструкции условного ветвления в языке Python имеет следующий вид:

```
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
...
elif boolean_expressionN:
    suiteN
else:
    else_suite
```

Инструкция может содержать ноль или более предложений `elif`. Заключительное предложение `else` также является необязательным. Если необходимо предусмотреть ветку для какого-то особого случая, который не требует никакой обработки, в качестве блока кода этой ветки можно использовать инструкцию `pass` (она ничего не делает и просто является инструкцией-заполнителем, используемой там, где должна находиться хотя бы одна инструкция).

В некоторых случаях можно сократить инструкцию `if ... else` до единственного *условного выражения*. Ниже приводится синтаксис условных выражений:

```
expression1 if boolean_expression else expression2
```

Если логическое выражение *boolean_expression* возвращает значение `True`, результатом всего условного выражения будет результат выражения *expression1*, в противном случае – результат выражения *expression2*.

В практике программирования часто применяется такой прием, когда в переменную сначала записывается значение по умолчанию, а затем в случае необходимости оно изменяется, например, по требованию пользователя или в результате выяснения типа платформы, на которой выполняется программа. Ниже приводится типичная реализация такого приема с использованием инструкции `if`:

```
offset = 20
if not sys.platform.startswith("win"):
    offset = 10
```

Переменная `sys.platform` хранит название текущей платформы, например, «win32» или «linux2». Тот же результат можно получить с помощью условного выражения:

```
offset = 20 if sys.platform.startswith("win") else 10
```

В данном случае нет необходимости использовать круглые скобки, но их использование поможет избежать малозаметных ловушек. Например, предположим, что нам необходимо записать в переменную `width`

значение 100 и прибавить к нему 10, если переменная margin имеет значение True. Мы могли бы написать такое выражение:

```
width = 100 + 10 if margin else 0    # ОШИБКА!
```

Особенно неприятно, что эта строка программного кода работает правильно, когда переменная margin имеет значение True, записывая значение 110 в переменную width. Но когда переменная margin имеет значение False, в переменную width вместо 100 будет записано значение 0. Это происходит потому, что интерпретатор Python воспринимает выражение 100 + 10 как часть *expression1* условного выражения. Решить эту проблему можно с помощью круглых скобок:

```
width = 100 + (10 if margin else 0)
```

Кроме того, круглые скобки делают программный код более понятным для человека.

Условные выражения могут использоваться для видоизменения сообщений, выводимых для пользователя. Например, при выводе числа обработанных файлов, вместо того чтобы печатать «0 file(s)», «1 file(s)»[1] или что-то подобное, можно было бы использовать пару условных выражений:

```
print("{0} file{1}".format((count if count != 0 else "no"),
                           ("s" if count != 1 else "")))
```

Эта инструкция будет выводить «no files», «1 file», «2 files» и т. д., что придаст программе более профессиональный вид.

Циклы

В языке Python есть две инструкции циклов – while и for ... in, которые имеют более сложный синтаксис, чем было показано в главе 1.

Циклы while

Ниже приводится полный синтаксис цикла while:

```
while boolean_expression:
    while_suite
else:
    else_suite
```

Предложение else является необязательным. До тех пор, пока выражение *boolean_expression* возвращает значение True, в цикле будет выполняться блок *while_suite*. Если выражение *boolean_expression* вернет

[1] Имеется в виду склонение по числам, то есть вместо «1 файл(ов)», «5 файл(ов)» можно выводить более правильно: «1 файл», «5 файлов». Но в отличие от английского языка реализация правильного склонения по числам в русском языке не уместится в два условных выражения. – *Прим. перев.*

значение False, цикл завершится, и при наличии предложения else будет выполнен блок *else_suite*. Если внутри блока *while_suite* выполняется инструкция continue, то управление немедленно передается в начало цикла и выражение *boolean_expression* вычисляется снова. Если цикл не завершается нормально, блок предложения else не выполняется.

Необязательное предложение else имеет несколько сбивающее с толку название, поскольку оно выполняется во всех в случаях, когда цикл нормально завершается. Если цикл завершается в результате выполнения инструкции break или return, когда цикл находится внутри функции или метода, или в результате исключения, то блок *else_suite* предложения else *не* выполняется. (При возникновении исключительной ситуации интерпретатор Python пропускает предложение else и пытается отыскать подходящий обработчик исключения, о чем будет рассказываться в следующем разделе.) Плюсом такой реализации является одинаковое поведение предложения else в циклах while, в циклах for ... in и в блоках try ... except.

Рассмотрим пример, демонстрирующий предложение else в действии. Методы str.index() и list.index() возвращают индекс заданной подстроки или элемента или возбуждают исключение ValueError, если подстрока или элемент не найдены. Метод str.find() делает то же самое, но в случае неудачи он не возбуждает исключение, а возвращает значение −1. Для списков не существует эквивалентного метода, но при желании мы могли бы создать такую функцию, использующую цикл while:

```
def list_find(lst, target):
    index = 0
    while index < len(lst):
        if lst[index] == target:
            break
        index += 1
    else:
        index = -1
    return index
```

Эта функция просматривает список в поисках заданного элемента target. Если искомый элемент будет найден, инструкция break завершит цикл и вызывающей программе будет возвращен соответствующий индекс. Если искомый элемент не будет найден, цикл достигнет конца списка и завершится обычным способом. В случае нормального завершения цикла будет выполнен блок в предложении else, индекс получит значение −1 и будет возвращен вызывающей программе.

Циклы for

Подобно циклу while, полный синтаксис цикла for ... in также включает необязательное предложение else:

```
for expression in iterable:
    for_suite
else:
    else_suite
```

В качестве выражения *expression* обычно используется либо единственная переменная, либо последовательность переменных, как правило, в форме кортежа. Если в качестве выражения *expression* используется кортеж или список, каждый элемент итерируемого объекта *iterable* распаковывается в элементы *expression*.

Если внутри блока *for_suite* встретится инструкция `continue`, управление будет немедленно передано в начало цикла и будет начата новая итерация. Если цикл завершается по выполнении всех итераций и в цикле присутствует предложение `else`, выполняется блок *else_suite*. Если выполнение цикла прерывается принудительно (инструкцией `break` или `return`), управление немедленно передается первой инструкции, следующей за циклом, а дополнительное предложение `else` при этом пропускается. Точно так же, когда возбуждается исключение, интерпретатор Python пропускает предложение `else` и пытается отыскать подходящий обработчик исключения (о чем будет рассказываться в следующем разделе).

Функция enumerate(), стр. 166

Ниже приводится версия функции `list_find()`, реализованная на базе цикла `for ... in`, которая так же, как и версия на базе цикла `while`, демонстрирует предложение `else` в действии:

```
def list_find(lst, target):
    for index, x in enumerate(lst):
        if x == target:
            break
    else:
        index = -1
    return index
```

Как видно из этого фрагмента, переменные, созданные в выражении *expression* цикла `for ... in`, продолжают существовать после завершения цикла. Как и любые локальные переменные, они прекращают свое существование после выхода из области видимости, включающей их.

Обработка исключений

Об ошибках и исключительных ситуациях интерпретатор Python сообщает посредством возбуждения исключений, хотя в некоторых библиотеках сторонних разработчиков еще используются устаревшие приемы, такие как возврат «ошибочного» значения.

Перехват и возбуждение исключений

Перехватывать исключения можно с помощью блоков `try ... except`, которые имеют следующий синтаксис:

```
try:
    try_suite
except exception_group1 as variable1:
    except_suite1
…
except exception_groupN as variableN:
    except_suiteN
else:
    else_suite
finally:
    finally_suite
```

Эта конструкция должна содержать хотя бы один блок `except`, а блоки `else` и `finally` являются необязательными. Блок *else_suite* выполняется, только если блок *try_suite* завершается обычным способом, и не выполняется в случае возникновения исключения. Если блок `finally` присутствует, он выполняется всегда и в последнюю очередь.

Каждая группа *exception_group* в предложении `except` может быть единственным исключением или кортежем исключений в круглых скобках. Часть `as` *variable* в каждой группе является необязательной. В случае ее использования в переменную *variable* записывается ссылка на исключение, которое возникло, благодаря этому к нему можно будет обратиться в блоке *except_suite*.

Если исключение возникнет во время выполнения блока *try_suite*, интерпретатор поочередно проверит каждое предложение `except`. Если будет найдена соответствующая группа *exception_group*, будет выполнен соответствующий блок *except_suite*. Соответствующей считается группа, в которой присутствует исключение того же типа, что и возникшее исключение, или возникшее исключение является подклассом[1] одного из исключений, перечисленных в группе.

Например, если при поиске по словарю возникнет исключение `KeyError`, первое предложение `except`, содержащее класс `Exception`, будет считаться соответствующим, так как `KeyError` является (косвенно) подклассом `Exception`. Если ни одна из групп не содержит класс `Exception` (в чем нет ничего необычного), но имеется группа с классом `Lookup-`

[1] Как будет показано в главе 6, в объектно-ориентированном программировании обычно создаются иерархии классов, то есть один класс (тип данных) наследует другой класс. В языке Python родоначальником любой иерархии является класс `object` – все остальные классы прямо или косвенно наследуют его. Подкласс – это класс, который наследует другой класс, поэтому все классы в языке Python (за исключением класса `object`) являются подклассами, так как все они так или иначе наследуют класс `object`.

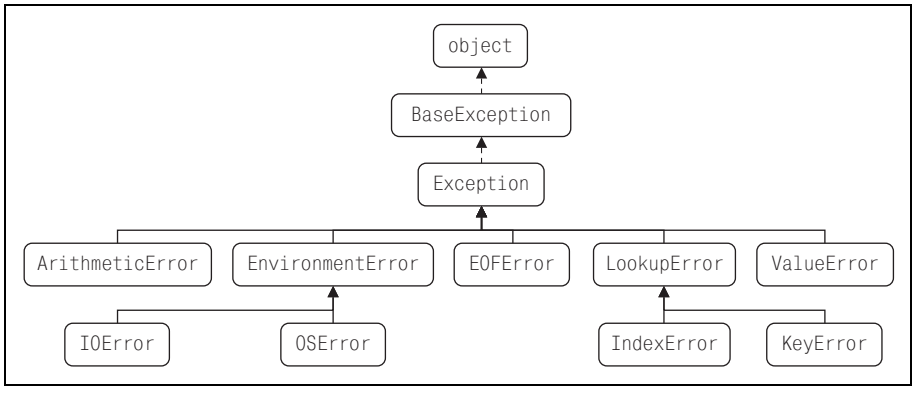

***Рис. 4.1.** Фрагмент иерархии классов исключений в языке Python*

`Error`, исключение `KeyError` будет соответствовать этой группе, потому что класс `KeyError` является подклассом класса `LookupError`. И если нет группы, в которой присутствовал бы класс `Exception` или `LookupError`, но имеется группа, содержащая класс `KeyError`, эта группа будет считаться соответствующей. На рис. 4.1 приводится фрагмент иерархии классов исключений.

Ниже приводится пример *неправильного* использования:

```
try:
    x = d[5]
except LookupError:    # НЕВЕРНЫЙ ПОРЯДОК
    print("Lookup error occurred")
except KeyError:
    print("Invalid key used")
```

Если в словаре `d` не будет найден элемент с ключом 5, для нас было бы желательно обработать исключение `KeyError`, а не более общее `LookupError`. Но в данном случае блок `except` с классом `KeyError` никогда не будет выполняться. В случае возникновения исключения `KeyError` соответствующим будет признан блок `except` с классом `LookupError`, потому что `LookupError` является базовым классом для `KeyError`, то есть класс `LookupError` находится выше класса `KeyError` в иерархии классов исключений. Поэтому в случае использования нескольких блоков `except` необходимо всегда располагать их сверху вниз в порядке от более специализированных (расположенных ниже в иерархии) к более общим (расположенных выше в иерархии).

```
try:
    x = d[k / n]
except Exception:    # ПЛОХАЯ ПРАКТИКА
    print("Something happened")
```

Обратите внимание, что обычно не принято указывать класс `Exception` в предложении `except`, так как оно будет соответствовать любому исключению и легко может скрыть логические ошибки в программном коде. Возможно, в этом примере предполагалось перехватить исключение `KeyError`, но если `n` имеет значение 0, то мы неумышленно перехватим и исключение `ZeroDivisionError`.

Имеется также возможность записать предложение в форме `except:`, то есть вообще без указания группы исключений. Подобный блок `except` будет перехватывать любые исключения, включая те, что наследуют класс `BaseException`, но не наследующие класс `Exception` (они не показаны на рис. 4.1). Этот вариант порождает те же проблемы, что и при использовании предложения `except Exception`, и даже еще хуже; такое предложение никогда не должно использоваться.

Если интерпретатор Python не обнаружит ни одного соответствующего предложения `except`, он начнет подъем вверх по стеку вызовов, пытаясь отыскать подходящий обработчик исключения. Если такой обработчик не будет найден, программа завершит свою работу с выводом диагностической информации и сообщения об ошибке.

Если исключение не возникло, будет выполнен необязательный блок `else`, если таковой имеется. И в любом случае, то есть независимо от того, возникло ли исключение или нет, и было ли оно обработано или интерпретатору предстоит выполнить подъем по стеку вызовов, *всегда* выполняется блок `finally`, если он присутствует. Если исключение не возникло или было обработано одним из блоков `except`, блок `finally` будет выполнен самым последним, но если для возникшего исключения не было найдено соответствующего блока `except`, то сначала будет выполнен блок `finally`, и только потом интерпретатор передаст исключение вверх по стеку вызовов. Такое гарантированное выполнение блока `finally` может быть очень полезным, когда необходимо обеспечить корректное освобождение ресурсов. На рис. 4.2 демонстрируется порядок выполнения типичной конструкции `try ... except ... finally`.

Ниже приводится окончательная версия функции `list_find()`, на этот раз она использует механизм обработки исключения:

```
def list_find(lst, target):
    try:
        index = lst.index(target)
    except ValueError:
        index = -1
    return index
```

Здесь мы использовали конструкцию `try ... except` для преобразования исключения в возвращаемое значение. Аналогичный подход можно использовать для перехвата одних исключений и возбуждения других – с этим приемом мы познакомимся очень скоро.

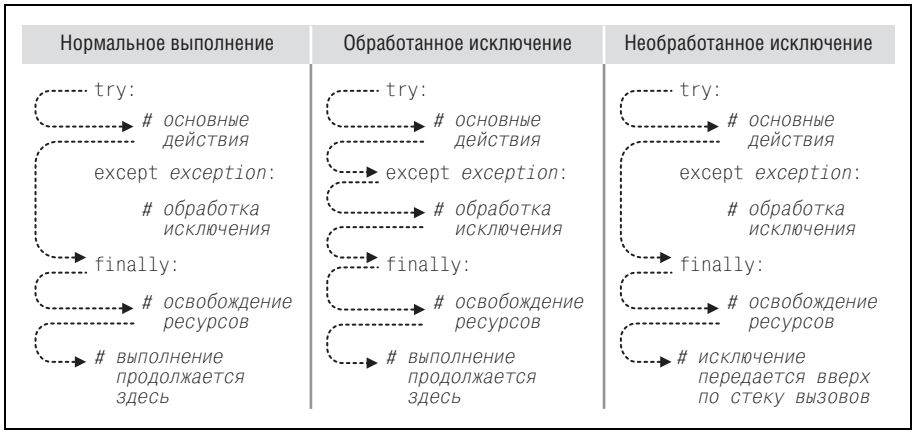

***Рис. 4.2.** Порядок выполнения конструкции try ... except ... finally*

Язык Python предоставляет возможность использовать более простую конструкцию `try ... finally`, которая в некоторых ситуациях может быть весьма удобна:

```
try:
    try_suite
finally:
    finally_suite
```

Неважно, что произойдет в блоке *try_suite* (кроме краха системы или программы!), в любом случае блок *finally_suite* будет выполнен. Того же эффекта, аналогичного использованию конструкции `try ... finally`, можно достичь с помощью инструкции `with` и менеджера контекста (о которых будет рассказываться в главе 8).

Очень часто конструкция `try ... except ... finally` используется для обработки ошибок, возникающих при работе с файлами. Например, программа *noblanks.py* принимает список имен файлов в виде аргументов командной строки и для каждого из них воспроизводит другой файл с тем же самым именем, но с расширением *.nb*, и с тем же содержимым, за исключением пустых строк. Ниже приводится функция `read_data()` из этой программы:

```
def read_data(filename):
    lines = []
    fh = None
    try:
        fh = open(filename, encoding="utf8")
        for line in fh:
            if line.strip():
                lines.append(line)
```

```
except (IOError, OSError) as err:
    print(err)
    return []
finally:
    if fh is not None:
        fh.close()
return lines
```

Изначально функция записывает в переменную `fh` значение `None`, так как вполне возможно, что вызов функции `open()` потерпит неудачу, тогда переменной `fh` ничего не будет присвоено (и в ней останется значение `None`) и будет возбуждено исключение. Если возникнет одно из исключений, которые мы определили (`IOError` или `OSError`), обработчик выведет сообщение об ошибке и вернет пустой список. Но, обратите внимание, что *прежде,* чем функция действительно вернет управление, будет выполнен блок `finally` и файл будет закрыт, если перед этим он был благополучно открыт.

Обратите также внимание, что если возникнет ошибка, связанная с кодировкой символов, файл все равно будет закрыт, хотя функция не предусматривает обработку соответствующего исключения (`ValueError`). Если это произойдет, интерпретатор сначала выполнит блок `finally`, а затем передаст исключение вверх по стеку вызовов, при этом возвращаемое значение будет отброшено, так как функция завершится в результате необработанного исключения. А так как в данном примере нет соответствующего блока `except`, который обрабатывал бы ошибки, связанные с кодировкой, программа завершит свою работу с выводом диагностической информации.

Предложение `except` в данном примере можно было бы записать более кратко:

```
except EnvironmentError as err:
    print(err)
    return []
```

Этот прием будет работать, так как `EnvironmentError` является базовым классом как для класса `IOError`, так и для класса `OSError`.

В главе 8 демонстрируется более компактный способ, гарантирующий закрытие файлов, не требующий наличия блока `finally`.

Менеджеры контекста, стр. 428

Возбуждение исключений

Исключения представляют собой удобное средство управления потоком выполнения. Мы можем воспользоваться этим, используя либо встроенные исключения, либо создавая свои собственные и возбуждая нужные нам, когда это необходимо. Возбудить исключение можно одним из двух способов:

```
raise exception(args)
raise
```

В первом случае, то есть когда явно указывается возбуждаемое исключение, оно должно быть либо встроенным, либо нашим собственным, наследующим класс `Exception`. Если исключению в виде аргумента передается некоторый текст, этот текст будет выведен на экран, если исключение не будет обработано программой. Во втором случае, то есть когда исключение не указывается, инструкция `raise` повторно возбудит текущее активное исключение, а в случае отсутствия активного исключения будет возбуждено исключение `TypeError`.

Собственные исключения

Собственные исключения – это наши собственные типы данных (классы). Создание классов будет рассматриваться в главе 6, но поскольку создать простейший тип собственного исключения не составляет никакого труда, мы покажем, как это делается:

```
class exceptionName(baseException): pass
```

Базовым классом *basException* должен быть либо класс `Exception`, либо один из его наследников.

Собственные исключения нередко используются, чтобы избежать глубоко вложенных циклов. Например, допустим, что у нас имеется объект `table`, хранящий записи (строки), каждая из которых состоит из полей (столбцов), в каждом из которых может иметься несколько значений (элементов). Тогда поиск определенного значения можно было бы реализовать примерно так:

```
found = False
for row, record in enumerate(table):
    for column, field in enumerate(record):
        for index, item in enumerate(field):
            if item == target:
                found = True
                break
        if found:
            break
    if found:
        break
if found:
    print("found at ({0}, {1}, {2})".format(row, column, index))
else:
    print("not found")
```

Эти 15 строк программного кода осложняет тот факт, что нам пришлось предусмотреть прерывание каждого цикла в отдельности. Альтернативное решение заключается в использовании нестандартного исключения:

```
class FoundException(Exception): pass

try:
    for row, record in enumerate(table):
        for column, field in enumerate(record):
            for index, item in enumerate(field):
                if item == target:
                    raise FoundException()
except FoundException:
    print("found at ({0}, {1}, {2})".format(row, column, index))
else:
    print("not found")
```

Этот прием позволил сократить программный код до десяти строк (или до 11, если включить определение класса исключения) и придал коду более удобочитаемый вид. Если искомый элемент будет найден, возбуждается наше собственное исключение и выполняется соответствующий блок `except`, при этом блок `else` не выполняется. Если искомый элемент не будет найден, исключение не возбуждается и тогда в конце выполняется блок `else`.

Рассмотрим еще один пример, демонстрирующий другие способы обработки исключений. Все фрагменты взяты из программы *checktags.py*, которая читает содержимое файлов HTML, имена которых передаются в виде аргументов командной строки, и выполняет некоторые простые проверки, чтобы убедиться, что все теги начинаются с символа «<» и заканчиваются символом «>» и все сущности оформлены правильно. В программе определяются четыре нестандартных исключения:

```
class InvalidEntityError(Exception): pass
class InvalidNumericEntityError(invalidEntityError): pass
class InvalidAlphaEntityError(invalidEntityError): pass
class InvalidTagContentError(Exception): pass
```

Второе и третье исключения наследуют первое; для чего это необходимо, мы увидим, когда будем обсуждать программный код, использующий эти исключение. Функция `parse()`, использующая эти исключения, содержит более 70 строк программного кода, поэтому мы покажем только ту часть функции, которая имеет непосредственное отношение к обработке исключений.

```
fh = None
try:
    fh = open(filename, encoding="utf8")
    errors = False
    for lino, line in enumerate(fh, start=1):
        for column, c in enumerate(line, start=1):
            try:
```

Этот фрагмент начинается вполне традиционно, записывая значение `None` в переменную, которая впоследствии будет ссылаться на объект

файла, и помещая все действия с файлом в блок try. Программа читает содержимое файла строку за строкой и каждую строку символ за символом.

Примечательно, что здесь имеется два блока try – внешний используется для обработки исключений, которые могут возникнуть при работе с объектом файла, а внутренний – для обработки исключений, возникающих в ходе синтаксического анализа.

```
...
elif state == PARSING_ENTITY:
    if c == ";":
        if entity.startswith("#"):
            if frozenset(entity[1:]) - HEXDIGITS:
                raise InvalidNumericEntityError()
        elif not entity.isalpha():
            raise InvalidAlphaEntityError()
    ...
```

Функция может находиться в нескольких состояниях, например, после чтения символа амперсанда (&) она входит в состояние PARSING_ENTITY и запоминает символы, расположенные между амперсандом и точкой с запятой (но не включая их), в строке entity.

Тип set, стр. 144

Часть программного кода, которая показана здесь, обрабатывает случай, когда точка с запятой была обнаружена в процессе чтения сущности. Если это числовая сущность (начинается комбинацией символов «&#», за которой следуют шестнадцатеричные цифры и символ «;», например: «&#20AC;»), мы преобразуем числовую часть в множество и исключаем из него все шестнадцатеричные цифры – если после этого множество окажется непустым, следовательно, в числе был указан как минимум один ошибочный символ, и мы возбуждаем собственное исключение. Если это текстовая сущность (комбинация, начинающаяся с символа «&», за которым следуют алфавитные символы и символ «;», например: «©»), мы возбуждаем исключение, если будет обнаружен неалфавитный символ.

```
...
except (invalidEntityError,
        InvalidTagContentError) as err:
    if isinstance(err, InvalidNumericEntityError):
        error = "invalid numeric entity"
    elif isinstance(err, InvalidAlphaEntityError):
        error = "invalid alphabetic entity"

    elif isinstance(err, InvalidTagContentError):
        error = "invalid tag"
    print("ERROR {0} in {1} on line {2} column {3}"
```

```
                    .format(error, filename, lino, column))
            if skip_on_first_error:
                raise
    ...
```

Если возникает исключение, связанное с синтаксическим анализом, оно будет перехвачено блоком except. Используя базовый класс InvalidEntityError, мы перехватим оба типа исключений – InvalidNumericEntityError и InvalidAlphaEntityError. После этого с помощью функции isinstance() проверяется, какое именно исключение возникло, и определяется соответствующее сообщение об ошибке. Встроенная функция isinstance() возвращает True, если первый ее аргумент имеет тот же тип, что и тип (или один из его базовых типов), переданный во втором аргументе.

Функция isinstance(), стр. 284

Можно было бы использовать отдельные блоки except для каждого из трех наших собственных исключений синтаксического анализа, но в данном случае, объединив обработку в одном блоке, нам удалось избежать необходимости повторять четыре последние строки (от инструкции print() до инструкции raise) в каждом из них.

Программа имеет два режима работы. Если переменная skip_on_first_error имеет значение False, программа продолжит проверку файла даже после обнаружения синтаксической ошибки, что может привести к выводу множества сообщений об ошибках для каждого файла. Если переменная skip_on_first_error имеет значение True, то после выявления синтаксической ошибки (одной и только одной) в файле программа выведет сообщение об ошибке и повторно возбудит исключение синтаксического анализа, которое будет перехвачено внешним блоком try (где выполняется обработка каждого файла).

```
    ...
    elif state == PARSING_ENTITY:
        raise EOFError("missing ';' at end of " + filename)
    ...
```

По завершении синтаксического анализа нам необходимо проверить, не оказались ли мы в середине сущности. Если это произошло, возбуждается встроенное исключение EOFError, сообщающее о встрече конца файла, которому мы передаем собственный текст сообщения. Точно так же для этой цели мы могли бы использовать свое собственное исключение.

```
except (invalidEntityError, InvalidTagContentError):
    pass # Уже было обработано
except EOFError as err:
    print("ERROR unexpected EOF:", err)
except EnvironmentError as err:
    print(err)
```

```
finally:
    if fh is not None:
        fh.close()
```

Во внешнем блоке `try` мы использовали отдельные блоки `except`, потому что в каждом конкретном случае обработка выполняется по-разному. Если было получено исключение синтаксического анализа, мы знаем, что соответствующее сообщение уже было выведено и нам нужно лишь прервать работу с эти файлом и перейти к следующему, поэтому нам ничего не требуется делать в обработчике исключений. Если было получено исключение `EOFError`, это может быть результат действительно преждевременного достижения конца файла либо повторного его возбуждения. В любом случае мы выводим сообщение и текст исключения. Если возникло исключение `EnvironmentError` (то есть если возникло исключение `IOError` или `OSError`), мы просто выводим сообщение исключения. В заключение, независимо от того, что произошло, если файл оказался открытым, мы закрываем его.

Собственные функции

Функции представляют собой средство, дающее возможность упаковывать и параметризовать функциональность. В языке Python можно создать четыре типа функций: глобальные функции, локальные функции, лямбда-функции и методы.

Все функции, которые мы создавали до сих пор, являются *глобальными* функциями. Глобальные объекты (включая функции) доступны из любой точки программного кода в том же модуле (то есть в том же самом файле *.py*), которому принадлежит объект. Глобальные объекты доступны также и из других модулей, как будет показано в следующей главе.

Локальные функции (их еще называют вложенными функциями) – это функции, которые объявляются внутри других функций. Эти функции видимы только внутри тех функций, где они были объявлены – они особенно удобны для создания небольших вспомогательных функций, которые нигде больше не используются. Мы познакомимся с ними в главе 7.

Лямбда-функции – это выражения, поэтому они могут создаваться непосредственно в месте их использования; они имеют множество ограничений по сравнению с обычными функциями.

Методы – это те же функции, которые ассоциированы с определенным типом данных и могут использоваться только в связке с этим типом данных; методы будут представлены в главе 6, когда будут рассматриваться вопросы объектно-ориентированного программирования.

В языке Python имеется множество встроенных функций, а стандартная библиотека и библиотеки сторонних разработчиков добавляют

еще сотни (тысячи, если посчитать еще и методы) поэтому большинство функций, которые нам могут потребоваться, уже написаны. По этой причине всегда стоит обращаться к электронной документации, чтобы увидеть, какие функции доступны. Смотрите врезку «Электронная документация».

Электронная документация

В этой книге дается полный охват языка Python 3, встроенных функций и наиболее часто используемых модулей из стандартной библиотеки, тем не менее в электронной документации можно найти значительный объем справочной информации о языке Python и особенно об обширнейшей стандартной библиотеке. Электронная документация доступна на сайте *docs.python.org*, а также поставляется в составе самого интерпретатора Python.

Для операционной системы Windows документация поставляется в формате справочных файлов Windows. Выберите пункт меню Пуск→Все программы→Python 3.x→Python Manuals (Start→All Programs→ Python 3.x→Python Manuals), чтобы запустить средство просмотра справочных файлов Windows. Этот инструмент обладает функциями индексирования и поиска, которые упрощают возможность поиска по документу. Пользователи операционной системы UNIX получают документацию в формате HTML. В дополнение к различным гиперссылкам в ней содержатся различные страницы с предметными указателями. Кроме того, в левой части каждой страницы присутствует очень удобная функция «Quick Search».

Наиболее часто начинающими пользователями используется документ «Library Reference», а опытными пользователями – документ «Global Module Index». Оба документа содержат ссылки, ведущие на страницы с описанием всей стандартной библиотеки Python, а, кроме того, документ «Library Reference» содержит ссылки на страницы с описанием всех встроенных функциональных возможностей языка Python.

Определенно имеет смысл ознакомиться с документацией, особенно с документами «Library Reference» и «Global Module Index», чтобы получить представление о том, что может предложить стандартная библиотека, и пощелкать мышью на темах, которые вас заинтересуют. Это даст вам первое впечатление о том, что доступно, и поможет запомнить, где можно отыскать документацию, которая будет представлять для вас интерес. (Краткое описание стандартной библиотеки языка Python приводится в главе 5.)

Кроме того, в интерпретаторе также имеется справочная система. Если вызвать встроенную функцию `help()` без аргументов, вы попадете в электронную справочную систему – чтобы получить в ней нужную информацию, просто следуйте инструкциям, а чтобы вернуться в интерпретатор – введите символ «`q`» или команду «`quit`». Если вы знаете, описание какого модуля или типа данных хотите получить, можно вызвать функцию `help()`, передав ей имя модуля или типа в виде аргумента. Например, выполнив инструкцию `help(str)`, вы получите информацию о типе данных `str`, включая описания всех его методов; инструкция `help(dict.update)` выведет информацию о методе `update()` типа данных `dict`; а инструкция `help(os)` отобразит информацию о модуле `os` (если перед этим он был импортирован).

Если вы уже знакомы с языком Python, то часто бывает достаточно просто просмотреть, какие атрибуты (например, методы) имеет тот или иной тип данных. Эту информацию можно получить с помощью функции `dir()`, например, вызов `dir(str)` перечислит все методы строк, а вызов `dir(os)` перечислит все константы и функции модуля `os` (опять же при условии, что модуль был предварительно импортирован).

Синтаксис создания функции (глобальной или локальной) имеет следующий вид:

```
def functionName(parameters):
    suite
```

Параметры *parameters* являются необязательными и при наличии более одного параметра записываются как последовательность идентификаторов через запятую или в виде последовательности пар *identifier=value*, о чем вскоре будет говориться подробнее. Например, ниже приводится функция, которая вычисляет площадь треугольника по формуле Герона:

```
def heron(a, b, c):
    s = (a + b + c) / 2
    return math.sqrt(s * (s - a) * (s - b) * (s - c))
```

Внутри функции каждый параметр, `a`, `b` и `c`, инициализируется соответствующими значениями, переданными в виде аргументов. При вызове функции мы должны указать все аргументы, например, `heron(3, 4, 5)`. Если передать слишком мало или слишком много аргументов, будет возбуждено исключение `TypeError`. Производя такой вызов, мы говорим, что используем *позиционные аргументы*, потому что каждый переданный аргумент становится значением параметра в соответ-

ствующей позиции. То есть в данном случае при вызове функции параметр a получит значение 3, параметр b – значение 4 и параметр c – значение 5.

Все функции в языке Python возвращают какое-либо значение, хотя вполне возможно (и часто так и делается) просто игнорировать это значение. Возвращаемое значение может быть единственным значением или кортежем значений, а сами значения могут быть коллекциями, поэтому практически не существует никаких ограничений на то, что могут возвращать функции. Мы можем покинуть функцию в любой момент, используя инструкцию return. Если инструкция return используется без аргументов или если мы вообще не используем инструкцию return, функция будет возвращать значение None. (В главе 6 мы рассмотрим инструкцию yield, которая в функциях определенного типа может использоваться вместо инструкции return.)

Некоторые функции имеют параметры, для которых может существовать вполне разумное значение по умолчанию. Например, ниже приводится функция, которая подсчитывает количество алфавитных символов в строке; по умолчанию подразумеваются алфавитные символы из набора ASCII:

```
def letter_count(text, letters=string.ascii_letters):
    letters = frozenset(letters)
    count = 0
    for char in text:
        if char in letters:
            count += 1
    return count
```

Здесь при помощи синтаксиса *parameter=default* было определено значение по умолчанию для параметра letters. Это позволяет вызывать функцию letter_count() с единственным аргументом, например, letter_count("Maggie and Hopey"). В этом случае внутри функции параметр letter будет содержать строку, которая была задана как значение по умолчанию. Но за нами сохраняется возможность изменить значение по умолчанию, например, указав дополнительный позиционный аргумент: letter_count("Maggie and Hopey", "aeiouAEIOU"), или используя именованный аргумент (об именованных аргументах рассказывается ниже): letter_count("Maggie and Hopey", letters="aeiouAEIOU").

Синтаксис параметров не позволяет указывать параметры, не имеющие значений по умолчанию, после параметров со значениями по умолчанию, поэтому такое определение: def bad(a, b=1, c):, будет вызывать синтаксическую ошибку. С другой стороны, мы не обязаны передавать аргументы в том порядке, в каком они указаны в определении функции – мы можем использовать именованные аргументы и передавать их в виде *name=value*.

Ниже демонстрируется короткая функция, возвращающая заданную строку, если ее длина меньше или равна заданной длине, и усеченную

версию строки с добавлением в конец значения параметра `indicator` – в противном случае:

```
def shorten(text, length=25, indicator="..."):
    if len(text) > length:
        text = text[:length - len(indicator)] + indicator
    return text
```

Вот несколько примеров вызова этой функции:

```
shorten("The Road")                          # вернет: 'The Road'
shorten(length=7, text="The Road")           # вернет: 'The ...'
shorten("The Road", indicator="&", length=7) # вернет: 'The Ro&'
shorten("The Road", 7, "&")                  # вернет: 'The Ro&'
```

Поскольку оба параметра, `length` и `indicator`, имеют значение по умолчанию, любой из них или даже оба сразу могут быть опущены, тогда будут использоваться значения по умолчанию – этот случай соответствует первому вызову. Во втором вызове оба аргумента являются именованными, поэтому их можно указывать в любом порядке. В третьем вызове используются позиционный аргумент и именованные аргументы. Первым указан позиционный аргумент (позиционные аргументы всегда должны предшествовать именованным аргументам), а за ним следуют два именованных аргумента. В четвертом вызове все аргументы позиционные.

Врезка «Чтение и запись текстовых файлов», стр. 157

Различие между обязательным и необязательным параметром заключается в наличии значения по умолчанию, то есть параметр со значением по умолчанию является необязательным (интерпретатор может использовать значение по умолчанию), а параметр без значения по умолчанию является обязательным (интерпретатор не может делать никаких предположений). Осторожное использование значений по умолчанию может упростить программный код и сделать вызовы функций более понятными. Вспомните, что функция `open()` имеет один обязательный аргумент (имя файла) и шесть необязательных аргументов. Используя смесь из позиционных и именованных аргументов, мы можем указывать только необходимые аргументы, опуская другие. Это дает нам возможность записать такой вызов: `open(filename, encoding="utf8")`, вместо того чтобы указывать все аргументы, например: `open(filename, "r", None, "utf8", None, None, True)`. Еще одно преимущество использования именованных аргументов состоит в том, что они способны сделать вызов функции более удобочитаемым, особенно в случае использования логических аргументов.

Значения по умолчанию создаются на этапе выполнения инструкции `def` (то есть в момент создания функции), а *не* в момент

ее вызова. Для неизменяемых аргументов, таких как строки или числа, это не имеет никакого значения, но в использовании изменяемых аргументов кроется труднозаметная ловушка.

```
def append_if_even(x, lst=[]): # ОШИБКА!
    if x % 2 == 0:
        lst.append(x)
    return lst
```

В момент создания этой функции параметр `lst` ссылается на новый список. Всякий раз, когда эта функция вызывается с одним первым параметром, параметр `lst` будет ссылаться на список, созданный как значение по умолчанию вместе с функцией – то есть при каждом таком вызове новый список создаваться не будет. Как правило, это не совсем то, что нам хотелось бы – мы ожидаем, что каждый раз, когда функция вызывается без второго аргумента, будет создаваться новый пустой список. Ниже приводится новая версия функции, на этот раз использующая правильный подход к работе с изменяемыми аргументами, имеющими значения по умолчанию:

```
def append_if_even(x, lst=None):
    if lst is None:
        lst = []
    if x % 2 == 0:
        lst.append(x)
    return lst
```

Здесь, всякий раз, когда функция вызывается без второго аргумента, мы создаем новый список. А если аргумент `lst` определен, используется он, как и в предыдущей версии функции. Такой прием, основанный на использовании значения по умолчанию `None` и создании нового объекта, должен применяться к словарям, спискам, множествам и любым другим изменяемым типам данных, которые предполагается использовать в виде аргументов со значениями по умолчанию. Ниже приводится немного более короткая версия функции, которая обладает тем же поведением:

```
def append_if_even(x, lst=None):
    lst = [] if lst is None else lst
    if x % 2 == 0:
        lst.append(x)
    return lst
```

Использование условного выражения позволяет сократить размер функции на одну строку для каждого параметра, имеющего изменяемое значение по умолчанию.

Имена и строки документирования

Использование осмысленных имен для функций и их параметров помогает понимать назначение функции другим программистам, а также,

спустя некоторое время после создания функции, и самому автору функции. Ниже приводятся несколько основных правил, которых мы рекомендуем придерживаться.

- Используйте единую схему именования и придерживайтесь ее неуклонно. В этой книге имена `ИМЕНА КОНСТАНТ` записываются символами в верхнем регистре; `имена Классов` (и исключений) записываются символами верхнего и нижнего регистра, причем каждое слово в имени начинается с символа верхнего регистра; похожим образом записываются `имена Функций` и методов графического интерфейса, за исключением первого символа, который всегда записывается в нижнем регистре; а все остальные `имена` записываются только символами нижнего регистра или `символами_нижнего_регистра_с_символом_подчеркивания`.
- Избегайте использовать аббревиатуры в любых именах, если эти аббревиатуры не являются стандартными и не получили широкого распространения.
- Соблюдайте разумный подход при выборе имен для переменных и параметров: имя `x` прекрасно подходит для координаты x, а имя `i` отлично подходит на роль переменной цикла, но вообще имена должны быть достаточно длинными и описательными. Имя должно описывать скорее назначение элемента данных, чем его тип (например, имя `amount_due` предпочтительнее, чем имя `money`), если только имя не является универсальным для конкретного типа данных, например, имя параметра `text` в функции `shorten()` (стр. 209).
- Имена функций и методов должны говорить о том, что они *делают* или что они *возвращают* (в зависимости от их назначения), и никогда – как они это делают, потому что эта характеристика может измениться со временем.

Ниже приводятся несколько примеров имен:

```
def find(l, s, i=0):                                  # НЕУДАЧНЫЙ ВЫБОР
def linear_search(l, s, i=0):                         # НЕУДАЧНЫЙ ВЫБОР
def first_index_of(sorted_name_list, name, start=0): # ХОРОШИЙ ВЫБОР
```

Все три функции возвращают индекс первого вхождения имени в списке имен, причем поиск в списке начинается с указанного индекса и используется алгоритм поиска, который предполагает, что список уже отсортирован.

Первый случай приходится признать неудачным, потому что имя функции ничего не говорит о том, что будут искать, а имена ее параметров (по всей видимости) указывают на их типы (`list`, `str`, `int`), но ничего не говорят об их назначении. Второй вариант также следует признать неудачным, потому что имя функции описывает алгоритм, использованный первоначально, но с течением времени алгоритм могут изменить. Это может быть неважно для того, кто будет пользоваться функцией, но может вводить в заблуждение тех, кто будет сопрово-

ждать программный код, если имя функции предполагает реализацию алгоритма линейного поиска, а в действительности со временем функцию могли переписать под использование алгоритма поиска методом дихотомии. Третий вариант можно назвать удачным, потому что имя функции говорит о том, что она возвращает, а имена параметров недвусмысленно показывают, что ожидает получить функция.

Ни одна из функций не имеет возможности указать, что произойдет, если поиск завершится неудачей – вернут ли они, скажем, значение –1, или возбудят исключение? В каком-то виде такая информация должна быть включена в описание, предоставляемое пользователям функции.

Мы можем добавить описание к любой функции, используя *строки документирования* – это обычные строки, которые следуют сразу за строкой с инструкцией `def` и перед программным кодом функции. Например, ниже приводится функция `shorten()`, которую мы уже видели ранее, но на этот раз приводится полный ее текст:

```
def shorten(text, length=25, indicator="..."):
    """Возвращает text или усеченную его копию с добавлением
       indicator в конце

    text - любая строка; length - максимальная длина возвращаемой
    строки string (включая indicator); indicator - строка,
    добавляемая в конец результата, чтобы показать,
    что текст аргумента text был усечен

    >>> shorten("The Road")
    'The Road'
    >>> shorten("No Country for Old Men", 20)
    'No Country for Ol...'
    >>> shorten("Cities of the Plain", 15, "*")
    'Cities of the *'
    """
    if len(text) > length:
        text = text[:length - len(indicator)] + indicator
    return text
```

Нет ничего необычного в том, что текст описания длиннее самой функции. В соответствии с общепринятыми соглашениями первая строка в описании должна представлять собой краткое, однострочное описание функции, затем следует пустая строка и далее следует полное описание функции, в конце которого приводится несколько примеров того, как может выглядеть использование функции в интерактивной оболочке. В главе 5 мы узнаем, как примеры, присутствующие в описании функции, могут использоваться для нужд модульного тестирования.

Распаковывание аргументов и параметров

Распаковывание последовательностей, стр. 137

В предыдущей главе мы видели, что для передачи позиционных аргументов можно использовать оператор распаковывания последовательностей (`*`). Например, если возникает необходимость вычислить площадь треугольника, а длины всех его сторон хранятся в списке, то мы могли бы вызвать функцию так: `heron(sides[0], sides[1], sides[2])`, или просто распаковать список и сделать вызов намного проще: `heron(*sides)`. Если элементов в списке (или в другой последовательности) больше, чем параметров в функции, мы можем воспользоваться операцией извлечения среза, чтобы извлечь нужное число аргументов.

Мы можем также использовать оператор распаковывания последовательности в списке параметров функции. Это удобно, когда необходимо создать функцию, которая может принимать переменное число позиционных аргументов. Ниже приводится функция `product()`, которая вычисляет произведение своих аргументов:

```
def product(*args):
    result = 1
    for arg in args:
        result *= arg
    return result
```

Эта функция имеет единственный аргумент с именем `args`. Наличие символа * перед ним означает, что внутри функции параметр `args` обретает форму кортежа, значениями элементов которого будут значения переданных аргументов. Ниже приводятся несколько примеров вызова функции:

```
product(1, 2, 3, 4) # args == (1, 2, 3, 4); вернет: 24
product(5, 3, 8)    # args == (5, 3, 8); вернет: 120
product(11)         # args == (11,); вернет: 11
```

Мы можем использовать именованные аргументы вслед за позиционными, как в функции, которая приводится ниже, вычисляющей сумму своих аргументов, каждый из которых возводится в заданную степень:

```
def sum_of_powers(*args, power=1):
    result = 0
    for arg in args:
        result += arg ** power
    return result
```

Эта функция может вызываться только с позиционными аргументами, например: `sum_of_powers(1, 3, 5)`, или как с позиционными, так и с именованным аргументами, например: `sum_of_powers(1, 3, 5, power=2)`.

Допускается также использовать символ «`*`» в качестве самостоятельного «параметра». В данном случае он указывает, что после символа

«*» не может быть других позиционных параметров, однако указание именованных аргументов допускается. Ниже приводится модифицированная версия функции heron(). На этот раз функция принимает точно три позиционных аргумента и один необязательный именованный аргумент.

```
def heron2(a, b, c, *, units="meters"):
    s = (a + b + c) / 2
    area = math.sqrt(s * (s - a) * (s - b) * (s - c))
    return "{0} {1}".format(area, units)
```

Ниже приводятся несколько примеров вызовов функции:

```
heron2(25, 24, 7)                 # вернет: '84.0 meters'
heron2(41, 9, 40, units="inches") # вернет: '180.0 inches'
heron2(25, 24, 7, "inches")       # ОШИБКА! Возбудит исключение TypeError
```

В третьем вызове мы попытались передать четыре позиционных аргумента, но оператор * не позволяет этого и вызывает исключение TypeError.

Поместив оператор * первым в списке параметров, мы тем самым полностью запретим использование *любых* позиционных аргументов и вынудим тех, кто будет вызывать ее, использовать именованные аргументы. Ниже приводится пример сигнатуры такой (вымышленной) функции:

```
def print_setup(*, paper="Letter", copies=1, color=False):
```

Мы можем вызывать функцию print_setup() без аргументов, допуская использование значений по умолчанию. Или изменить некоторые или все значения по умолчанию, например: print_setup(paper="A4", color=True). Но если мы попытаемся использовать позиционные аргументы, например: print_setup("A4"), будет возбуждено исключение TypeError.

Так же, как мы распаковываем последовательности для заполнения позиционных параметров, можно распаковывать и отображения – с помощью оператора распаковывания отображений (**).[1] Мы можем использовать оператор **, чтобы передать содержимое словаря в функцию print_setup(). Например:

```
options = dict(paper="A4", color=True)
print_setup(**options)
```

В данном случае пары «ключ-значение» словаря options будут распакованы, и каждое значение будет ассоциировано с параметром, чье имя соответствует ключу этого значения. Если в словаре обнаружится ключ, не совпадающий ни с одним именем параметра, будет возбуждено исключение TypeError. Любые аргументы, для которых в словаре не

[1] Как мы уже видели в главе 2, когда ** используется в качестве двухместного оператора, он является аналогом функции pow().

найдется соответствующего элемента, получат значение по умолчанию, но если такие аргументы не имеют значения по умолчанию, будет возбуждено исключение `TypeError`.

Кроме того, имеется возможность использовать оператор распаковывания вместе с параметрами в объявлении функции. Это позволяет создавать функции, способные принимать любое число именованных аргументов. Ниже приводится функция `add_person_details()`, которая принимает номер карточки социального страхования и фамилию в виде позиционных аргументов, а также произвольное число именованных аргументов:

```
def add_person_details(ssn, surname, **kwargs):
    print("SSN =", ssn)
    print(" surname =", surname)
    for key in sorted(kwargs):
        print(" {0} = {1}".format(key, kwargs[key]))
```

Функция print()

Функция `print()` может принимать произвольное число позиционных аргументов и имеет три именованных аргумента: `sep`, `end` и `file`. Все именованные аргументы имеют значение по умолчанию. В качестве значения по умолчанию для параметра `sep` используется пробел – если функции передано два или более позиционных аргументов, при выводе они отделяются друг от друга значением `sep`, но если функция получит единственный позиционный аргумент, этот параметр в выводе не участвует. В качестве значения по умолчанию для параметра `end` используется символ `\n`, именно по этой причине функция `print()` завершает вывод своих аргументов переводом строки. В качестве значения по умолчанию для параметра `file` используется `sys.stdout`, поток стандартного вывода, который обычно представляет консоль.

Имеется возможность переопределять значение любого именованного аргумента, если значения по умолчанию чем-то не устраивают. Например, в аргументе `file` можно передать объект файла, открытый на запись или на дополнение в конец, а в аргументах `sep` и `end` можно передавать любые строки, включая пустые.

Когда необходимо вывести несколько элементов в одной и той же строке, обычно применяется прием, когда функция `print()` вызывается с аргументом `end`, в качестве значения которого используется требуемый разделитель, а в самом конце вызывается функция `print()` без аргументов, только для того, чтобы вывести символ перевода строки. Например, смотрите функцию `print_digits()` (стр. 213).

Эта функция может вызываться как только с двумя позиционными аргументами, так и с дополнительной информацией, например: `add_person_details(83272171, "Luther", forename="Lexis", age=47)`. Такая возможность обеспечивает огромную гибкость. Конечно, мы можем также одновременно принимать переменное число позиционных аргументов и переменное число именованных аргументов:

```
def print_args(*args, **kwargs):
    for i, arg in enumerate(args):
        print("positional argument {0} = {1}".format(i, arg))
    for key in kwargs:
        print("keyword argument {0} = {1}".format(key, kwargs[key]))
```

Эта функция просто выводит полученные аргументы. Она может вызываться вообще без аргументов или с произвольным числом позиционных и именованных аргументов.

Доступ к переменным в глобальной области видимости

Иногда бывает удобно иметь несколько глобальных переменных, доступных из разных функций программы. В этом нет ничего плохого, если речь идет о «константах», но в случае переменных – это не самый лучший выход, хотя для коротких одноразовых программ это в некоторых случаях можно считать допустимым.

Программа *digit_names.py* принимает необязательный код языка («en» или «fr») и число в виде аргументов командной строки и выводит названия всех цифр заданного числа. То есть если в командной строке программе было передано число «123», она выведет «one two three». В программе имеется три глобальные переменные:

```
Language = "en"

ENGLISH = {0: "zero", 1: "one", 2: "two", 3: "three", 4: "four",
           5: "five", 6: "six", 7: "seven", 8: "eight", 9: "nine"}
FRENCH = {0: "zйro", 1: "un", 2: "deux", 3: "trois", 4: "quatre",
          5: "cinq", 6: "six", 7: "sept", 8: "huit", 9: "neuf"}
```

Мы следуем соглашению, в соответствии с которым имена переменных, играющих роль констант, записываются только символами верхнего регистра, и установили английский язык по умолчанию. (В языке Python отсутствует прямой способ создания констант, вместо этого он полностью полагается на то, что программист будет неуклонно следовать общепринятым соглашениям.) В некотором другом месте программы выполняется обращение к переменной `Language`, и ее значение используется при выборе соответствующего словаря:

```
def print_digits(digits):
    dictionary = ENGLISH if Language == "en" else FRENCH
    for digit in digits:
        print(dictionary[int(digit)], end=" ")
    print()
```

Когда интерпретатор Python встречает имя переменной Language внутри функции, он пытается отыскать его в локальной области видимости (в области видимости функции) и не находит. Поэтому он продолжает поиск в глобальной области видимости (в области видимости файла *.py*), где и обнаруживает его. Назначение именованного аргумента end, используемого в первом вызове функции print(), описывается во врезке «Функция print()».

Ниже приводится содержимое функции main() программы. Она изменяет значение переменной Language в случае необходимости и вызывает функцию print_digits() для вывода результата.

```
def main():
    if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
        print("usage: {0} [en|fr] number".format(sys.argv[0]))
        sys.exit()

    args = sys.argv[1:]
    if args[0] in {"en", "fr"}:
        global Language
        Language = args.pop(0)
    print_digits(args.pop(0))
```

Обратите внимание на использование инструкции global в этой функции. Эта инструкция используется для того, чтобы сообщить интерпретатору, что данная переменная существует в глобальной области видимости (в области видимости файла *.py*) и что операция присваивания должна применяться к глобальной переменной; без этой инструкции операция присваивания создаст локальную переменную с тем же именем.

Если не использовать инструкцию global, программа сохранит свою работоспособность, но когда интерпретатор встретит переменную Language в условной инструкции if, он попытается отыскать ее в локальной области видимости (в области видимости функции) и, не обнаружив ее, создаст новую локальную переменную с именем Language, оставив глобальную переменную Language без изменений. Эта малозаметная ошибка будет проявляться только в случае запуска программы с аргументом «fr», потому что в этом случае будет создана новая локальная переменная Language, в которую будет записано значение «fr», а глобальная переменная Language, которая используется функцией print_digits(), по-прежнему будет иметь значение «en».

В сложных программах лучше вообще не использовать глобальные переменные, за исключением констант, которые не требуют употребления инструкции global.

Лямбда-функции

Лямбда-функции – это функции, для создания которых используется следующий синтаксис:

```
lambda parameters: expression
```

Часть *parameters* является необязательной, а если она присутствует, то обычно представляет собой простой список имен переменных, разделенных запятыми, то есть позиционных аргументов, хотя при необходимости допускается использовать полный синтаксис определения аргументов, используемый в инструкции `def`. Выражение *expression* не может содержать условных инструкций или циклов (хотя условные выражения являются допустимыми), а также не может содержать инструкцию `return` (или `yield`). Результатом лямбда-выражения является анонимная функция. Когда вызывается лямбда-функция, она возвращает результат вычисления выражения *expression*. Если выражение *expression* представляет собой кортеж, оно должно быть заключено в круглые скобки.

Функции-генераторы, стр. 324

Ниже приводится пример простой лямбда-функции, которая добавляет (или не добавляет) суффикс «s» в зависимости от того, имеет ли аргумент значение 1:

```
s = lambda x: "" if x == 1 else "s"
```

Лямбда-выражение возвращает анонимную функцию, которая присваивается переменной `s`. Любая (вызываемая) переменная может вызываться как функция при помощи круглых скобок, поэтому после выполнения некоторой операции можно при помощи функции `s()` вывести сообщение с числом обработанных файлов, например: `print("{0} file{1} processed".format(count, s(count))`.

Лямбда-функции часто используются в виде аргумента `key` встроенной функции `sorted()` или метода `list.sort()`. Предположим, что имеется список, элементами которого являются трехэлементные кортежи (номер группы, порядковый номер, название), и нам необходимо отсортировать этот список различными способами. Ниже приводится пример такого списка:

```
elements = [(2, 12, "Mg"), (1, 11, "Na"), (1, 3, "Li"), (2, 4, "Be")]
```

Отсортировав список, мы получим следующий результат:

```
[(1, 3, 'Li'), (1, 11, 'Na'), (2, 4, 'Be'), (2, 12, 'Mg')]
```

Ранее, когда мы рассматривали функцию `sorted()`, то видели, что имеется возможность изменить порядок сортировки, если в аргументе `key` передать требуемую функ-

Функция `sorted()`, стр. 164, 170

цию. Например, если необходимо отсортировать список не по естественному порядку: номер группы, порядковый номер и название, а по порядковому номеру и названию, то мы могли бы написать маленькую функцию `def ignore0(e): return e[1], e[2]` и передавать ее в аргументе `key`. Но создавать в программе массу крошечных функций, подобных этой, очень неудобно, поэтому часто используется альтернативный подход, основанный на применении лямбда-функций:

```
elements.sort(key=lambda e: (e[1], e[2]))
```

Здесь в качестве значения аргумента `key` используется выражение `lambda e: (e[1], e[2])`, которому в виде аргумента `e` последовательно передаются все трехэлементные кортежи из списка. Круглые скобки, окружающие лямбда-выражение, обязательны, когда выражение является кортежем и лямбда-функция создается как аргумент другой функции. Для достижения того же эффекта можно было бы использовать операцию извлечения среза:

```
elements.sort(key=lambda e: e[1:3])
```

Немного более сложная версия обеспечивает возможность сортировки по названию, без учета регистра символов, и порядковому номеру:

```
elements.sort(key=lambda e: (e[2].lower(), e[1]))
```

Ниже приводятся два эквивалентных способа создания функции, вычисляющей площадь треугольника по известной формуле $\frac{1}{2} \times$ *основание* $\times$ *высота*:

```
area = lambda b, h: 0.5 * b * h
```

```
def area(b, h):
    return 0.5 * b * h
```

Мы можем вызвать функцию `area(6, 5)` независимо от того, была ли она создана как лямбда-функция или с помощью инструкции `def`, и результат будет один и тот же.

Словари со значениями по умолчанию, стр. 161

Другая замечательная область применения лямбда-функций – создание словарей со значениями по умолчанию. В предыдущей главе говорилось, что при обращении к такому словарю с несуществующим ключом будет создан соответствующий элемент с указанным ключом и со значением по умолчанию. Ниже приводятся несколько примеров создания таких словарей:

```
minus_one_dict = collections.defaultdict(lambda: -1)
point_zero_dict = collections.defaultdict(lambda: (0, 0))
message_dict = collections.defaultdict(lambda: "No message available")
```

При обращении к словарю `minus_one_dict` с несуществующим ключом будет создан новый элемент с указанным ключом и со значением -1. Точно так же при обращении к словарю `point_zero_dict` вновь созданный элемент получит в качестве значения кортеж `(0, 0)`, а при обра-

щении к словарю message_dict значением по умолчанию будет строка «No message available».

Утверждения

Что произойдет, если функция получит аргументы, имеющие ошибочные значения? Что случится, если в реализации алгоритма будет допущена ошибка и вычисления будут выполнены неправильно? Самое *неприятное*, что может произойти, – это то, что программа будет выполняться без каких-либо видимых проблем, но будет давать неверные результаты. Один из способов избежать таких коварных проблем состоит в том, чтобы писать тесты, о которых кратко будет рассказано в главе 5. Другой способ состоит в том, чтобы определить предварительные условия и ожидаемый конечный результат, и сообщать об ошибке, если они не соответствуют друг другу. В идеале следует использовать как тестирование, так и метод на основе сравнения предварительных условий и ожидаемых результатов.

Предварительные условия и ожидаемый результат можно задать с помощью инструкции assert, которая имеет следующий синтаксис:

```
assert boolean_expression, optional_expression
```

Если выражение *boolean_expression* возвращает значение False, возбуждается исключение AssertionError. Если задано необязательное выражение *optional_expression*, оно будет использовано в качестве аргумента исключения AssertionError, что удобно для передачи сообщений об ошибках. Однако следует отметить, что утверждения предназначены для использования разработчиками, а не конечными пользователями. Проблемы, возникающие в процессе нормальной эксплуатации программы, такие как отсутствующие файлы или ошибочные аргументы командной строки, должны обрабатываться другими средствами, например, посредством вывода сообщений об ошибках или записи сообщений в файл журнала.

Ниже приводятся две версии функции product(). Обе версии эквивалентны в том смысле, что обе они *требуют*, чтобы все передаваемые им аргументы имели ненулевое значение, а вызов с нулевыми значениями рассматривается как ошибка программиста.

```
def product(*args): # пессимистичная
    assert all(args), "0 argument"
    result = 1
    for arg in args:
        result *= arg
    return result
```

```
def product(*args): # оптимистичная
    result = 1
    for arg in args:
        result *= arg
    assert result, "0 argument"
    return result
```

«Пессимистичная» версия, слева, проверяет все аргументы (точнее – до первого нулевого значения) при каждом вызове. «Оптимистичная» версия, справа, проверяет результат – если хотя бы один аргумент имеет нулевое значение, то и результат будет равен 0.

Если любую из этих версий вызвать со значением 0 в одном из аргументов, будет возбуждено исключение AssertionError и в поток стандартного вывода сообщений об ошибках (sys.stderr – обычно консоль) будет выведено следующее:

```
Traceback (most recent call last):
  File "program.py", line 456, in <module>
    x = product(1, 2, 0, 4, 8)
  File "program.py", line 452, in product
    assert result, "0 argument"
AssertionError: 0 argument
```

Интерпретатор автоматически выведет диагностическую информацию с именем файла, именем функции и номером строки, а также текст сообщения, указанного нами.

Но как быть с инструкциями assert, после того как программа будет готова к выпуску в виде окончательной версии (при этом она, безусловно, успешно проходит все тесты и не нарушает ни одного утверждения)? Мы можем сообщить интерпретатору о том, что больше не требуется выполнять инструкции assert, то есть их нужно отбрасывать во время выполнения программы. Для этого программа должна запускаться с ключом командной строки -O, например python -O program.py. Другой способ добиться этого состоит в том, чтобы установить переменную окружения PYTHONOPTIMIZE в значение 0.[1] Если наши пользователи не пользуются строками документирования (обычно им этого и не требуется), мы можем использовать ключ -OO, который эффективно удаляет как инструкции assert, так и строки документирования: обратите внимание, что для установки такого поведения нет переменной окружения. Некоторые разработчики используют упрощенный подход: они создают копии программ, где все инструкции assert закомментированы, и в случае прохождения всех тестов они выпускают версию программы без инструкций assert.

Пример: make_html_skeleton.py

В этом разделе мы объединим некоторые приемы, описанные в этой главе, и продемонстрируем их в контексте законченной программы.

Очень маленькие веб-сайты часто создаются и обслуживаются вручную. Один из способов облегчить эту работу состоит в том, чтобы написать программу, которая будет генерировать заготовки файлов HTML, которые позднее будут наполняться содержимым. Программа *make_html_skeleton.py* выполняется в интерактивном режиме, она запрашивает у пользователя различные сведения и затем создает заготовку файла HTML. Функция main() содержит цикл, позволяющий создавать одну заготовку за другой, и сохраняет общую информацию (на-

[1] Это буква «O», а не цифра 0. – *Прим. перев.*

пример, информацию об авторских правах), что избавляет пользователей от необходимости вводить ее снова и снова. Ниже приводится пример типичного сеанса работы с программой:

```
make_html_skeleton.py

Make HTML Skeleton

Enter your name (for copyright): Harold Pinter
Enter copyright year [2008]: 2009
Enter filename: career-synopsis
Enter title: Career Synopsis
Enter description (optional): synopsis of the career of Harold Pinter
Enter a keyword (optional): playwright
Enter a keyword (optional): actor
Enter a keyword (optional): activist
Enter a keyword (optional):
Enter the stylesheet filename (optional): style
Saved skeleton career-synopsis.html

Create another (y/n)? [y]:

Make HTML Skeleton

Enter your name (for copyright) [Harold Pinter]:
Enter copyright year [2009]:
Enter filename:
Cancelled

Create another (y/n)? [y]: n
```

Обратите внимание, что при создании второй заготовки имя и год получили значения по умолчанию, введенные ранее, поэтому пользователю не пришлось вводить их вторично. Но для имени файла значение по умолчанию отсутствует, поэтому, когда имя файла не было указано, процедура создания заготовки была прервана.

Теперь, когда мы увидели, как пользоваться программой, мы готовы приступить к изучению программного кода. Программа начинается двумя инструкциями импорта:

```
import datetime
import xml.sax.saxutils
```

Модуль `datetime` предоставляет ряд простых функций для создания объектов `datetime.date` и `datetime.time`. Модуль `xml.sax.saxutils` содержит удобную функцию `xml.sax.saxutils.escape()`, которая принимает строку и возвращает эквивалентную ей строку, в которой специальные символы языка разметки HTML («`&`», «`<`» и «`>`») замещаются их эквивалентами («`&`», «`<`» и «`>`»).

Далее определяются три глобальные строки, которые используются в качестве шаблонов.

```
COPYRIGHT_TEMPLATE = "Copyright (c) {0} {1}. All rights reserved."

STYLESHEET_TEMPLATE = ('<link rel="stylesheet" type="text/css" '
                       'media="all" href="{0}" />\n')

HTML_TEMPLATE = """<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" \
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
<head>
<title>{title}</title>
<!-- {copyright} -->
<meta name="Description" content="{description}" />
<meta name="Keywords" content="{keywords}" />
<meta equiv="content-type" content="text/html; charset=utf-8" />
{stylesheet}\
</head>
<body>

</body>
</html>
"""
```

Метод str.format(), стр. 100

Эти строки будут использоваться как шаблоны для вызова метода `str.format()`. В шаблоне `HTML_TEMPLATE` в качестве замещаемых полей мы использовали не числовые индексы, а имена, например, `{title}`. Ниже мы увидим, что для передачи значений в эти поля нам необходимо будет использовать именованные аргументы.

```
class CancelledError(Exception): pass
```

Затем определяется нестандартное исключение; мы встретимся с ним в паре функций программы.

Функция `main()` программы устанавливает некоторые начальные значения и входит в цикл. На каждой итерации пользователю предлагается ввести некоторую информацию о странице HTML, которая будет сгенерирована, и после создания каждой страницы предоставляется возможность завершить программу.

```
def main():
    information = dict(name=None, year=datetime.date.today().year,
                       filename=None, title=None, description=None,
                       keywords=None, stylesheet=None)
    while True:
        try:
            print("\nMake HTML Skeleton\n")
            populate_information(information)
            make_html_skeleton(**information)
        except CancelledError:
            print("Cancelled")
        if (get_string("\nCreate another (y/n)?", default="y").lower()
```

```
            not in {"y", "yes"}):
            break
```

Функция `datetime.date.today()` возвращает объект `datetime.date`, который хранит текущую дату. Нам требуется лишь значение атрибута `year` этого объекта. Во все остальные элементы данных записывается значение `None`, так как для них не существует разумных значений по умолчанию.

В цикле `while` программа выводит заголовок и вызывает функцию `populate_information()`, передавая ей словарь `information`. Внутри функции `populate_information()` производится заполнение этого словаря. Затем вызывается функция `make_html_skeleton()`, она принимает большое число аргументов, но чтобы явно не указывать значение каждого из них, мы просто распаковываем словарь `information`.

Если пользователь прерывает процесс создания заготовки страницы, например, отказом от ввода обязательного значения, программа выводит сообщение «Cancelled» (отменено). В конце каждой итерации (независимо от того, было ли отменено создание заготовки страницы или нет) пользователю задается вопрос: не желает ли он создать еще одну заготовку. Если пользователь отвечает отказом, производится выход из цикла и программа завершает работу.

```
def populate_information(information):
    name = get_string("Enter your name (for copyright)", "name",
                      information["name"])
    if not name:
        raise CancelledError()
    year = get_integer("Enter copyright year", "year",
                       information["year"], 2000,
                       datetime.date.today().year + 1, True)
    if year == 0:
        raise CancelledError()
    filename = get_string("Enter filename", "filename")
    if not filename:
        raise CancelledError()
    if not filename.endswith((".htm", ".html")):
        filename += ".html"
    ...
    information.update (name=name, year=year, filename=filename,
                        title=title, description=description,
                        keywords=keywords, stylesheet=stylesheet)
```

Мы опустили программный код, который запрашивает заголовок и текст описания, ключевые слова HTML и имя файла с таблицами стилей. Во всех этих случаях используется функция `get_string()`, которую мы увидим очень скоро. Достаточно лишь отметить, что эта функция принимает текст вопроса, «имя» соответствующей переменной (для вывода в сообщении об ошибке) и необязательное значение по умолчанию. Точно так же функция `get_integer()` принимает текст вопроса,

имя переменной, значение по умолчанию, минимальное и максимальное значения, а также признак – допустимо ли значение 0.

В конце функция заполняет словарь `information` новыми значениями, используя именованные аргументы. В каждой паре *key=value* имя *key* соответствует имени ключа в словаре, значение которого замещается указанным значением *value*, и в данном случае каждое значение *value* является переменной с тем же именем, что и соответствующий ей ключ словаря.

Эта функция не имеет явного возвращаемого значения (поэтому она возвращает значение `None`). Она может также завершаться в случае появления исключения `CancelledError`, в этом случае исключение будет передано вверх по стеку вызовов и обработано в функции `main()`.

Функцию `make_html_skeleton()` мы рассмотрим в два этапа.

```
def make_html_skeleton(year, name, title, description, keywords,
                       stylesheet, filename):
    copyright = COPYRIGHT_TEMPLATE.format(year,
                                          xml.sax.saxutils.escape(name))
    title = xml.sax.saxutils.escape(title)
    description = xml.sax.saxutils.escape(description)
    keywords = ",".join([xml.sax.saxutils.escape(k)
                         for k in keywords]) if keywords else ""
    stylesheet = (STYLESHEET_TEMPLATE.format(stylesheet)
                  if stylesheet else "")
    html = HTML_TEMPLATE.format(title=title, copyright=copyright,
                                description=description,
                                keywords=keywords,
                                stylesheet=stylesheet)
```

Чтобы получить текст с указанием авторских прав, мы вызываем метод `str.format()` для строки `COPYRIGHT_TEMPLATE`, передавая год и имя (дополнительно выполняя экранирование служебных символов HTML) в виде позиционных аргументов для замены полей `{0}` и `{1}`. В тексте заголовка и описания мы просто экранируем служебные символы HTML.

В случае ключевых слов у нас может быть два варианта действий, которые реализуются с использованием условного выражения. Если ключевые слова не были введены, то в качестве значения переменной `keywords` будет использоваться пустая строка. В противном случае с помощью генератора списков выполняется обход всех ключевых слов и создается новый список строк, в каждой из которых выполняется экранирование служебных символов HTML. После этого, с помощью метода `str.join()`, мы объединяем элементы списка в единую строку, разделяя их запятыми.

Текст переменной `stylesheet` создается аналогичным способом, что и текст с указанием авторских прав, но с применением условного выражения, чтобы в случае отсутствия имен файлов таблиц стилей получалась пустая строка.

Текст для переменной `html` создается из шаблона `HTML_TEMPLATE`, где для передачи данных в замещаемые поля используются не позиционные аргументы, как в других строках шаблонов, а именованные.

Метод str.format(), стр. 100

```
fh = None
try:
    fh = open(filename, "w", encoding="utf8")
    fh.write(html)
except EnvironmentError as err:
    print("ERROR", err)
else:
    print("Saved skeleton", filename)
finally:
    if fh is not None:
        fh.close()
```

Как только заготовка файла HTML будет готова, мы записываем ее в файл с заданным именем. После этого пользователь извещается, что файл заготовки был сохранен, или выводится сообщение об ошибке, если что-то пошло не так. Как обычно, чтобы гарантировать закрытие файла, если он был открыт, используется предложение `finally`.

```
def get_string(message, name="string", default=None,
               minimum_length=0, maximum_length=80):
    message += ": " if default is None else " [{0}]: ".format(default)
    while True:
        try:
            line = input(message)
            if not line:
                if default is not None:
                    return default
                if minimum_length == 0:
                    return ""
                else:
                    raise ValueError("{0} may not be empty".format(
                                     name))
            if not (minimum_length <= len(line) <= maximum_length):
                raise ValueError("{0} must have at least {1} and "
                        "at most {2} characters".format(
                        name, minimum_length, maximum_length))
            return line
        except ValueError as err:
            print("ERROR", err)
```

Функция имеет один обязательный аргумент `message` и четыре необязательных аргумента. Если значение аргумента `default` определено, оно включается в строку `message`, чтобы пользователь мог видеть значение по умолчанию, которое будет использоваться, если он просто нажмет клавишу Enter, не вводя никакого текста. Остальная часть функции заключена в бесконечный цикл. Цикл может быть прерван вводом

допустимой строки или в результате простого нажатия клавиши Enter, когда используется значение по умолчанию (если определено).

Кроме того, пользователь может прервать цикл и завершить работу программы, нажав комбинацию клавиш Ctrl+C, – в этом случае возбуждается исключение `KeyboardInterrupt`, но так как это исключение не обрабатывается ни одним из обработчиков, имеющихся в программе, это приведет к завершению программы с выводом диагностической информации. Следовало ли оставлять такую возможность прерывать цикл? Если бы мы этого не сделали и в программе обнаружилась бы ошибка, мы не оставили бы пользователю никакой возможности прервать работу программы, кроме как уничтожить процесс. Если нет достаточно веских причин препятствовать завершению программы по нажатию комбинации Ctrl+C, не следует обрабатывать это исключение ни в одном из обработчиков.

Примечательно, что эта функция достаточно универсальна и может использоваться не только в программе *make_html_skeleton.py*, но и во многих других интерактивных программах подобного типа. Такого многократного использования функции можно было бы добиться простым копированием текста, но такой прием может стать источником головной боли для того, кто будет сопровождать программы. В следующей главе мы узнаем, как создавать собственные модули, вмещающие функциональные возможности, которые могут совместно использоваться большим числом программ.

```
def get_integer(message, name="integer", default=None, minimum=0,
                maximum=100, allow_zero=True):
    ...
```

Эта функция по своей структуре настолько близка к функции `get_string()`, что нет необходимости воспроизводить ее здесь. (Безусловно, эта функция присутствует в исходных текстах примеров к книге.) Параметр `allow_zero` может быть полезен, когда 0 не является допустимым значением, но когда желательно обеспечить возможность ввода ошибочного значения, чтобы предоставить способ прервать процедуру создания заготовки. Другой способ, который можно было бы использовать, заключается в том, чтобы определить недопустимое значение в качестве значения по умолчанию; тогда возврат такого значения означал бы, что пользователь отменил операцию.

Последняя инструкция в программе – это простой вызов функции `main()`. Общий объем программы составляет чуть больше 150 строк, и она демонстрирует некоторые особенности языка Python, которые были представлены в этой и в предыдущих главах.

В заключение

В этой главе мы рассмотрели полный синтаксис всех управляющих структур языка Python. Здесь также было показано, как возбуждать и обрабатывать исключения и как создавать свои типы исключений.

Большая часть главы была посвящена созданию собственных функций. Мы увидели, как создаются функции, познакомились с некоторыми правилами выбора имен для функций и их параметров. Мы также увидели, как можно добавлять описание функций. Подробно был рассмотрен универсальный синтаксис определения параметров и передачи аргументов в языке Python, включая возможность передачи фиксированного и переменного числа позиционных и именованных аргументов, а также возможность определения для аргументов значений по умолчанию – как неизменяемых, так и изменяемых типов. Кроме того, мы коротко повторили порядок использования оператора распаковывания последовательностей `*` и показали, как выполнять распаковывание отображений с помощью оператора `**`.

Когда возникает необходимость присвоить глобальной переменной новое значение внутри функции, она должна быть объявлена с помощью инструкции `global`, чтобы предотвратить создание оператором присваивания новой локальной переменной. Однако вообще глобальные переменные лучше использовать только в качестве констант.

Лямбда-функции часто используются в качестве значения аргументов других функций или в других случаях, где функция должна передаваться в виде параметра. В этой главе было показано, что лямбда-функции могут использоваться как для создания анонимных функций, так и для создания коротких именованных функций – путем присваивания их переменным.

В главе также было рассмотрено применение инструкции `assert`. Эта инструкция очень удобна для проверки истинности предварительных условий и результатов при каждом обращении к функции и может оказать действенную помощь в создании надежных программ и в ликвидации ошибок.

В этой главе мы рассмотрели фундаментальные основы создания функций, а, кроме того, в нашем распоряжении имеется еще масса других приемов. Сюда входят возможность создания динамических функций (эти функции создаются во время выполнения программы, причем их реализация может изменяться в зависимости от обстоятельств), рассматриваемых в главе 5; локальных (вложенных) функций, рассматриваемых в главе 7; а также рекурсивных функций, генераторов функций и т. д., о чем будет рассказываться в главе 8.

В языке Python имеется значительное число встроенных функций и обширнейшая стандартная библиотека, тем не менее все равно остается вероятность, что мы напишем такие функции, которые пригодятся

во многих наших программах. Копирование функций из файла в файл может превратить в кошмар сопровождение таких программ, но, к счастью, Python предоставляет простое решение этой проблемы: модули. В следующей главе мы узнаем, как создавать свои собственные модули со своими функциями в них. Мы увидим, как выполнять импортирование функциональных возможностей из стандартной библиотеки и из наших модулей, а также коротко рассмотрим, что может предложить стандартная библиотека, чтобы нам не пришлось повторно изобретать колесо.

Упражнения

Напишите интерактивную программу обслуживания списков строк в файлах.

При запуске программа должна создать список всех файлов с расширением *.lst* в текущем каталоге. Воспользуйтесь функцией `os.listdir(".")`, чтобы получить список всех файлов, и отфильтруйте из него те файлы, которые не имеют расширения *.lst*. В случае отсутствия таких файлов программа должна попросить пользователя ввести имя файла и добавить расширение *.lst*, если пользователь не сделал этого. Если были найдены один или более файлов *.lst*, программа должна вывести их имена в виде списка пронумерованных строк, начиная с 1. Пользователю должно быть предложено ввести номер желаемого файла или 0; в последнем случае программа должна попросить у пользователя ввести имя нового файла.

Если был указан существующий файл, программа должна прочитать его содержимое. Если файл пуст или было указано имя нового файла, программа должна вывести сообщение «no items are in the list» (список не содержит элементов).

В случае отсутствия элементов должно быть предложено два варианта действий: «Add» (добавить) и «Quit» (выйти). Если список содержит один или более элементов, строки из списка должны выводиться пронумерованными, начиная с 1, а из доступных действий должны быть предложены варианты «Add» (добавить), «Delete» (удалить), «Save» (сохранить) (если файл еще не сохранялся) и «Quit» (выйти). Если пользователь выбирает действие «Quit» и при этом имеются несохраненные изменения, ему должна быть предоставлена возможность сохранить их. Ниже приводится пример сеанса работы с программой (большая часть пустых строк, а также заголовок «List Keeper», который выводится всякий раз при выводе списка, были удалены из листинга):

```
Choose filename: movies

-- no items are in the list --
[A]dd [Q]uit [a]: a
Add item: Love Actually
```

```
1: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]: a
Add item: About a Boy

1: About a Boy
2: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]:
Add item: Alien

1: About a Boy
2: Alien
3: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]: k
ERROR: invalid choice--enter one of 'AaDdSsQq'
Press Enter to continue...
[A]dd [D]elete [S]ave [Q]uit [a]: d
Delete item number (or 0 to cancel): 2

1: About a Boy
2: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]: s
Saved 2 items to movies.lst
Press Enter to continue...

1: About a Boy
2: Love Actually
[A]dd [D]elete [Q]uit [a]:
Add item: Four Weddings and a Funeral

1: About a Boy
2: Four Weddings and a Funeral
3: Love Actually
[A]dd [D]elete [S]ave [Q]uit [a]: q
Save unsaved changes (y/n) [y]:
Saved 3 items to movies.lst
```

Функция `main()` должна быть не очень большой (не более 30 строк) и должна содержать только основной цикл программы. Напишите функцию, которая будет получать имя нового или существующего файла (и в последнем случае загружать элементы списка), и функцию, которая будет выводить перечень доступных действий и принимать выбор пользователя. Напишите также функции, которые будут добавлять элемент, удалять элемент, выводить список (либо имен файлов, либо элементов списка строк), загружать список и сохранять список. Вставьте в свою программу копии функций `get_string()` и `get_integer()` из программы *make_html_skeleton.py* или напишите свои собственные версии.

При выводе элементов списка строк или имен файлов ширина поля для вывода номеров строк должна быть равна 1, если список содержит менее десяти элементов, 2 – если в списке менее 100 элементов и 3 – в противном случае.

Всегда выводите элементы списка в алфавитном порядке, без учета регистра символов, и следите за состоянием списка (за наличием несохраненных изменений). Действие «Save» должно предлагаться только при наличии несохраненных изменений, а перед выходом программа должна спрашивать у пользователя, не желает ли он сохранить изменения, только если таковые имеются. Добавление и удаление элементов считаются действиями, которые изменяют список, а после выполнения операции сохранения список снова должен считаться неизменным.

Пример решения находится в файле *Listkeeper.py* и занимает менее 200 строк программного кода.

- Модули и пакеты
- Обзор стандартной библиотеки языка Python

5

Модули

Функции позволяют упаковывать фрагменты программного кода, чтобы его можно было многократно использовать по всей программе, а модули обеспечивают средство объединения функций (и, как мы увидим в следующей главе, наших собственных типов данных) в коллекции, чтобы их можно было использовать в разных программах. В языке Python имеются также средства создания *пакетов* – наборов модулей, объединенных, как правило, по функциональным признакам или вследствие зависимости друг от друга.

В первом разделе этой главы описывается синтаксис операции импортирования функциональных возможностей из модулей и пакетов, входящих в состав стандартной библиотеки, или из наших собственных модулей и пакетов. Затем в этом же разделе будет показано, как создавать собственные пакеты и модули. Будут продемонстрированы два примера собственных модулей. Из них первый пример является вводным, а во втором демонстрируется, как решаются многочисленные проблемы, возникающие на практике, такие как платформенная независимость и тестирование.

Во втором разделе дается краткий обзор стандартной библиотеки языка Python. Очень важно знать, что может предложить стандартная библиотека, потому что использование предопределенных функциональных возможностей существенно ускоряет программирование, позволяя не создавать все и вся с чистого листа. Кроме того, многие модули из стандартной библиотеки используются очень широко. Они тщательно протестированы и обладают высокой надежностью. Помимо краткого обзора будет приведено несколько небольших примеров, иллюстрирующих типичные случаи использования. До-

Врезка «Электронная документация», стр. 203

полнительно будут приводиться ссылки на описания модулей в других главах.

Модули и пакеты

Модуль в языке Python – это обычный файл с расширением *.py*. Модуль может содержать любой программный код на языке Python. Каждая программа, которую мы писали до сих пор, находилась в отдельном файле *.py*, который можно считать не только программой, но и модулем. Основное различие между модулем и программой состоит в том, что программа предназначена для того, чтобы ее запускали, тогда как модуль предназначен для того, чтобы его импортировали и использовали в программах.

Не все модули располагаются в файлах с расширением *.py*, например, модуль `sys` встроен в Python, а некоторые модули написаны на других языках программирования (чаще всего на языке C). Однако большая часть библиотеки языка Python написана именно на языке Python, так, например, добавляя инструкцию `import collections`, мы получаем возможность создавать именованные кортежи вызовом функции `collections.namedtuple()`, а функциональные возможности, к которым мы получаем доступ, находятся в файле модуля *collections.py*. Для наших программ совершенно неважно, на каком языке программирования написан модуль, потому что все модули импортируются и используются одним и тем же способом.

Импортирование может выполняться несколькими синтаксическими конструкциями, например:

```
import importable
import importable1, importable2, ..., importableN
import importable as preferred_name
```

Пакеты, стр. 234

Здесь под *importable* подразумевается имя модуля, такое как `collections`, но точно так же это может быть пакет или модуль из пакета, и тогда все части имени отделяются друг от друга точками (`.`), например, `os.path`. Первые две конструкции мы используем на протяжении всей книги. В них нет ничего сложного, и они являются самыми безопасными, потому что позволяют избежать конфликтов имен, так как вынуждают программиста всегда использовать полные квалифицированные имена.

Третья конструкция позволяет давать импортируемому модулю или пакету имя по выбору – теоретически это может привести к конфликтам имен, но на практике синтаксис `as` обычно используется, чтобы как раз избежать их. Подобное переименование, в частности, удобно использовать при экспериментировании с различными реализациями одного и того же модуля. Например, допустим, что у нас имеется два модуля `MyModuleA` и `MyModuleB`, которые имеют один и тот же API (Appli-

cation Programming Interface – прикладной программный интерфейс); мы могли бы в программе записать инструкцию `import MyModuleA as MyModule`, а позднее легко переключиться на использование `import MyModuleB as MyModule`.

Где должна находиться инструкция `import`? Обычно все инструкции `import` помещаются в начало файла *.py*, после строки «shebang» и после описания модуля. И, как уже говорилось в главе 1, мы рекомендуем сначала импортировать модули стандартной библиотеки, затем модули сторонних разработчиков и в последнюю очередь свои собственные модули.

Ниже приводятся еще несколько вариантов использования инструкции `import`:

```
from importable import object as preferred_name
from importable import object1, object2, ..., objectN
from importable import (object1, object2, object3, object4, object5,
    object6, ..., objectN)
from importable import *
```

Эти синтаксические конструкции могут приводить к конфликтам имен, поскольку они обеспечивают непосредственный доступ к импортируемым объектам (переменным, функциям, типам данных или модулям). Если для импортирования большого числа объектов необходимо использовать синтаксис `from ... import`, мы можем расположить инструкцию импорта в нескольких строках, либо экранируя каждый символ перевода строки, кроме последнего, либо заключая список имен объектов в круглые скобки, как показано в третьем примере.

В последней синтаксической конструкции символ «`*`» означает «импортировать все имена, которые не являются частными». На практике это означает, что будут импортированы все объекты из модуля за исключением тех, чьи имена начинаются с символа подчеркивания, либо, если в модуле определена глобальная переменная `__all__` со списком имен, будут импортированы все объекты, имена которых перечислены в переменной `__all__`.

Функция `__all__()`, стр. 236

Ниже приводятся несколько примеров использования инструкции `import`:

```
import os
print(os.path.basename(filename)) # безопасный доступ по полным
                                  # квалифицированным именам

import os.path as path
print(path.basename(filename)) # есть риск конфликта имен с модулем path

from os import path
print(path.basename(filename)) # есть риск конфликта имен с модулем path

from os.path import basename
print(basename(filename))      # есть риск конфликта имен с модулем basename
```

```
from os.path import *
print(basename(filename))      # есть риск множественных конфликтов имен
```

Синтаксис `from` *`importable`* `import *` используется для импортирования всех объектов из модуля (или из всех модулей пакета) – это могут быть сотни имен. В случае `from os.path import *` будет импортировано почти 40 имен, включая имена `dirname`, `exists` и `split`, которые, вполне вероятно, мы могли бы использовать в качестве имен для наших собственных переменных или функций.

Например, если записать инструкцию `from os.path import dirname`, мы получим удобную возможность вызывать функцию `dirname()`, не указывая полное квалифицированное имя. Но если ниже в нашем программном коде будет встречена инструкция `dirname = '.'`, то после ее выполнения ссылка на объект `dirname` будет указывать уже не на функцию `dirname()`, а на строку `'.'`. Поэтому, если мы попытаемся вызвать функцию `dirname()`, мы получим исключение `TypeError`, потому что теперь имя `dirname` ссылается на строку, а не на вызываемый объект.

Ввиду того, что синтаксис `import *` потенциально опасен появлением конфликтов имен, некоторые коллективы разработчиков вырабатывают свои правила, устанавливающие, что в их разработках может использоваться только синтаксис `import` *`importable`*. Однако некоторые крупные пакеты, в частности библиотеки GUI (Graphical User Interface – графический интерфейс пользователя), нередко импортируются таким способом, потому что они включают огромное число функций и классов (собственных типов данных), для которых было бы слишком утомительно вводить вручную полные имена.

Возникает естественный вопрос – как интерпретатор узнает, где искать импортируемые модули и пакеты? Встроенный модуль `sys` имеет список с именем `sys.path`, в котором хранится перечень каталогов, составляющих *путь поиска Python*. Первый каталог в этом списке – это каталог, где находится сама программа, даже если она вызывается из другого каталога. Далее в списке находятся пути к каталогам из переменной окружения `PYTHONPATH`, если она определена. И в конце списка находятся пути к каталогам стандартной библиотеки языка Python – они определяются на этапе установки Python.

Когда модуль импортируется впервые, если он не является встроенным, интерпретатор пытается отыскать его поочередно в каждом из каталогов, перечисленных в списке `sys.path`. Как следствие этого, если мы создаем модуль или программу, имя которого совпадает с именем библиотечного модуля, наш модуль будет найден первым, что неизбежно будет приводить к проблемам. Чтобы избежать этого, никогда не создавайте программы или модули, имена которых совпадают с именами модулей или каталогов верхнего уровня в библиотеке, если только вы не пытаетесь подставить свою собственную реализацию и ваше переопределение преднамеренно. (Модулем верх-

него уровня называется файл *.py*, который находится в одном из каталогов, включенных в путь поиска Python, а не в каком-нибудь подкаталоге, вложенном в один из этих каталогов.) Например, в системе Windows в путь поиска Python обычно включается каталог с именем *C:\Python30\Lib*, поэтому на этой платформе мы не должны создавать модуль с именем *Lib.py*, так же как модуль, имя которого совпадает с именем любого модуля из каталога *C:\Python30\Lib*.

Один из способов быстро проверить, используется ли то или иное имя модуля, состоит в том, чтобы попытаться импортировать модуль. Сделать это можно в консоли, вызвав интерпретатор с ключом `-c` («execute code» – выполнить программный код), за которым следует указать инструкцию `import`. Например, если необходимо проверить, существует ли модуль с именем `Music.py` (или каталог верхнего уровня *Music* в пути поиска Python), можно ввести в консоли следующую команду:

```
python -c "import Music"
```

Если в ответ будет получено исключение `ImportError`, можно быть уверенным, что модуль или каталог верхнего уровня с таким именем не используется; любой другой вывод (или его отсутствие) означает наличие такого имени. К сожалению, такой прием не дает полной гарантии, что впоследствии с этим именем не будет возникать никаких проблем, поскольку позднее мы можем установить пакет или модуль, созданный сторонним разработчиком, имеющий такое же имя, хотя на практике такая проблема возникает достаточно редко.

Например, если мы создадим модуль *os.py*, он будет конфликтовать с библиотечным модулем `os`. Но если мы создадим модуль *path.py*, то никаких проблем возникать не будет, поскольку этот модуль пришлось бы импортировать как модуль `path`, тогда как библиотечный модуль должен импортироваться как `os.path`. В этой книге имена файлов наших собственных модулей всегда будут начинаться с символа верхнего регистра; это позволит избежать конфликтов имен (по крайней мере в UNIX), потому что имена файлов библиотечных модулей состоят исключительно из символов нижнего регистра.

Программа может импортировать некоторые модули, которые в свою очередь импортируют другие модули, включая те, что уже были импортированы. Это не является проблемой. Всякий раз, когда выполняется попытка импортировать модуль, интерпретатор Python сначала проверяет – не был ли импортирован требуемый модуль ранее. Если модуль еще не был импортирован, Python выполняет скомпилированный байт-код модуля, создавая тем самым переменные, функции и другие объекты модуля, после чего добавляет во внутреннюю структуру запись о том, что модуль был импортирован. При любых последующих попытках импортировать этот модуль интерпретатор будет обнаруживать, что модуль уже импортирован и не будет выполнять никаких действий.

Когда интерпретатору требуется скомпилированный байт-код модуля, он генерирует его автоматически – этим Python отличается от таких языков программирования, как Java, где компилирование в байт-код должно выполняться явно. Сначала интерпретатор попытается отыскать файл, имя которого совпадает с именем файла, имеющего расширение *.py*, но имеющий расширение *.pyo* – это оптимизированный байт-код скомпилированной версии модуля. Если файл с расширением *.pyo* не будет найден (или он более старый, чем файл с расширением *.py*), интерпретатор попытается отыскать одноименный файл с расширением *.pyc* – это неоптимизированный байт-код скомпилированной версии модуля. Если интерпретатор обнаружит актуальную скомпилированную версию модуля, он загрузит ее; в противном случае Python загрузит файл с расширением *.py* и скомпилирует его в байт-код. В любом случае интерпретатор загрузит в память модуль в виде скомпилированного байт-кода.

Если интерпретатор выполнил компиляцию файла с расширением *.py*, он сохранит скомпилированную версию в одноименном файле с расширением *.pyc* (или *.pyo*, если интерпретатор был запущен с ключом командной строки `-O`[1], или если в переменной окружения `PYTHONOPTIMIZE` установлено значение `0`), при этом каталог должен быть доступен для записи. Сохранения байт-кода можно избежать, если запускать интерпретатор с ключом командной строки `-B` или установив переменную окружения `PYTHONDONTWRITEBYTECODE`.

Использование файлов со скомпилированным байт-кодом ускоряет запуск программы, поскольку интерпретатору остается только загрузить и выполнить программный код, минуя этап компиляции (и сохранения, если это возможно), хотя сама скорость работы программы от этого не зависит. При установке Python компиляция модулей стандартной библиотеки в байт-код обычно является частью процесса установки.

Пакеты

Пакет – это простой каталог, содержащий множество модулей и файл с именем *__init__.py*. Например, допустим, что у нас имеется некоторое множество файлов модулей, предназначенных для чтения и записи графических файлов различных форматов с именами *Bmp.py*, *Jpeg.py*, *Png.py*, *Tiff.py* и *Xpm.py*, в каждом из которых имеются функции `load()`, `save()` и т. д.[2] Мы могли бы сохранить все эти модули в одном каталоге с программой, но в крупных программных продуктах, ис-

[1] Это символ «O», а не цифра 0. – *Прим. перев.*

[2] Широкая поддержка операций с графическими файлами обеспечивается различными модулями сторонних разработчиков, из которых наиболее примечательной является библиотека Python Imaging Library (*www.pythonware.com/products/pil*).

пользующих массу собственных модулей, модули для работы с графикой, скорее всего, лучше хранить отдельно. Поместив их в свой собственный подкаталог, например *Graphics*, их можно хранить все вместе. А если поместить в каталог *Graphics* пустой файл *__init__.py*, этот каталог превратится в пакет:

```
Graphics/
    __init__.py
    Bmp.py
    Jpeg.py
    Png.py
    Tiff.py
    Xpm.py
```

Пока каталог *Graphics* является подкаталогом каталога с программой или находится в пути поиска Python, мы будем иметь возможность импортировать любой из этих модулей и использовать их. Мы должны сделать все возможное, чтобы гарантировать несовпадение имени нашего модуля верхнего уровня (`Graphics`) с каким-либо из имен верхнего уровня в стандартной библиотеке – с целью избежать конфликтов имен. (В системе UNIX это легко обеспечить, достаточно лишь использовать в качестве первого символа имени символ верхнего регистра, так как в именах модулей стандартной библиотеки используются только символы нижнего регистра.) Ниже показано, как импортировать и использовать наши модули:

```
import Graphics.Bmp
image = Graphics.Bmp.load("bashful.bmp")
```

В небольших программах некоторые программисты предпочитают использовать более короткие имена, и язык Python позволяет делать это двумя, немного отличающимися способами.

```
import Graphics.Jpeg as Jpeg
image = Jpeg.load("doc.jpeg")
```

Здесь мы импортировали модуль `Jpeg` из пакета `Graphics` и сообщили интерпретатору, что вместо полного квалифицированного имени `Graphics.Jpeg` хотим использовать более короткое имя `Jpeg`.

```
from Graphics import Png
image = Png.load("dopey.png")
```

Этот фрагмент программного кода напрямую импортирует модуль `Png` из пакета `Graphics`. Данная синтаксическая конструкция (`import ... from`) обеспечивает непосредственный доступ к модулю `Png`.

Мы не обязаны использовать в нашем программном коде оригинальные имена модулей. Например:

```
from Graphics import Tiff as picture
image = picture.load("grumpy.tiff")
```

Здесь мы используем модуль `Tiff`, но внутри нашей программы переименовали его в модуль `picture`.

В некоторых ситуациях бывает удобно загружать все модули пакета одной инструкцией. Для этого необходимо отредактировать файл *__init__.py* пакета, записав в него инструкцию, которая указывала бы, какие модули должны загружаться. Эта инструкция должна присваивать список с именами модулей специальной переменной `__all__`. Например, ниже приводится необходимая строка для файла *Graphics/__init__.py*:

```
__all__ = ["Bmp", "Jpeg", "Png", "Tiff", "Xpm"]
```

Этим ограничивается необходимое содержимое файла *__init__.py*, помимо этого, мы можем поместить в него любой программный код, какой только пожелаем. Теперь мы можем использовать другую разновидность инструкции `import`:

```
from Graphics import *
image = Xpm.load("sleepy.xpm")
```

Синтаксис `from` *package* `import *` напрямую импортирует все имена модулей, упомянутые в списке `__all__`. То есть после выполнения этой инструкции мы получим прямой доступ не только к модулю `Xpm`, но и ко всем другим модулям.

Как отмечалось ранее, этот синтаксис может применяться и к модулям, то есть `from` *module* `import *`, в этом случае будут импортированы все функции, переменные и другие объекты, определяемые модулем (за исключением тех, чьи имена начинаются с символа подчеркивания). При необходимости точно указать, что должно быть импортировано при использовании синтаксической конструкции `from` *module* `import *`, мы можем определить список `__all__` непосредственно в модуле; в этом случае инструкция `from` *module* `import *` будет импортировать только те объекты, имена которых присутствуют в списке `__all__`.

До сих пор мы демонстрировали только один уровень вложенности, но Python позволяет создавать столько уровней вложенности пакетов, сколько нам заблагорассудится. То есть мы можем поместить в каталог *Graphics* подкаталог, скажем, *Vector*, с файлами модулей внутри него *Eps.py* и *Svg.py*:

```
Graphics/
    __init__.py
    Bmp.py
    Jpeg.py
    Png.py
    Tiff.py
    Vector/
        __init__.py
        Eps.py
```

```
        Svg.py
    Xpm.py
```

Чтобы каталог *Vector* превратился в пакет, в него необходимо поместить файл *__init__.py*, который, как уже говорилось, может быть пустым или определять список `__all__` для обеспечения удобства тем программистам, которые предпочтут использовать инструкцию импортирования `from Graphics.Vector import *`.

Для доступа к вложенному пакету мы можем использовать обычный синтаксис, который использовали ранее:

```
import Graphics.Vector.Eps
image = Graphics.Vector.Eps.load("sneezy.eps")
```

Полные квалифицированные имена могут оказаться слишком длинными, поэтому некоторые программисты пытаются привести иерархию модулей к плоскому виду, чтобы избежать необходимости вручную вводить такие имена:

```
import Graphics.Vector.Svg as Svg
image = Svg.load("snow.svg")
```

Мы всегда можем использовать свои собственные короткие имена для модулей, как показано в этом примере, хотя это повышает риск появления конфликтов имен.

Собственные модули

Поскольку модули – это всего лишь файлы с расширением *.py*, они создаются без особых формальностей. В этом разделе мы рассмотрим два нестандартных модуля. Первый модуль, `TextUtil` (в файле *TextUtil.py*), содержит всего три функции: `is_balanced()`, возвращающую `True`, если в строке, переданной ей, соблюдена парность скобок разных типов; `shorten()` (продемонстрированную ранее, на стр. 209) и `simplify()`, способную удалять лишние пробелы и другие символы из строки. При рассмотрении этого модуля мы также покажем, как использовать программный код в строках документирования в качестве модульных тестов.

Второй модуль, `CharGrid` (в файле *CharGrid.py*), содержит сетку символов и позволяет «рисовать» линии, прямоугольники и текст в сетке и отображать сетку в консоли. Этот модуль демонстрирует некоторые приемы, с которыми мы не сталкивались ранее, и является более типичным примером более крупных и более сложных модулей.

Модуль TextUtil

Структура этого модуля (и большинства других модулей) немного отличается от структуры программы. Первая строка модуля – это строка «shebang», вслед за которой следует несколько строк комментариев (обычно упоминание об авторских правах и информация о лицензионном соглашении). Затем, как правило, следует строка в тройных ка-

вычках, в которой дается краткий обзор содержимого модуля, часто с несколькими примерами использования – это строка документирования модуля. Ниже приводится начало файла *TextUtil.py* (правда, без комментария с упоминанием о лицензионном соглашении):

```
#!/usr/bin/env python3
# Copyright (c) 2008 Qtrac Ltd. All rights reserved.
"""
Этот модуль предоставляет несколько функций манипулирования строками.
>>> is_balanced("(Python (is (not (lisp))))")
True
>>> shorten("The Crossing", 10)
'The Cro...'
>>> simplify(" some text with spurious whitespace ")
'some text with spurious whitespace'
"""

import string
```

Строку документирования этого модуля можно сделать доступной программам (или другим модулям), если импортировать модуль как `TextUtil.__doc__`. Вслед за строкой документирования следуют инструкции импортирования, в данном случае – единственная инструкция, и далее находится остальная часть модуля.

Функция shorten(), стр. 209

Мы уже видели полный текст функции `shorten()`, поэтому не будем повторно воспроизводить его здесь. И поскольку в настоящее время нас интересуют модули, а не функции, мы продемонстрируем только программный код функции `is_balanced()`, хотя функцию `simplify()` приведем полностью, вместе со строкой документирования.

Ниже приводится функция `simplify()`, разбитая на две части:

```
def simplify(text, whitespace=string.whitespace, delete=""):
    r"""Возвращает текст, из которого удалены лишние пробелы.

    Параметр whitespace - это строка символов, каждый из которых
    считается символом пробела.Если параметр delete не пустой,
    он должен содержать строку, и тогда все символы, входящие
    в состав строки delete, будут удалены из строки результата.

    >>> simplify(" this and\n that\t too")
    'this and that too'
    >>> simplify(" Washington D.C.\n")
    'Washington D.C.'
    >>> simplify(" Washington D.C.\n", delete=",;:.")
    'Washington DC'
    >>> simplify(" disemvoweled ", delete="aeiou")
    'dsmvwld'
    """
```

Вслед за строкой с инструкцией def следует строка документирования функции, первая строка которой в соответствии с соглашениями является коротким однострочным описанием; за ней следуют пустая строка, более подробное описание и затем несколько примеров, записанных так, как если бы они выполнялись в интерактивной оболочке. Поскольку в строке документирования присутствуют кавычки, мы должны либо экранировать их символом обратного слеша, либо, как в данном случае, использовать «сырую» строку в тройных кавычках.

«Сырые» строки, стр. 85

```
    result = []
    word = ""
    for char in text:
        if char in delete:
            continue
        elif char in whitespace:
            if word:
                result.append(word)
                word = ""
        else:
            word += char
    if word:
        result.append(word)
    return " ".join(result)
```

Список result используется для хранения «слов» – строк, не имеющих пробельных или удаляемых символов. Внутри функции выполняются итерации по символам в параметре text, с пропуском удаляемых символов. Если встречается пробельный символ и в переменной word содержится хотя бы один символ, полученное слово добавляется в список result, после чего в переменную word записывается пустая строка; в противном случае пробельный символ пропускается. Любые другие символы добавляются к создаваемому слову. В конце функция возвращает единственную строку, содержащую все слова из списка result, разделенные пробелом.

Функция is_balanced() следует тому же шаблону: за строкой с инструкцией def находится строка документирования с коротким однострочным описанием, пустой строкой, полным описанием и несколькими примерами, вслед за которой идет сам программный код. Ниже приводится только программный код функции, без строки документирования:

```
def is_balanced(text, brackets="()[]{}<>"):
    counts = {}
    left_for_right = {}
    for left, right in zip(brackets[::2], brackets[1::2]):
        assert left != right, "the bracket characters must differ"
        counts[left] = 0
```

```
            left_for_right[right] = left
    for c in text:
        if c in counts:
            counts[c] += 1
        elif c in left_for_right:
            left = left_for_right[c]
            if counts[left] == 0:
                return False
            counts[left] -= 1
    return not any(counts.values())
```

Функция создает два словаря. Ключами словаря `counts` являются символы открывающих скобок («`(`», «`[`», «`{`» и «`<`»), а значениями – целые числа. Ключами словаря `left_for_right` являются символы закрывающих скобок («`)`», «`]`», «`}`» и «`>`»), а значениями – соответствующие им символы открывающих скобок. Сразу после создания словарей функция начинает выполнять итерации по символам в параметре `text`. Всякий раз, когда встречается символ открывающей скобки, соответствующее ему значение в словаре `count` увеличивается на 1. Точно так же, когда встречается символ закрывающей скобки, функция определяет соответствующий ему символ открывающей скобки. Если счетчик для этого символа равен 0, это означает, что была встречена лишняя закрывающая скобка, поэтому можно сразу же возвращать `False`; в противном случае счетчик уменьшается на 1. По окончании просмотра текста, если все открывающие скобки имеют парные им закрывающие скобки, все счетчики должны быть равны 0, поэтому, если хотя бы один счетчик не равен 0, функция возвращает `False`; в противном случае она возвращает `True`.

До этого момента рассматриваемый модуль ничем не отличался от любого другого файла с расширением *.py*. Если бы файл *TextUtil.py* был программой, вполне возможно, что в нем присутствовали бы и другие функции, а в конце стоял бы единственный вызов одной из этих функций, запускающий обработку. Но так как это модуль, который предназначен для того, чтобы его импортировали, одних определений функций вполне достаточно. Теперь любая программа или модуль смогут импортировать модуль `TextUtil` и использовать его:

```
import TextUtil

text = " a puzzling conundrum "
text = TextUtil.simplify(text) # text == 'a puzzling conundrum'
```

Если нам потребуется сделать модуль `TextUtil` доступным определенной программе, нам достаточно будет поместить файл *TextUtil.py* в один каталог с программой. Сделать файл *TextUtil.py* доступным для всех наших программ можно несколькими способами. Первый состоит в том, чтобы поместить модуль в подкаталог *site-packages*, находящийся в дереве каталогов, куда был установлен Python (в системе Windows это обычно каталог *C:\Python30\Lib\site-packages*, но в Mac OS X и других

версиях UNIX путь к этому каталогу будет иным). Данный каталог находится в пути поиска Python, поэтому интерпретатор всегда будет отыскивать любые модули, находящиеся здесь. Второй способ заключается в создании каталога, специально предназначенного для наших собственных модулей, которые мы предполагаем использовать в наших программах, и добавлении пути к этому каталогу в переменную окружения `PYTHONPATH`. Третий способ состоит в том, чтобы поместить модуль в *локальный* подкаталог *site-packages* – каталог *%APPDATA%/Python/Python30/site-packages* в Windows, и *~/.local/lib/python3.0/site-packages* в UNIX (включая Mac OS X), который находится в пути поиска Python. Второй и третий подходы предпочтительнее, так как в этих двух случаях ваш программный код будет храниться отдельно от официальной версии Python.

Иметь модуль `TextUtil` само по себе уже неплохо, но если в конечном счете предполагается использовать его во множестве программ, то наверняка хотелось бы пребывать в уверенности, что он работает именно так, как заявлено. Один из самых простых способов состоит в том, чтобы выполнить примеры, которые приводятся в строках документирования, и убедиться, что они дают ожидаемые результаты. Сделать это можно, добавив всего три строки в конец файла модуля:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Всякий раз, когда выполняется импортирование модуля, интерпретатор создает для него переменную с именем `__name__` и сохраняет имя модуля в этой переменной. Имя модуля – это просто имя файла *.py*, только без расширения. Поэтому в данном случае, когда модуль будет импортироваться, переменная `__name__` получит значение `"TextUtil"` и условие в инструкции `if` не будет соответствовать `True`, то есть две последние строки выполняться не будут. Это означает, что последние три строки ничего не меняют, когда модуль импортируется.

Всякий раз, когда файл с расширением *.py* запускается как программа, интерпретатор Python создает в программе переменную с именем `__name__` и записывает в нее строку `"__main__"`. То есть, если мы *запустим* файл *TextUtil.py* как программу, интерпретатор запишет в переменную `__name__` строку `"__main__"`, условие в инструкции `if` вернет `True` и две последние строки будут выполнены.

Функция `doctest.testmod()` с помощью механизма интроспекции Python выявляет все функции в модуле и их строки документирования, после чего пытается выполнить все фрагменты программного кода, которые приводятся в строках документирования. При запуске модуля таким способом вывод на экране появится только при наличии ошибок. Сначала это может привести в замешательство, так как создается впечатление, будто вообще ничего не происходит; но если интерпретатору

передать ключ командной строки -v, на экране может появиться примерно следующее:

```
Trying:
    is_balanced("(Python (is (not (lisp))))")
Expecting:
    True
ok
...
Trying:
    simplify(" disemvoweled ", delete="aeiou")
Expecting:
    'dsmvwld'
ok
4 items passed all tests:
    3 tests in __main__
    5 tests in __main__.is_balanced
    3 tests in __main__.shorten
    4 tests in __main__.simplify
15 tests in 4 items.
15 passed and 0 failed.
Test passed.
```

Мы использовали многоточия, чтобы показать, что было опущено множество строк. Если в модуле имеются функции (или классы, или методы), не имеющие тестов, при запуске интерпретатора с ключом -v они будут перечислены. Обратите внимание, что модуль doctest обнаружил тесты как в строке документирования модуля, так и в строках документирования функций.

Примеры в строках документирования, которые могут выполняться как тесты, называют *доктестами* (*doctests*). Обратите внимание, что при написании доктестов мы вызываем функцию simplify(), не используя полное квалифицированное имя (поскольку доктесты находятся непосредственно в самом модуле). За пределами модуля, после выполнения инструкции import TextUtil, мы должны использовать квалифицированные имена, например, TextUtil.is_balanced().

В следующем подразделе мы увидим, как реализовать более полноценные тесты – в частности, проверку случаев, когда ожидаются отказы, – например, когда неверные входные данные должны приводить к возбуждению исключения. Мы также рассмотрим некоторые другие проблемы, связанные с созданием модулей, включая инициализацию модуля, учет различий между платформами и обеспечение возможности импортировать программами или модулями, при использовании синтаксиса from *module* import *, только тех объектов, которые мы хотим сделать общедоступными.

Модуль CharGrid

Модуль `CharGrid` хранит в памяти сетку символов. Он предоставляет функции, позволяющие «рисовать» в сетке линии, прямоугольники и текст, а также функции отображения сетки в консоли. Ниже приводятся доктесты из строки документирования модуля:

```
>>> resize(14, 50)
>>> add_rectangle(0, 0, *get_size())
>>> add_vertical_line(5, 10, 13)
>>> add_vertical_line(2, 9, 12, "!")
>>> add_horizontal_line(3, 10, 20, "+")
>>> add_rectangle(0, 0, 5, 5, "%")
>>> add_rectangle(5, 7, 12, 40, "#", True)
>>> add_rectangle(7, 9, 10, 38, " ")
>>> add_text(8, 10, "This is the CharGrid module")
>>> add_text(1, 32, "Pleasantville", "@")
>>> add_rectangle(6, 42, 11, 46, fill=True)
>>> render(False)
```

Функция `CharGrid.add_rectangle()` принимает четыре обязательных аргумента: номер строки и номер столбца верхнего левого угла, а также номер строки и номер столбца правого нижнего угла. В пятом необязательном аргументе можно определить символ, который будет использоваться для рисования сторон прямоугольника, а в шестом аргументе типа Boolean можно указать, следует ли выполнять заливку прямоугольника (тем же самым символом, который используется для рисования сторон). В первом вызове третий и четвертый аргументы передаются путем распаковывания двухэлементного кортежа (ширина и высота), который возвращает функция `CharGrid.get_size()`.

По умолчанию, прежде чем вывести содержимое сетки, функция `CharGrid.render()` очищает экран, но чтобы предотвратить это, ей можно передать значение `False`, что и было сделано в данном случае. Ниже приводится изображение сетки, полученной в результате выполнения доктестов:

```
%%%%%**********************************************
%   %                           @@@@@@@@@@@@@@@@ *
%   %                           @Pleasantville@ *
%   %     ++++++++++            @@@@@@@@@@@@@@@@ *
%%%%%                                            *
*     ##################################         *
*     ##################################   ****  *
*     ##                               ##  ****  *
*     ## This is the CharGrid module ##  ****  *
* !   ##                               ##  ****  *
* ! | ##################################  ****  *
* ! | ##################################         *
*   |                                            *
**********************************************
```

Модуль CharGrid начинается точно так же, как и модуль TextUtil — со строки «shebang», с упоминания об авторских правах и лицензионном соглашении. В строке документирования модуля приводится его описание, вслед за которым находятся доктесты, упомянутые выше. Следующий ниже программный код начинается двумя инструкциями импорта: одна импортирует модуль sys, а другая — модуль subprocess. Модуль subprocess подробно будет рассматриваться в главе 9.

В модуле используется две тактики обработки ошибок. Некоторые функции имеют параметр типа char, то есть фактически строку, содержащую единственный символ. Нарушение этого требования рассматривается как фатальная ошибка программирования, поэтому для проверки длины аргументов используется инструкция assert. Передача номеров строк и столбцов со значениями, выходящими за пределы сетки, хотя и считается ошибкой, но рассматривается как нормальная ситуация, поэтому в подобных случаях возбуждается наше собственное исключение.

Теперь мы рассмотрим наиболее показательные и наиболее важные фрагменты программного кода модуля, начав с исключений:

```
class RangeError(Exception): pass
class RowRangeError(RangeError): pass
class ColumnRangeError(RangeError): pass
```

Ни одна из функций в модуле, возбуждающих исключения, не возбуждает исключение RangeError, они всегда возбуждают конкретное исключение в зависимости от того, номер строки или столбца вышел за пределы сетки. Но, используя существующую иерархию исключений, мы даем пользователю модуля возможность выбирать, будет ли он обрабатывать конкретные исключения или перехватывать их по базовому классу RangeError. Обратите также внимание на то, что внутри доктестов используются неквалифицированные имена исключений, но когда модуль импортируется инструкцией import CharGrid, необходимо использовать полные квалифицированные имена исключений: CharGrid.RangeError, CharGrid.RowRangeError и CharGrid.ColumnRangeError.

```
_CHAR_ASSERT_TEMPLATE = ("char must be a single character: '{0}' "
                         "is too long")
_max_rows = 25
_max_columns = 80
_grid = []
_background_char = " "
```

Здесь определяются некоторые частные данные для использования внутри модуля. Имена частных переменных начинаются с символа подчеркивания, поэтому, когда модуль будет импортироваться инструкцией from CharGrid import *, ни одна из этих переменных не будет импортирована. (Как вариант, можно было бы использовать список __all__.) Переменная _CHAR_ASSERT_TEMPLATE — это строка, предназначенная для вызова метода str.format(), — позднее мы увидим, что такой

прием широко используется для генерации сообщений об ошибках в инструкциях assert. Назначение остальных переменных будет поясняться по мере того, как мы будем сталкиваться с ними.

```
if sys.platform.startswith("win"):
    def clear_screen():
        subprocess.call(["cmd.exe", "/C", "cls"])
else:
    def clear_screen():
        subprocess.call(["clear"])
clear_screen.__doc__ = """Clears the screen using the underlying \
window system's clear screen command"""
```

Очистка экрана консоли в разных системах выполняется по-разному. В Windows необходимо выполнить программу cmd.exe с соответствующими аргументами, а в большинстве систем UNIX запускается программа clear. Функция subprocess.call() из модуля subprocess позволяет запускать внешние программы, поэтому мы можем использовать ее для очистки экрана с учетом особенностей системы. Строка sys.platform хранит имя операционной системы, под управлением которой выполняется программа, например, «win32» или «linux2». Поэтому один из способов учесть различия между платформами – определить функцию clear_screen(), как показано ниже:

```
def clear_screen():
    command = (["clear"] if not sys.platform.startswith("win") else
               ["cmd.exe", "/C", "cls"])
    subprocess.call(command)
```

Недостаток такого подхода заключается в том, что, даже зная, что тип платформы не изменится в процессе работы программы, мы все равно вынуждены выполнять проверку при каждом вызове функции.

Чтобы избежать необходимости проверки типа операционной системы, под управлением которой выполняется программа, при каждом вызове функции clear_screen(), мы создаем платформозависимую функцию clear_screen() на этапе импортирования модуля и с этого момента постоянно используем ее. Это возможно благодаря тому, что в языке Python инструкция def является самой обычной инструкцией – когда интерпретатор достигает условной инструкции if, он выполняет либо первую, либо вторую инструкцию def, *динамически* создавая ту или иную версию функции clear_screen(). Так как определение функции находится за пределами какой-либо другой функции (или класса, о чем будет рассказываться в следующей главе), она по-прежнему остается в глобальной области видимости и обращаться к ней можно так же, как к любой другой функции в модуле.

После создания функции мы явно определяем строку документирования для нее – такой прием позволяет избежать необходимости дважды записывать одну и ту же строку документирования в двух местах, а, кроме того, иллюстрирует, что строка документирования – это всего

лишь атрибут функции. В число прочих атрибутов входят имя модуля функции и ее собственное имя.

```
def resize(max_rows, max_columns, char=None):
    """Изменяет размер сетки, очищает содержимое и изменяет символ фона,
    если аргумент char не равен None
    """
    assert max_rows > 0 and max_columns > 0, "too small"
    global _grid, _max_rows, _max_columns, _background_char
    if char is not None:
        assert len(char) == 1, _CHAR_ASSERT_TEMPLATE.format(char)
        _background_char = char
    _max_rows = max_rows
    _max_columns = max_columns
    _grid = [[_background_char for column in range(_max_columns)]
             for row in range(_max_rows)]
```

Эта функция использует инструкцию `assert` для обеспечения политики выявления ошибок программирования; в первом случае – ошибки при попытке установить размеры сетки меньше, чем 1×1. Если символ фона определен, применяется еще одна инструкция `assert`, чтобы гарантировать, что эта строка содержит точно один символ; в противном случае возбуждается исключение с текстом сообщения из шаблона `_CHAR_ASSERT_TEMPLATE`, в котором поле {0} замещается полученной строкой `char`.

К сожалению, мы вынуждены использовать инструкцию `global`, потому что внутри этой функции приходится изменять глобальные переменные. Это как раз тот случай, когда на помощь может прийти объектно-ориентированное программирование, с которым мы познакомимся в главе 6.

Генераторы списков, стр. 142

Содержимое для переменной `_grid` создается с помощью двух вложенных друг в друга генераторов списков. Прием с применением оператора дублирования списка, такой как `[[char] * columns] * rows`, не даст должного результата, потому что внутренние списки будут представлять собой всего лишь поверхностные копии одного и того же списка. Вместо генераторов списков можно было бы использовать вложенные циклы `for ... in`:

```
_grid = []
for row in range(_max_rows):
    _grid.append([])
    for column in range(_max_columns):
        _grid[-1].append(_background_char)
```

Но такой фрагмент сложнее для понимания и гораздо длиннее, чем генераторы списков.

Поскольку основной целью нашего рассмотрения является реализация модуля, мы рассмотрим лишь одну функцию рисования, чтобы

получить представление о том, как это рисование выполняется. Ниже приводится функция add_horizontal_line(), поделенная на две части:

```
def add_horizontal_line(row, column0, column1, char="-"):
    """Добавляет в сетку горизонтальную линию, используя указанный символ

    >>> add_horizontal_line(8, 20, 25, "=")
    >>> char_at(8, 20) == char_at(8, 24) == "="
    True
    >>> add_horizontal_line(31, 11, 12)
    Traceback (most recent call last):
    ...
    RowRangeError
    """
```

Строка документирования содержит два теста, один из которых, как предполагается, будет проходить успешно, а другой будет возбуждать исключение. Задавая в доктестах исключения, необходимо вставлять строку «Traceback»; она всегда одна и та же и сообщает модулю doctest, что ожидается исключение. Затем взамен строк с диагностическими сообщениями (количество которых может быть разным) следует указать многоточие и завершить тест строкой с именем исключения, которое ожидается получить. Функция char_at() — одна из тех, что предоставляется этим модулем; она возвращает символ в заданной позиции строки и столбца сетки.

```
    assert len(char) == 1, _CHAR_ASSERT_TEMPLATE.format(char)
    try:
        for column in range(column0, column1):
            _grid[row][column] = char
    except IndexError:
        if not 0 <= row <= _max_rows:
            raise RowRangeError()
        raise ColumnRangeError()
```

Реализация функции начинается с той же проверки длины аргумента char, которая производилась и в функции resize(). Вместо того чтобы явно проверять аргументы с номерами строки и столбцов, функция работает в предположении, что аргументы имеют допустимые значения. Если из-за обращения к несуществующей строке или столбцу возбуждается исключение IndexError, функция перехватывает его и возбуждает соответствующее исключение, характерное для модуля. Такой стиль программирования соответствует выражению «проще попросить прощения, чем разрешения» и считается более свойственным программированию на языке Python, чем стиль «осмотрись, прежде чем прыгнуть», при котором проверки выполняются заранее. Реализация с опорой на исключения вместо предварительной проверки является более эффективной, когда исключения возникают достаточно редко. (Контрольные инструкции assert мы не относим к стилю «осмотрись, прежде чем прыгнуть», потому что такие ошибки никогда не

должны возникать и они часто убираются из окончательной версии программного кода.)

Практически в самом конце модуля, после определения всех функций, имеется единственный вызов функции `resize()`:

```
resize(_max_rows, _max_columns)
```

Этот вызов инициализирует сетку с размерами по умолчанию (25×80), чем обеспечивает безопасное использование модуля в импортирующем программном коде. Без этого вызова импортирующая программа или модуль должны были бы явно вызывать функцию `resize()` для инициализации сетки, что вынуждало бы программистов помнить об этом факте и приводило бы к множественным попыткам инициализации.

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Последние три строки в модуле являются обычными для модулей, использующих модуль `doctest` для выполнения доктестов.

Модуль `CharGrid` имеет один существенный недостаток: он поддерживает только одну сетку символов. Одно из решений, избавляющих от этого недостатка, заключается в создании коллекции сеток, но это попутно означает, что пользователи модуля должны были бы указывать ключ или индекс требуемой сетки во всех вызовах функций, чтобы идентифицировать сетку, над которой выполняется операция. В случаях, когда возникает необходимость иметь несколько экземпляров объекта, лучшее решение состоит в том, чтобы определить класс (собственный тип данных) и создать столько экземпляров (объектов данного типа), сколько потребуется. Дополнительное преимущество реализации на основе класса заключается в том, что мы можем отказаться от использования инструкции `global`, сохраняя данные в виде атрибутов (статических элементов) класса. Как создавать классы, мы узнаем в следующей главе.

Обзор стандартной библиотеки языка Python

Стандартная библиотека языка Python обычно описывается, как «батарейки, входящие в комплект поставки», и обеспечивает доступ к широкому кругу функциональных возможностей, насчитывая в своем составе свыше 200 пакетов и модулей. Этот раздел представляет собой краткий обзор того, что может предложить стандартная библиотека, разделенный на тематические подразделы; из обзора исключены пакеты и модули, представляющие слишком узконаправленный интерес, а также модули, характерные для той или иной платформы. В ходе описания будут демонстрироваться небольшие примеры, чтобы дать представление, что представляют собой те или иные пакеты и модули,

а перекрестные ссылки будут указывать страницы в книге, где можно найти дополнительные сведения об этих пакетах и модулях.

Обработка строк

Модуль `string` содержит ряд полезных констант, таких как `string.ascii_letters` и `string.hexdigits`. Кроме того, он предоставляет класс `string.Formatter`, на основе которого можно создать подкласс, обеспечивающий собственные средства форматирования.[1] Модуль `textwrap` может использоваться для расстановки в тексте символов перевода строки, чтобы ограничить ширину строк заданным значением, а также для уменьшения отступов.

Модуль `struct` содержит функции упаковывания и распаковывания чисел, логических значений и строк в/из объекты типа `bytes`, используя их двоичное представление. Это может потребоваться для организации передачи данных между программой и низкоуровневыми библиотеками, написанными на языке C. Модули `struct` и `textwrap` используются программой *convert-incidents.py*, описываемой в главе 7.

Тип данных bytes, стр. 344

Модуль struct, стр. 349

Модуль `difflib` содержит классы и методы сравнения последовательностей, таких как строки, способные воспроизводить результаты сравнения как в стандартных форматах «diff», так и в формате HTML.

Самый мощный модуль в языке Python, связанный с обработкой строк, – это модуль `re` (regular expression – регулярные выражения). Он подробно будет рассматриваться в главе 12.

Класс `io.StringIO` может использоваться для создания объектов, подобных строкам, которые ведут себя как текстовые файлы, размещенные в памяти. Это может быть удобно, когда необходимо использовать программный код, выполняющий запись в файл, для записи в строку.

Пример: класс io.StringIO

Выполнить запись в текстовый файл в языке Python можно разными способами. Один из способов состоит в использовании метода `write()` объекта файла, другой – в использовании функции `print()` с именованным аргументом `file`, указывающим на объект файла, открытый для записи. Например:

```
print("An error message", file=sys.stdout)
sys.stdout.write("Another error message\n")
```

[1] Создание подкласса (*subclassing*), или специализация класса (*spezializing*), означает создание собственного типа данных (класса) на основе (на базе, *based on*) другого класса. Эта тема подробно рассматривается в главе 6.

В обоих случаях текст сообщения будет выведен в sys.stdout — объект файла, представляющий «стандартный поток вывода», который обычно связан с консолью и отличается от sys.stderr, «стандартного потока вывода сообщений об ошибках», только тем, что при работе с последним используется небуферизованный вывод. (Интерпретатор автоматически создает и открывает sys.stdin, sys.stdout и sys.stderr при запуске программы.) По умолчанию функция print() добавляет символ перевода строки, хотя такое ее поведение можно изменить с помощью именованного аргумента end, передав в нем пустую строку.

В некоторых ситуациях бывает удобно перехватывать и записывать в строку все то, что предназначено для вывода в файл. Добиться этого можно с помощью класса io.StringIO, экземпляр которого можно использовать как обычный объект файла, который записывает данные в строку. Если при создании объекта io.StringIO была указана начальная строка, его также можно использовать для чтения, как если бы это был файл.

Выполнив инструкцию import io, мы сможем использовать io.StringIO для перехвата любой информации, предназначенной для вывода в объект файла, такой как sys.stdout:

```
sys.stdout = io.StringIO()
```

Если эту строку поместить в начало программы, вслед за инструкциями импорта, но перед любыми инструкциями, использующими sys.stdout, то любой текст, записываемый в sys.stdout, в действительности будет передаваться объекту io.StringIO, созданному этой строкой кода и заменившему стандартный объект файла sys.stdout. Теперь при выполнении приведенных выше строк с вызовами print() и sys.stdout.write() выводимый ими текст будет попадать в объект io.StringIO, а не на консоль. (Оригинальное значение sys.stdout можно восстановить в любой момент, для чего достаточно выполнить инструкцию sys.stdout = sys.__stdout__.)

Чтобы получить все строки, записанные в объект io.StringIO, можно вызвать метод io.StringIO.getvalue(). В данном случае вызовом метода sys.stdout.getvalue() можно получить строку, содержащую весь выводившийся текст. Эту строку можно напечатать, сохранить в файл журнала или отправить через сетевое соединение, как и любую другую строку. Немного ниже (на стр. 266) мы увидим еще один пример использования класса io.StringIO.

Работа с аргументами командной строки

Если нам потребуется иметь возможность в программе обрабатывать текст, который может быть получен в результате перенаправления в консоли или находиться в файлах, имена которых перечислены в командной строке, мы можем воспользоваться функцией fileinput.input() из модуля fileinput. Эта функция выполняет итерации по всем

строкам, полученным в результате операции перенаправления в консоли (если они есть), или по всем строкам из файлов, имена которых перечислены в командной строке, как будто это единая последовательность строк. Модуль может сообщать имя текущего файла и номер строки с помощью функций `fileinput.filename()` и `fileinput.lineno()`, а также предоставляет возможность работать с некоторыми типами сжатых файлов.

Для работы с параметрами командной строки в стандартной библиотеке имеется два модуля – `optparse` и `getopt`. Модуль `getopt` популярен, так как он прост в использовании и к тому же давно входит в состав библиотеки. Модуль `optparse` более новый и обладает более широкими возможностями.

Пример: модуль optparse

Пример csv2html.py, стр. 119

Вспомните описание программы *csv2html.py*, которое приводилось в главе 2. В упражнениях к этой главе мы предложили расширить программу так, чтобы она могла принимать аргументы командной строки: аргумент «maxwidth», принимающий целое число, и аргумент «format», принимающий строку. В решении (*csv2html2_ans.py*) для обработки аргументов имеется функция объемом 26 строк. Ниже приводится начало функции `main()` для *csv2html2_opt.py* – версии программы, в которой вместо нашей собственной функции для обработки аргументов командной строки используется модуль `optparse`:

```
def main():
    parser = optparse.OptionParser()
    parser.add_option("-w", "--maxwidth", dest="maxwidth", type="int",
            help=("the maximum number of characters that can be "
                  "output to string fields [default: %default]"))
    parser.add_option("-f", "--format", dest="format",
            help=("the format used for outputting numbers "
                  "[default: %default]"))
    parser.set_defaults(maxwidth=100, format=".0f")
    opts, args = parser.parse_args()
```

Для обработки аргументов потребовалось всего девять строк программного кода плюс строка с инструкцией `import optparse`. Кроме того, нам не пришлось явно обрабатывать параметры `-h` и `--help` – эти параметры обслуживаются самим модулем `optparse`, который для вывода соответствующего сообщения использует текст из именованных аргументов `help`, где текст «%default» замещается значениями по умолчанию соответствующих параметров.

Обратите также внимание, что теперь параметры можно указывать в привычном для системы UNIX стиле – как с помощью коротких, так с помощью длинных имен параметров, начинающихся с символа дефи-

са. Короткие имена удобны для организации взаимодействий с пользователем в консоли, а длинные имена более понятны при использовании в сценариях командной оболочки. Например, чтобы ограничить максимальную ширину 80 символами, мы можем использовать любой из следующих вариантов определения параметра: `-w80`, `-w 80`, `--maxwidth=80` или `--maxwidth 80`. После разбора параметров командной строки доступ к их значениям можно получить с помощью имен, указываемых в аргументах `dest`, например, `opts.maxwidth` и `opts.format`. Все аргументы командной строки, которые не были обработаны (обычно это имена файлов), помещаются в список `args`.

Если в процессе разбора командной строки возникает ошибка, синтаксический анализатор модуля `optparse` произведет вызов `sys.exit(2)`. Это приведет к завершению программы и возврату операционной системе числа 2 в качестве возвращаемого значения программы. Традиционно значение 2 свидетельствует об ошибке в использовании программы, значение 1 используется для индикации об ошибках любого другого типа и значение 0 означает благополучное завершение. Когда функция `sys.exit()` вызывается без аргументов, операционной системе возвращается значение 0.

Математические вычисления и числа

В дополнение к встроенным типам чисел `int`, `float` и `complex` библиотека предоставляет числовые типы `decimal.Decimal` и `fractions.Fraction`. Также в библиотеке имеется три математические библиотеки: `math`, содержащая стандартные математические функции; `cmath` – математические функции для работы с комплексными числами; `random`, содержащая множество функций генерации случайных чисел. Все эти модули были представлены в главе 2.

В модуле `numbers` имеются различные числовые абстрактные классы языка Python (классы, которые могут наследоваться, но которые не могут использоваться непосредственно). Их удобно использовать для проверки того, что объект, пусть это будет `x`, принадлежит к любому числовому типу с помощью вызова `isinstance(x, numbers.Number)` или к какому-нибудь определенному типу, например, `isinstance(x, numbers.Integral)`.

Специалисты, занимающиеся программированием научных или инженерных вычислений, найдут полезным пакет `NumPy`, разрабатываемый сторонними разработчиками. Этот пакет предоставляет высокоэффективную реализацию многомерных массивов, основных функций линейной алгебры и преобразований Фурье, а также инструменты интеграции с программным кодом на языках C, C++ и Fortran. Пакет `SciPy` включает `NumPy` и дополняет его модулями, предназначенными для выполнения статистических вычислений, обработки сигналов и изображений, модулями с генетическими алгоритмами и многими другими. Оба пакета доступны бесплатно на сайте *www.scipy.org*.

Время и дата

Модули `calendar` и `datetime` содержат функции и классы, предназначенные для работы с датами и временем. Однако они основаны на абстрактном Григорианском календаре, поэтому они не годятся для работы с датами в календарях, предшествовавших Григорианскому. Дата и время – это очень сложная тема. В разное время и в разных местах использовались разные календари. Продолжительность суток не равна точно 24 часам, продолжительность года не равна точно 365 дням, существует летнее и зимнее время, а также различные часовые пояса. Класс `datetime.datetime` (но не в классе `datetime.date`) предоставляет поддержку работы с часовыми поясами, хотя она не включается по умолчанию. Однако имеются модули сторонних производителей, которые с успехом восполняют этот недостаток, например, `dateutil` (*www.labix.org/python-dateutil*) и `mxDateTime` (*www.egenix.com/products/python/mxBase/mxDateTime*).

Модуль `time` используется для работы с отметками времени, которые являются простыми числовыми значениями, представляющими число секунд, прошедших от начала эпохи (1970-01-01T00:00:00 в UNIX). Этот модуль может использоваться для получения на машине отметок текущего времени UTC (Coordinated Universal Time – универсальное глобальное время) или локального времени, учитывающего переход на летнее время, а также для создания строк, представляющих дату, время и дату/время, отформатированных разными способами. Кроме того, он может использоваться для анализа строк, содержащих дату и время.

Пример: модули calendar, datetime и time

Объекты типа `datetime.datetime` обычно создаются программным способом, тогда как объекты, хранящие дату/время UTC, обычно получают информацию из внешних источников, таких как время создания файла. Ниже приводится несколько примеров:

```
import calendar, datetime, time
moon_datetime_a = datetime.datetime(1969, 7, 20, 20, 17, 40)
moon_time = calendar.timegm(moon_datetime_a.utctimetuple())
moon_datetime_b = datetime.datetime.utcfromtimestamp(moon_time)
moon_datetime_a.isoformat()     # вернет: '1969-07-20T20:17:40'
moon_datetime_b.isoformat()     # вернет: '1969-07-20T20:17:40'
time.strftime("%Y-%m-%dT%H:%M:%S", time.gmtime(moon_time))
```

Переменная `moon_datetime_a` является объектом типа `datetime.datetime` и хранит дату и время посадки корабля «Аполлон 11» на поверхность Луны. Переменная `moon_time` имеет тип `int` и хранит число секунд, прошедших от начала эпохи до момента посадки на Луну. Это число возвращает функция `calendar.timegm()`, которая принимает объект типа `time_struct`, возвращаемый функцией `datetime.datetime.utctimetuple()`, и возвращает число секунд, которое представляет тип `time_struct`.

(Поскольку посадка на Луну произошла до начала эпохи UNIX, число получится отрицательным.) Переменная `moon_datetime_b` является объектом типа `datetime.datetime`, и ее значение было получено из целочисленной переменной `moon_time`, чтобы продемонстрировать возможность преобразования числа секунд, прошедших с начала эпохи в объект типа `datetime.datetime`.[1] Последние три строки возвращают идентичные строки, содержащие дату/время в формате ISO 8601.

Текущие дату/время UTC можно получить в виде объекта `datetime.datetime`, вызвав функцию `datetime.datetime.utcnow()`, а в виде числа секунд, прошедших с начала эпохи, – вызвав функцию `time.time()`. Для получения локальных даты/времени можно использовать `datetime.datetime.now()` или `time.mktime(time.localtime ())`.

Алгоритмы и типы коллекций

Модуль `bisect` содержит функции поиска в отсортированных последовательностях, таких как отсортированные списки, а также функции вставки элементов с сохранением порядка сортировки. Функции этого модуля используют алгоритм поиска методом половинного деления, поэтому они отличаются очень высокой скоростью работы. Модуль `heapq` содержит функции для преобразования последовательности, такой как список, в «кучу» – разновидности коллекции, где первым элементом (в позиции с индексом 0) всегда является наименьший элемент, и функции для добавления и удаления элементов, при которых последовательность остается кучей.

Словари со значениями по умолчанию, стр. 161

Пакет `collections` содержит определения таких типов данных, как словарь `collections.defaultdict` и кортеж `collections.namedtuple`, которые уже рассматривались ранее. Кроме того, в этом модуле объявляются типы данных `collections.UserList` и `collections.UserDict`, хотя на практике чаще используются встроенные подклассы типов `list` и `dict`, чем эти типы данных. Еще один тип данных – `collections.deque` – похож на список, но если список обеспечивает очень быстрое добавление и удаление элементов в конце списка, то очереди `collections.deque` обеспечивают очень быстрое добавление и удаление элементов на обоих концах очереди – как в конце, так и в начале.

Именованные кортежи, стр. 134

В пакете `collections` также присутствуют определения нечисловых абстрактных классов Python (классы, которые могут наследоваться, но которые нельзя использовать непосредственно). Они будут обсуждаться в главе 8.

---

[1] К сожалению, в системе Windows функция `datetime.datetime.utcfromtimestamp()` не может обрабатывать отрицательные отметки времени, то есть отметки времени, предшествующие дате 1 января 1970 года.

Модуль `array` содержит определение типа последовательности `array.array`, способной хранить числа или символы весьма экономным способом. Этот тип данных напоминает списки, за исключением того, что объекты этого типа могут хранить только элементы определенного типа, который определяется на этапе его создания, поэтому, в отличие от списков, они не могут одновременно хранить объекты разных типов. Упоминавшийся ранее пакет `NumPy` также предоставляет эффективную реализацию массивов.

Модуль `weakref` содержит средства создания слабых ссылок, которые ведут себя подобно обычным ссылкам на объекты, за исключением того, что если единственная ссылка на объект – слабая ссылка, то такой объект может считаться готовым к утилизации. Это предотвращает сохранение объекта в памяти из-за присутствия ссылки на него. Естественно, имеется возможность проверить существование объекта, на который указывает слабая ссылка, и при его наличии мы можем с помощью этой ссылки обратиться к объекту.

Пример: модуль heapq

Модуль `heapq` содержит средства преобразования списка в кучу, а также для добавления элементов в кучу и удаления их из кучи, сохраняя порядок следования элементов в списке, *характерный для кучи*. Куча – это двоичное дерево, обладающее свойствами кучи, когда первый элемент (находящийся в позиции с индексом 0) является самым маленьким.[1] Каждое *поддерево* в куче также является кучей, поэтому любое поддерево тоже обладает всеми свойствами кучи. Ниже показано, как можно создать кучу с чистого листа:

```
import heapq
heap = []
heapq.heappush(heap, (5, "rest"))
heapq.heappush(heap, (2, "work"))
heapq.heappush(heap, (4, "study"))
```

Если список уже существует, его можно преобразовать в кучу с помощью функции `heapq.heapify(`*`alist`*`)`, которая выполнит необходимое переупорядочивание элементов списка. Наименьший элемент может быть удален из кучи с помощью функции `heapq.heappop(`*`heap`*`)`.

```
for x in heapq.merge([1, 3, 5, 8], [2, 4, 7], [0, 1, 6, 8, 9]):
    print(x, end=" ") # выведет: 0 1 1 2 3 4 5 6 7 8 8 9
```

Функция `heapq.merge()` принимает произвольное число отсортированных итерируемых объектов в виде аргументов и возвращает итератор, позволяющий выполнить итерации по всем элементам всех итерируемых объектов в порядке возрастания.

---

[1] Строго говоря, модуль `heapq` реализует тип кучи *min heap*. Кучи, где первый элемент всегда является наибольшим, относятся к типу *max heap*.

Форматы файлов, кодировки и сохранение данных

Кодировки символов, стр. 112

Стандартная библиотека имеет обширную поддержку стандартных форматов файлов и кодировок. Модуль `base64` содержит функции чтения и записи с использованием кодировок Base16, Base32 и Base64 в соответствии с RFC 3548.[1] Модуль `quopri` содержит функции чтения и записи в формате «quoted-printable».[2] Этот формат определяется документом RFC 1521 и используется для представления данных MIME (Multipurpose Internet Mail Extensions – многоцелевые расширения электронной почты Интернета). Модуль `uu` содержит функции чтения и записи данных в формате uuencode. Документ RFC 1832 определяет «External Data Representation Standard» (стандарт представления внешних данных), а модуль `xdrlib` содержит функции чтения и записи данных в этом формате.

Существуют также модули, предоставляющие возможность чтения и записи архивных файлов наиболее популярных форматов. Модуль `bz2` обеспечивает возможность работы с файлами *.bz2*, модуль `gzip` обеспечивает возможность работы с файлами *.gz*, модуль `tarfile` обеспечивает возможность работы с файлами *.tar*, *.tar.gz* (а также *.tgz*) и *.tar.bz2* и модуль `zipfile` обеспечивает возможность работы с файлами *.zip*. В этом подразделе мы увидим пример использования модуля `tarfile`, а немного ниже (на стр. 266) будет представлен небольшой пример, в котором используется модуль `gzip`. Еще раз с модулем `gzip` мы встретимся в главе 7.

Кроме того, стандартная библиотека обеспечивает поддержку некоторых форматов представления аудиоданных – например, модуль `aifc` реализует поддержку формата AIFF (Audio Interchange File Format – формат файлов для обмена аудиоданными) и модуль `wave` обеспечивает возможность для работы с файлами *.wav* (несжатыми). Некоторыми разновидностями аудиоданных можно манипулировать с помощью модуля `audioop`, а модуль `sndhdr` предоставляет пару функций, позволяющих определить тип аудиоданных, хранящихся в файле, и некоторые характеристики этих данных, такие как частота дискретизации.

Формат представления конфигурационных файлов (подобный формату файлов *.ini* в системе Windows) определяется документом RFC 822,

---

[1] RFC (Request for Comments – запрос на комментарии и предложения) – это документы, используемые для определения различных интернет-технологий. Каждый документ имеет уникальный идентификационный номер, и многие из них со временем становятся официальными стандартами.

[2] Способ 7-битной кодировки, когда символы, не входящие в набор ASCII, преобразуются в их шестнадцатеричные коды, записанные латиницей. – *Прим. перев.*

а модуль `configparser` предоставляет функции чтения и записи таких файлов.

Многие приложения, такие как Excel, могут читать и писать данные в формате CSV (Comma Separated Value – значения, разделенные запятыми) или в его разновидностях, таких как значения, разделенные символами табуляции. Модуль `csv` обеспечивает средства чтения и записи этих форматов и в состоянии учитывать некоторые особенности, препятствующие возможности непосредственной обработки файлов CSV.

В дополнение к поддержке различных форматов файлов стандартная библиотека содержит пакеты и модули, обеспечивающие средства сохранения данных. Модуль `pickle` используется для сохранения на диске и восстановления с диска произвольных объектов Python (включая целые коллекции) – подробнее об этом модуле рассказывается в главе 7. Помимо этого, стандартная библиотека поддерживает файлы DBM различных типов – эти файлы напоминают словари за исключением того, что их содержимое хранится на диске, а не в памяти, а их ключи и значения должны быть либо объектами типа bytes, либо строками. Модуль shelve, описываемый в главе 11, может использоваться для работы с файлами DBM со строковыми ключами и произвольными объектами Python в качестве значений – модуль незаметно для пользователя преобразует объекты Python в объекты типа bytes и обратно. Модули для работы с файлами DBM, прикладной программный интерфейс к базам данных и использование встроенной базы данных SQLite рассматриваются в главе 11.

Пример: модуль base64

Модуль `base64` главным образом используется для обработки двоичных данных, внедренных в сообщения электронной почты в виде текста ASCII. Он также может использоваться для сохранения двоичных данных в файлах с расширением *.py*. Первый шаг состоит в том, чтобы преобразовать двоичные данные в формат Base64. В следующем фрагменте предполагается, что модуль `base64` уже был импортирован, а путь к файлу *.png* хранится в переменной `left_align_png`:

```
binary = open(left_align_png, "rb").read()
ascii_text = ""
for i, c in enumerate(base64.b64encode(binary)):
    if i and i % 68 == 0:
        ascii_text += "\\\n"
    ascii_text += chr(c)
```

left_align.png

Этот фрагмент программного кода читает файл в режиме двоичного доступа и преобразует его в строку символов ASCII, в формате Base64. После каждого шестьдесят восьмого символа к строке добавляется комбинация символа обратного слеша и перевода строки. Это ограничивает

Тип данных bytes, стр. 344

ширину строк 68 символами ASCII и гарантирует, что при обратном чтении данных символы перевода строки будут проигнорированы (потому что символы обратного слеша экранируют их). Текст ASCII, полученный таким способом, может сохраняться в виде литерала типа `bytes` в файле с расширением *.py*, например:

```
LEFT_ALIGN_PNG = b"""\
iVBORw0KGgoAAAANSUhEUgAAACAAAAAgCAYAAABzenr0AAAABGdBTUEAALGPC/xhBQAA\
...
bmquu8PAmVT2+CwVV6rCyA9UfFMCkI+bN6p18tCWqcUzrDOwBh2zVCR+JZVeAAAAAElF\
TkSuQmCC"""
```

Мы опустили большую часть строк, заместив их многоточием.

Данные могут быть преобразованы обратно в первоначальный формат, как показано ниже:

```
binary = base64.b64decode(LEFT_ALIGN_PNG)
```

Двоичные данные могут быть записаны в файл с помощью цепочки вызовов: `open(`*`filename`*`, "wb").write(`*`binary`*`)`. Двоичные данные в файлах *.py* занимают значительно больше места, чем в оригинальной форме, но такая возможность может быть полезной, когда нам потребуется написать программу, хранящую все необходимые двоичные данные в виде единственного файла *.py*.

Пример: модуль tarfile

В большинстве версий Windows отсутствует встроенная поддержка работы с форматом *.tar*, который очень широко используется в системах UNIX. Этот недостаток легко можно ликвидировать с помощью модуля `tarfile` из стандартной библиотеки Python, который способен создавать и распаковывать архивы *.tar* и *.tar.gz* (которые называют *тарболлами*), а при наличии дополнительных библиотек еще и архивы *.tar.bz2*. Ниже приводятся ключевые выдержки из программы *untar.py*, способной распаковывать тарболлы средствами модуля `tarfile`. Начинается программа с инструкций импортирования:

```
BZ2_AVAILABLE = True
try:
    import bz2
except ImportError:
    BZ2_AVAILABLE = False
```

Модуль `bz2` используется для работы с форматом сжатия bzip2, но операция импортирования будет терпеть неудачу, если интерпретатор Python был собран без доступа к библиотеке bzip2. (Версии Python для Windows всегда собираются со встроенной поддержкой сжатия bzip2, поэтому отсутствовать она может только в некоторых сборках для UNIX.) Здесь учитывается возможность того, что модуль может быть недоступен, именно поэтому используется блок `try ... except` и логическая переменная, к которой можно будет обратиться позже (хотя

здесь мы не будем приводить программный код, который обращается к ней).

```
UNTRUSTED_PREFIXES = tuple(["/", "\\"] +
        [c + ":" for c in string.ascii_letters])
```

Эта инструкция создает кортеж (`'/'`, `'\'`, `'A:'`, `'B:'`, `...`, `'Z:'`, `'a:'`, `'b:'`, `...`, `'z:'`). Любое имя файла в тарболле, начинающееся с указанных префиксов, считается подозрительным — в именах файлов в тарболле не должны использоваться абсолютные пути, поскольку это влечет за собой риск перезаписи системных файлов; поэтому в качестве предварительной меры мы не будем распаковывать файлы, имена которых начинаются с указанных префиксов.

```
def untar(archive):
    tar = None
    try:
        tar = tarfile.open(archive)
        for member in tar.getmembers():
            if member.name.startswith(UNTRUSTED_PREFIXES):
                print("untrusted prefix, ignoring", member.name)
            elif ".." in member.name:
                print("suspect path, ignoring", member.name)
            else:
                tar.extract(member)
                print("unpacked", member.name)
    except (tarfile.TarError, EnvironmentError) as err:
        error(err)
    finally:
        if tar is not None:
            tar.close()
```

Каждый файл в тарболле называется *членом*. Функция `tarfile.getmembers()` возвращает список объектов `tarfile.TarInfo`, по одному для каждого члена. Имена файлов членов, включая пути, хранятся в атрибуте `tarfile.TarInfo.name`. Если имя начинается с одного из подозрительных префиксов или содержит `..` в пути, программа выводит сообщение об ошибке; в противном случае вызывается функция `tarfile.extract()`, сохраняющая член на диск. Модуль `tarfile` определяет множество собственных исключений, но в программе используется упрощенный подход к обработке ошибок, поэтому, когда возбуждается какое-либо исключение, она просто выводит текст сообщения об ошибке и завершает работу.

```
def error(message, exit_status=1):
    print(message)
    sys.exit(exit_status)
```

Функция `error()` приведена здесь лишь для полноты картины. Функция `main()` (которая здесь не приводится) выводит сообщение о порядке использования, если программа была запущена с ключом `-h` или

--help; в противном случае она выполняет некоторые основные проверки, после чего вызывает функцию untar(), передавая ей имя файла тарболла.

Работа с файлами, каталогами и процессами

Модуль shutil предоставляет высокоуровневые функции для работы с файлами и каталогами, включая shutil.copy() и shutil.copytree(), позволяющие копировать файлы и целые деревья каталогов; shutil.move(), позволяющую перемещать деревья каталогов, и shutil.rmtree(), позволяющую удалять целые деревья каталогов, даже непустые.

Временные файлы и каталоги должны создаваться с помощью модуля tempfile, который включает все необходимые для этого функции, например, tempfile.mkstemp(), и обеспечивает максимально возможную безопасность временных файлов.

Модуль filecmp может использоваться для сравнения файлов – с помощью функции filecmp.cmp() и целых каталогов – с помощью функции filecmp.cmpfiles().

Одна из областей, где особенно эффективно могут использоваться программы на языке Python, – это управление ходом выполнения других программ. Реализовать такое управление можно средствами модуля subprocess, позволяющими запускать другие процессы, взаимодействовать с ними с помощью каналов и получать возвращаемые значения. Этот модуль описывается в главе 9. Существует более мощная альтернатива, в виде модуля multiprocessing, обладающего обширными возможностями распределения работы между несколькими процессами и сбора результатов. Этот модуль нередко может использоваться как альтернатива многопоточной обработке данных.

Модуль os обеспечивает платформонезависимый доступ к средствам операционной системы. Переменная os.environ хранит объект отображения, элементами которого являются имена переменных окружения и их значения. Рабочий каталог программы можно получить с помощью функции os.getcwd(), а изменить его можно с помощью функции os.chdir(). Кроме того, модуль содержит функции для низкоуровневой работы с файлами на основе их дескрипторов. Функция os.access() может использоваться для определения наличия файла или его доступности для чтения или записи. Функция os.listdir() возвращает список записей (то есть имен файлов и каталогов, за исключением элементов . и ..) в указанном каталоге. Функция os.stat() возвращает различные сведения о файле или каталоге, такие как режим доступа, время последнего обращения и размер.

Каталоги могут создаваться с помощью функции os.mkdir() или, если потребуется попутно создать промежуточные каталоги, с помощью функции os.makedirs(). Пустые каталоги могут удаляться с помощью функции os.rmdir(), а деревья каталогов, содержащие только пустые

каталоги, – с помощью функции os.removedirs(). Файлы или каталоги могут удаляться с помощью функции os.remove(), а переименовываться с помощью функции os.rename().

Функция os.walk() позволяет выполнять итерации по всему дереву каталогов, по очереди извлекая все имена файлов и каталогов.

Кроме того, модуль os содержит множество низкоуровневых, платформозависимых функций, например, для работы с дескрипторами файлов, а также для ветвления (только в системах UNIX), порождения дочерних процессов и для запуска процессов.

Модуль os предоставляет функции для взаимодействия с операционной системой, в частности, для работы с файловой системой, а модуль os.path содержит набор функций для работы со строками (путями к файлам) и некоторые вспомогательные функции для работы с файловой системой. Функция os.path.abspath() возвращает абсолютный путь для своего аргумента, с удалением избыточных разделителей имен каталогов и элементов .... Функция os.path.split() возвращает кортеж, содержащий 2 элемента, первый элемент которого содержит путь, а второй – имя файла (который будет представлен пустой строкой, если имя файла в указанном пути не задано). Эти две части могут быть получены по отдельности, с помощью функций os.path.dirname() и os.path.basename() соответственно. Имя файла также может быть разбито на две части – имя и расширение, с помощью функции os.path.splittext(). Функция os.path.join() принимает произвольное число строк путей и возвращает единый путь, используя платформозависимый разделитель каталогов.

Если в программе потребуется получить комплекс сведений о файле или каталоге, можно использовать функцию os.stat(), но когда необходимы только отдельные элементы информации, можно использовать соответствующие функции из модуля os.path, например, os.path.exists(), os.path.getsize(), os.path.isfile() или os.path.isdir().

Модуль mimetypes включает функцию mimetypes.guess_type(), которая пытается определить тип MIME файла.

Пример: модули os и os.path

Ниже показано, как можно использовать модули os и os.path для создания словаря, каждый ключ которого представляет собой имя файла (включая его путь), а значение – отметку времени (количество секунд, прошедших от начала эпохи), когда произошло последнее изменение файла, для всех файлов в каталоге path:

```
date_from_name = {}
for name in os.listdir(path):
    fullname = os.path.join(path, name)
    if os.path.isfile(fullname):
        date_from_name[fullname] = os.path.getmtime(fullname)
```

Этот фрагмент программного кода выглядит очень понятным, но он может использоваться для составления перечня файлов только в одном каталоге. Если необходимо выполнить обход всего дерева каталогов, можно воспользоваться функцией `os.walk()`.

Ниже приводится фрагмент программы *finddup.py*.[1] Программный код создает словарь, каждый ключ которого представляет собой кортеж из двух элементов (размер файла и имя файла), где имя файла не содержит пути к нему. Каждое значение словаря – это список полных имен файлов, соответствующих имени файла в ключе и имеющих тот же размер:

```
data = collections.defaultdict(list)

for root, dirs, files in os.walk(path):
    for filename in files:
        fullname = os.path.join(root, filename)
        key = (os.path.getsize(fullname), filename)
        data[key].append(fullname)
```

Для каждого каталога функция `os.walk()` возвращает путь к корневому каталогу поддерева и два списка, один из них – это список подкаталогов в каталоге, а второй – список файлов в каталоге. Чтобы получить полный путь к файлу, необходимо объединить путь к корню и имя файла. Примечательно, что здесь не требуется выполнять рекурсию, так как функция `os.walk()` делает это сама. После сбора всей необходимой информации можно выполнить обход получившегося словаря и вывести отчет о возможных дубликатах файлов:

```
for size, filename in sorted(data):
    names = data[(size, filename)]
    if len(names) > 1:
        print("{0} ({1} bytes) may be duplicated "
              "({2} files):".format(filename, size, len(names)))
        for name in names:
            print("\t{0}".format(name))
```

Поскольку в качестве ключей словаря используются кортежи (размер и имя файла), нам не требуется использовать функцию `key`, чтобы отсортировать данные по размеру файла. Если какому-либо кортежу (размер, имя файла) соответствует более одного имени файла в списке, это могут быть дубликаты одного и того же файла.

```
...
shell32.dll (8460288 bytes) may be duplicated (2 files):
        \windows\system32\shell32.dll
        \windows\system32\dllcache\shell32.dll
```

[1] В главе 9 приводится более сложная версия программы поиска дубликатов файлов, *findduplicates-t.py*, которая использует многопоточный режим работы и применяет вычисление контрольных сумм MD5.

Это последний элемент из вывода, содержащего 3 282 строки, полученного командой `finddup.py \windows` в системе Windows XP.

Работа с сетями и Интернетом

Пакеты и модули для работы с сетями и Интернетом составляют основную часть стандартной библиотеки Python. На самом низком уровне модуль `socket` предоставляет наиболее фундаментальные функциональные возможности для работы с сетями, среди которых имеются функции создания сокетов, выполнения запросов к DNS (Domain Name System – система доменных имен) и обработки IP-адресов (internet Protocol – протокол Интернета). Настроить шифрование и аутентификацию при работе с сокетами можно с помощью модуля `ssl`. Модуль `socketserver` предоставляет реализации серверов TCP (Transmission Control Protocol – протокол управления передачей) и UDP (User Datagram Protocol – протокол пользовательских дейтаграмм). Эти серверы могут обрабатывать запросы непосредственно или создавать отдельные процессы (за счет ветвления) и потоки управления для обработки каждого запроса. Асинхронная обработка сокетов на стороне клиентов и серверов может быть реализована с помощью модуля `asyncore` и построенного на его основе более высокоуровневого модуля `asynchat`.

В стандартной библиотеке Python имеется реализация WSGI (Web Server Gateway Interface – интерфейс шлюза веб-сервера), представляющая собой стандартный интерфейс между веб-серверами и веб-приложениями, написанными на языке Python. В поддержку стандарта пакет `wsgiref` предоставляет рекомендации по внедрению WSGI и содержит модули для реализации серверов HTTP, совместимых с требованиями спецификаций WSGI, способных обрабатывать заголовки ответов и сценарии CGI (Common Gateway Interface – общий шлюзовой интерфейс). Кроме того, модуль `http.server` предоставляет реализацию сервера HTTP, которому можно определить обработчик запросов (стандартная реализация предоставляется) для запуска сценариев CGI. Модули `http.cookies` и `http.cookiejar` содержат функции для работы с cookies, а поддержка сценариев CGI предоставляется модулями `cgi` и `cgitb`.

Доступ к запросам HTTP на стороне клиента может быть реализован с помощью модуля `http.client`, хотя более простой и удобный доступ к адресам URL обеспечивается модулями из пакета `urllib`: `urllib.parse`, `urllib.request`, `urllib.response`, `urllib.error` и `urllib.robotparser`. Загрузка файлов из Интернета выполняется очень просто, как показано ниже:

```
fh = urllib.request.urlopen("http://www.python.org/index.html")
html = fh.read().decode("utf8")
```

Функция `urllib.request.urlopen()` возвращает объект, который ведет себя практически как объект файла, открытый для чтения в двоичном

режиме. Этот фрагмент получает файл *index.html* с веб-сайта Python (в виде объекта `bytes`) и запоминает его в виде строки в переменной `html`. Имеется также возможность загружать файлы и сохранять их в локальной файловой системе с помощью функции `urllib.request.urlretrieve()`.

Имеется возможность производить синтаксический анализ документов HTML и XHTML с помощью модуля `html.parser`; адреса URL могут анализироваться и создаваться с помощью модуля `urllib.parse`; а файлы *robots.txt* могут анализироваться с помощью модуля `urllib.robotparser`. Данные в формате JSON (JavaScript Object Notation – формат записи объектов JavaScript) могут читаться и записываться с помощью модуля `json`.

Помимо поддержки серверов и клиентов HTTP в библиотеке имеется поддержка XML-RPC (Remote Procedure Call – вызов удаленных процедур), реализованная в виде модулей `xmlrpc.server` и `xmlrpc.client`. Дополнительные возможности для работы на стороне клиента с протоколом FTP (File Transpotr Protocol – протокол передачи файлов) реализованы в виде модуля `ftplib`; для работы с протоколом NNTP (Network News Transport Protocol – сетевой протокол передачи новостей) – в виде модуля `nntplib`; для работы с протоколом TELNET – в виде модуля `telnetlib`.

Модуль `smtpd` предоставляет реализацию сервера SMTP (Simple Mail Transport Protocol – упрощенный протокол электронной почты), модуль `smtplib` предоставляет возможность реализации клиентов электронной почты для протокола SMTP, модуль `imaplib` – для протокола IMAP4 (internet Message Access Protocol – протокол интерактивного доступа к электронной почте) и модуль `poplib` – для протокола POP3 (Post Office Protocol – протокол электронной почты). Возможность доступа к почтовым ящикам различных форматов обеспечивает модуль `mailbox`. Отдельные сообщения электронной почты (включая сообщения, состоящие из нескольких частей) могут создаваться и обрабатываться средствами модуля `email`.

Если возможностей пакетов и модулей стандартной библиотеки окажется недостаточно, можно обратиться к Twisted (*www.twistedmatrix.com*) – обширной библиотеке средств для работы с сетями, разрабатываемой сторонними разработчиками. Кроме того, существует множество сторонних библиотек, предназначенных для разработки веб-приложений, включая Django (*www.djangoproject.com*) и Turbogears (*www.turbogears.org*), а также Plone (*www.plone.org*) и Zope (*www.zope.org*), представляющих собой целые платформы для разработки систем управления содержимым. Все эти библиотеки написаны на языке Python.

XML

Для парсинга документов XML широко используются два основных подхода. Один из них основан на анализе DOM (Document Object Model – объектная модель документа), а другой – на использовании SAX (Simple API for XML – упрощенный прикладной интерфейс для работы с документами XML). В библиотеке имеется два парсера DOM – один из них представлен модулем `xml.dom`, а второй – модулем `xml.dom.minidom`. Парсер SAX представлен модулем `xml.sax`. Мы уже использовали функцию `xml.sax.saxutils.escape()` из модуля `xml.sax.saxutils` (для экранирования служебных символов «&», «<» и «>»). Существует также функция `xml.sax.saxutils.quoteattr()`, которая выполняет то же действие, но дополнительно экранирует кавычки (чтобы текст можно было использовать в атрибутах тегов); обратное преобразование можно выполнить с помощью функции `xml.sax.saxutils.unescape()`.

В библиотеке существует еще два парсера. Модуль `xml.parsers.expat` может использоваться для работы с документами XML с применением библиотеки expat, при наличии этой библиотеки в системе, и модуль `xml.etree.ElementTree` может использоваться для работы с документами XML через интерфейс словарей и списков. (По умолчанию парсеры DOM и дерева элементов за кулисами сами используют парсер, использующий библиотеку `expat`.)

Порядок создания документов XML вручную, с применением модулей DOM и деревьев элементов, и парсинг с использованием парсеров DOM, SAX и дерева элементов, описывается в главе 7.

Пример: модуль xml.etree.ElementTree

Парсеры DOM и SAX предоставляют прикладной программный интерфейс, которым пользуются опытные программисты, хорошо знающие формат XML, а модуль `xml.etree.ElementTree` предлагает более простой и более естественный для языка Python подход к парсингу и созданию документов XML. Модуль дерева элементов совсем недавно был добавлен в стандартную библиотеку[1] и потому может оказаться незнакомым для некоторых читателей. Поэтому мы представим здесь очень короткий пример, чтобы можно было составить общее представление об этом модуле. В главе 7 будет представлен более сложный пример и проведено сравнение программного кода, использующего парсеры DOM и SAX.

Веб-сайт организации NOAA (National Oceanic and Atmospheric Administration – Национальное управление по исследованию океанов и атмосферы) правительства США предоставляет самые разнообразные данные, включая файл в формате XML, в котором перечислены метео-

[1] Модуль `xml.etree.ElementTree` был включен в стандартную библиотеку в версии Python 2.5.

рологические станции США. Файл насчитывает свыше 20 000 строк и содержит сведения примерно о двух тысячах метеорологических станций. Ниже приводится типичный пример одной из записей:

```
<station>
    <station_id>KBOS</station_id>
    <state>MA</state>
    <station_name>Boston, Logan International Airport</station_name>
    ...
    <xml_url>http://weather.gov/data/current_obs/KBOS.xml</xml_url>
</station>
```

Мы исключили несколько строк и уменьшили отступы. Размер файла составляет примерно 840 Кбайт, поэтому мы сжали его с помощью gzip до более приемлемого размера в 72 Кбайт. К сожалению, парсер на основе анализа элементов дерева требует либо имя файла, либо объект файла, но он не в состоянии работать со сжатыми файлами, так как с его точки зрения такие файлы являются набором случайных двоичных данных. Решить эту проблему можно, выполнив следующие два действия:

```
binary = gzip.open(filename).read()
fh = io.StringIO(binary.decode("utf8"))
```

Тип данных bytes, стр. 344

Функция `gzip.open()` из модуля `gzip` напоминает встроенную функцию `open()`, за исключением того, что она читает файлы, сжатые при помощи утилиты gzip (то есть с файлы с расширением *.gz*), просто как двоичные данные. Нам необходимо обеспечить доступность этих данных для парсера в виде файла, поэтому мы использовали метод `bytes.decode()` для преобразования двоичных данных в строку с кодировкой символов UTF-8 (эта кодировка по умолчанию используется для файлов XML) и создали объект `io.StringIO`, напоминающий файл, со строкой, вмещающей все содержимое файла XML.

Тип данных io.StringIO, стр. 249

```
tree = xml.etree.ElementTree.ElementTree()
root = tree.parse(fh)
stations = []
for element in tree.getiterator("station_name"):
    stations.append(element.text)
```

Здесь мы создали новый объект `xml.etree.ElementTree.ElementTree` и передали ему объект файла, откуда он будет читать содержимое файла XML, который нам требуется проанализировать. Парсер требует, чтобы ему был передан объект файла, открытого для чтения, хотя в действительности он читает его содержимое в строку объекта `io.StringIO`. Нам требуется извлечь из файла названия метеорологических станций, и это легко сделать с помощью метода `xml.etree.ElementTree.ElementTree.getiterator()`, который возвращает итератор, выполняющий итерации по всем объектам `xml.etree.ElementTree.Element`, имеющим

тег с указанным именем. Чтобы извлечь текст, достаточно воспользоваться атрибутом `text` элемента. Как и в случае с функцией `os.walk()`, нам не требуется предусматривать рекурсивную обработку – метод-итератор сам сделает все необходимое. Если метод вызвать без имени тега, то с помощью полученного итератора можно будет выполнить обход всех элементов документа XML.

Прочие модули

В книге недостаточно места, чтобы охватить почти 200 пакетов и модулей, входящих в состав стандартной библиотеки. Тем не менее этого краткого обзора вполне достаточно, чтобы получить представление о некоторых ключевых пакетах, применяемых в наиболее важных областях программирования. В последнем подразделе этого раздела мы рассмотрим еще несколько областей, представляющих для нас интерес.

В предыдущем разделе мы видели, насколько просто создавать тесты в строках документирования и запускать их с помощью модуля `doctest`. В составе библиотеки имеется также платформа модульного тестирования, реализованная в виде модуля `unittest`, – это версия платформы тестирования JUnit языка Java, реализованная для языка Python. Кроме того, модуль `doctest` предоставляет некоторые возможности интеграции с модулем `unittest`. Помимо этого, существуют платформы тестирования, созданные сторонними разработчиками, например `py.test` (*codespeak.net/py/dist/*) и `nose` (*www.somethingaboutorange.com/mrl/projects/nose/*).

Приложения, работающие в неинтерактивном режиме, такие как серверы, часто сообщают о проблемах посредством записи сообщений в файлы журналов. Модуль `logging` предоставляет универсальный интерфейс для записи сообщений в файлы журналов, а также он способен отправлять сообщения с помощью запросов HTTP GET и POST, посредством сокетов или по электронной почте.

В библиотеке имеется множество модулей, позволяющих выполнять интроспекцию и манипулирование программным кодом, и хотя их обсуждение выходит далеко за рамки этой книги, тем не менее следует упомянуть о модуле `pprint`, который содержит функции форматированного вывода объектов Python, включая коллекции, что иногда бывает удобно при отладке. В главе 8 будет представлен простой пример использования модуля `inspect`, выполняющий интроспекцию существующих объектов.

Модуль `threading` предоставляет поддержку создания многопоточных приложений, а модуль `queue` реализует три различных типа очередей, которые могут безопасно использоваться в многопоточных приложениях. Тема управления несколькими потоками выполнения будет рассматриваться в главе 8.

В языке Python отсутствует встроенная поддержка создания приложений с графическим интерфейсом, тем не менее имеется несколько библиотек графического интерфейса, которые могут использоваться в программах на языке Python. Модуль `tkinter` обеспечивает доступ к библиотеке Tk, которая обычно устанавливается вместе с системой. Программирование графического интерфейса рассматривается в главе 13.

Поверхностное и глубокое копирование, стр. 173

Модуль `abc` (Abstract Base Class – базовый абстрактный класс) предоставляет функции, необходимые для создания базовых абстрактных классов. Этот модуль будет рассматриваться в главе 8.

Модуль `copy` предоставляет функции `copy.copy()` и `copy.deepcopy()`, которые уже обсуждались в главе 3.

Доступ к *внешним функциям*, то есть к функциям в разделяемых библиотеках (файлы *.dll* в Windows, *.dylib* – в Mac OS X и файлы *.so* – в Linux), обеспечивается модулем `ctypes`. В языке Python имеется также поддержка C API, благодаря чему имеется возможность создавать нестандартные типы данных и функции на языке C и обеспечивать их доступность из программного кода на языке Python. Обсуждение модуля `ctypes` и поддержки C API выходит далеко за рамки этой книги.

Если ни один из пакетов и модулей, упомянутых в этом разделе, не обеспечивает необходимые функциональные возможности, то прежде чем приступать к разработке собственных функций, ознакомьтесь с описанием глобального каталога модулей Python (Global Module Index); возможно, там вы найдете подходящий модуль, поскольку здесь мы не в состоянии упомянуть все существующие модули. В случае неудачи попробуйте поискать нужный модуль в каталоге пакетов Python (Python Package Index – *pypi.python.org/pypi*), в котором содержится несколько тысяч расширений для Python – от маленьких модулей, состоящих из единственного файла, и до огромных пакетов библиотек и платформ, насчитывающих сотни модулей.

В заключение

Эта глава была начата с рассмотрения нескольких разновидностей синтаксиса, используемых для импортирования пакетов, модулей и объектов, находящихся внутри модулей. Мы отметили, что многие программисты предпочитают использовать синтаксис `import` *`importable`*, чтобы избежать конфликтов имен, и что не следует давать программам и модулям имена, совпадающие с модулями или каталогами Python верхнего уровня.

Мы также обсудили пакеты Python. Пакеты – это обычные каталоги, содержащие файл *__init__.py* и один или более модулей *.py*. Файл *__init__.py* может быть пустым, но для поддержки синтаксиса `from` *`importable`* `import *` мы можем создать в этом файле специальную пере-

менную `__all__`, представляющую собой список имен в модуле. Кроме того, в файл *__init__.py* можно поместить любой программный код, выполняющий инициализацию. Также было отмечено, что пакеты могут вкладываться друг в друга, для чего достаточно просто создать подкаталоги, каждый из которых содержит свой собственный файл *__init__.py*.

Были описаны два нестандартных модуля. Первый из них предоставляет всего несколько функций и имеет очень простые доктесты. Второй модуль более сложный, имеет свои собственные исключения, использует возможность динамического создания функций с платформозависимой реализацией, частные глобальные данные, более сложные доктесты и выполняет функцию инициализации.

Примерно половина главы была посвящена обзору стандартной библиотеки языка Python. Было упомянуто несколько модулей, предназначенных для работы со строками, и представлена пара примеров использования объектов `io.StringIO`. Один из примеров продемонстрировал, как можно записать текст в файл либо с использованием встроенной функции `print()`, либо с использованием метода объекта файла `write()`, и как можно использовать объект `io.StringIO` вместо настоящего файла. В предыдущих главах мы обрабатывали аргументы командной строки, непосредственно читая содержимое `sys.argv`, но при обзоре поддержки обработки аргументов командной строки, включенной в библиотеку, мы познакомились с модулем `optparse`, который существенно упрощает работу с аргументами командной строки, – далее мы широко будем использовать этот модуль.

Была упомянута имеющаяся в языке превосходная поддержка работы с числами, числовые типы в библиотеке, три модуля с математическими функциями, а также поддержка научных и инженерных вычислений, предоставляемая проектом `SciPy`. Коротко были описаны библиотечные и созданные сторонними разработчиками классы для работы с датой/временем, а также представлены примеры, демонстрирующие, как можно получить текущие дату и время и как выполнять преобразования между типом `datetime.datetime` и количеством секунд, прошедших от начала эпохи. Также были рассмотрены дополнительные типы коллекций и алгоритмы работы с упорядоченными последовательностями, реализованные в стандартной библиотеке, наряду с несколькими примерами использования функций из модуля `heapq`.

Были представлены модули поддержки различных способов кодирования файлов (не имеющих отношения к кодировкам символов), модули для работы со сжатыми файлами в наиболее популярных форматах архивирования, а также модули поддержки работы с аудиоданными. Был дан пример, демонстрирующий порядок использования кодировки Base64 для сохранения двоичных данных в файлах *.py*, а также программа, выполняющая распаковывание тарболлов. Библиотекой предоставляется существенная поддержка операций над файлами

и каталогами, причем все эти операции реализованы в виде платформонезависимых функций. В приведенных примерах было показано, как можно создать словарь с именами файлов в виде ключей и временем последнего изменения в виде значений, а также продемонстрировано, как выполнить рекурсивный обход дерева каталогов с целью выявления дубликатов файлов, основываясь на их именах и размерах.

Огромную долю библиотеки занимают модули для реализации сетевых взаимодействий. Мы очень коротко рассмотрели, что имеется в библиотеке, начиная от обычных сокетов (включая сокеты с шифрованием трафика) до серверов TCP, UDP и HTTP и поддержки WSGI. Также были упомянуты модули, предназначенные для работы с cookies, сценариями CGI и данными протокола HTTP, средства синтаксического анализа HTML, XHTML и адресов URL. Были упомянуты прочие модули, включая модули для работы с протоколом XML-RPC и высокоуровневыми протоколами, такими как TP и NNTP, а также поддержка работы с протоколом электронной почты SMTP, как на стороне клиента, так и на стороне сервера, и поддержка протоколов IMAP4 и POЗ на стороне клиента.

Помимо всего прочего была упомянута имеющаяся в составе библиотеки мощная поддержка возможности записи и парсинга формата XML, включая парсеры DOM, SAX и дерева элементов, а также модуль `expat`. Был приведен пример использования модуля `xml.etree.ElementTree`. Также были упомянуты некоторые другие пакеты и модули, имеющиеся в библиотеке.

Стандартная библиотека языка Python представляет собой чрезвычайно ценный ресурс, который позволит сэкономить массу сил и времени, и во многих случаях позволяет писать более короткие программы, опирающиеся на функциональные возможности, предоставляемые библиотекой. Кроме того, существуют еще буквально тысячи пакетов сторонних разработчиков, восполняющих любую нехватку возможностей, которую можно обнаружить в стандартной библиотеке. Все эти предопределенные функциональные возможности позволяют нам сосредоточиться на предметной стороне решаемой задачи, оставляя большую часть деталей реализации за библиотечными модулями.

Этой главой заканчивается обсуждение фундаментальных принципов процедурного программирования. В последующих главах, и в частности в главе 8, мы познакомимся с более передовыми и более специализированными приемами процедурного программирования, а в следующей главе будут представлены приемы объектно-ориентированного программирования. Использование языка Python в качестве исключительно процедурного языка программирования вполне возможно и даже оправданно, особенно при создании небольших программ, но при разработке средних и крупных программ, собственных пакетов и модулей, а также для создания долгоживущих проектов, как правило, предпочтительнее использовать объектно-ориентированный подход.

К счастью, все, о чем рассказывалось до сих пор, с успехом может применяться и в объектно-ориентированном программировании, поэтому в следующих главах мы продолжим накапливать наши знания и навыки, основываясь на уже заложенном фундаменте.

Упражнение

Напишите программу, демонстрирующую содержимое каталогов подобно тому, как это делает команда `dir` в Windows или `ls` в UNIX. Преимущество наличия собственной программы отображения каталогов состоит в том, что мы можем заложить в нее предпочитаемые параметры по умолчанию и использовать одну и ту же программу в любой системе, не утруждая себя необходимостью запоминать различия между командами `dir` и `ls`. Программа должна иметь следующий интерфейс:

```
Usage: ls.py [options] [path1 [path2 [... pathN]]]

The paths are optional; if not given . is used.

Options:
  -h, --help      show this help message and exit
  -H, --hidden    show hidden files [default: off]
  -m, --modified  show last modified date/time [default: off]
  -o ORDER, --order=ORDER
                  order by ('name', 'n', 'modified', 'm', 'size', 's')
                  [default: name]
  -r, --recursive recurse into subdirectories [default: off]
  -s, --sizes     show sizes [default: off]
```

(Вывод программы был несколько изменен, чтобы уместить его в ширину книжной страницы.)

Ниже приводится пример вывода содержимого небольшого каталога с помощью команды `ls.py -ms -os misc/`:

```
2007-04-10 15:49:01          322 misc/chars.pyw
2007-08-01 11:24:57        1,039 misc/pfa-bug.pyw
2007-10-12 09:00:27        2,445 misc/test.lout
2007-04-10 15:50:31        2,848 misc/chars.png
2008-02-11 14:17:03       12,184 misc/abstract.pdf
2008-02-05 14:22:38      109,788 misc/klmqtintro.lyx
2007-12-13 12:01:14    1,359,950 misc/tracking.pdf
                                 misc/phonelog/
7 files, 1 directory
```

Мы использовали группировку ключей командной строки (она обрабатывается модулем `optparse` автоматически), но тот же самый эффект можно было бы получить, используя ключи по отдельности, например, `ls.py -m -s -os misc/`, или даже применив более плотную группировку, `ls.py -msos misc/`, или используя длинные имена параметров, `ls.py --modified --sizes --order=size misc/`, или любую их комбинацию.

Обратите внимание на наличие ключа, управляющего включением в вывод программы «скрытых» файлов или каталогов, имена которых начинаются с точки (`.`).

Упражнение довольно сложное. Вам придется ознакомиться с документацией к модулю `optparse`, чтобы узнать, как объявлять параметры, которые принимают значение `True`, и как определить фиксированный перечень параметров. Если пользователь определяет в вызове параметр `--recursive`, программа должна выполнить обход файлов (но не каталогов) с помощью функции `os.walk()`; в противном случае она должна использовать для получения списка файлов и каталогов функцию `os.listdir()`.

Еще один подводный камень – организация пропуска скрытых каталогов при рекурсии. Их можно удалять из списка `dirs`, возвращаемого `os.walk()`, и тем самым пропускать их, модифицируя список. Но будьте внимательны – не присваивайте новое значение непосредственно переменной `dirs`, поскольку это не повлияет на список, на который она ссылается, а просто (и совершенно бесполезно) заместит его. Подход, использованный в решении, основан на присваивании срезу всего списка, то есть `dirs[:] = [dir for dir in dirs if not dir.startswith(".")]`.

Функция `locale.setlocale()`, стр. 108

Лучший способ группировки разрядов при отображении размеров файлов состоит в том, чтобы импортировать модуль `locale`, вызвать функцию `locale.setlocale()` для получения региональных настроек пользователя и использовать спецификатор формата `n`. Общий размер программы *ls.py*, разбитой на четыре функции, будет составлять около 130 строк.

- Объектно-ориентированный подход
- Собственные классы
- Собственные классы коллекций

6

Объектно-ориентированное программирование

Во всех предыдущих главах мы широко использовали объекты, но при этом наш стиль программирования был исключительно процедурным. Язык Python одновременно поддерживает различные стили программирования – он позволяет программировать в процедурном стиле, объектно-ориентированном стиле и функциональном стиле, а также допускает смешивание стилей в любых пропорциях, не вынуждая нас писать программы, придерживаясь какого-то определенного стиля.

Вполне возможно написать любую программу исключительно в процедурном стиле, и для небольших программ (скажем, до 500 строк) это совершенно нормальный выбор. Но большинству программ, особенно средних и крупных, объектно-ориентированный стиль дает значительные преимущества.

В этой главе рассматриваются все фундаментальные концепции и приемы объектно-ориентированного программирования на языке Python. Первый раздел предназначен для тех, кто не обладает еще достаточным опытом, и для тех, у кого имеется опыт процедурного программирования (на таких языках, как C или Fortran). Второй раздел начинается со знакомства с некоторыми проблемами, свойственными процедурному программированию, которые могут быть решены с использованием объектно-ориентированного стиля. Затем коротко описываются особенности объектно-ориентированного программирования на языке Python и поясняется соответствующая терминология. Далее следуют два основных раздела главы.

Второй раздел охватывает тему создания собственных типов данных, способных хранить единственный элемент (хотя сами элементы могут иметь множество атрибутов), а в третьем разделе рассказывается о создании собственных типов коллекций, которые способны хранить про-

извольное число объектов любых типов. В этих разделах рассматривается большинство аспектов объектно-ориентированного программирования на языке Python, хотя обсуждение некоторых, более сложных тем отложено до главы 8.

Объектно-ориентированный подход

В этом разделе мы коснемся некоторых проблем, характерных для процедурного стиля программирования, на примере ситуации, когда в программе необходимо реализовать представление большого числа окружностей. Минимальные данные, необходимые для задания окружности, – это координаты ее центра (x, y) и радиус. Самое простое решение заключается в том, чтобы использовать для представления каждой окружности кортеж из трех элементов. Например:

```
circle = (11, 60, 8)
```

Один из недостатков такого подхода – в неочевидности назначения каждого элемента кортежа. Мы можем подразумевать `(x, y, radius)` или `(radius, x, y)`. Другой недостаток состоит в том, что обращаться к элементам кортежа мы можем только по их индексам. Если представить, что у нас имеется две функции, `distance_from_origin(`*`x, y`*`)` и `edge_distance_from_origin(`*`x, y, radius`*`)`, при обращении к ним нам потребуется использовать операцию распаковывания кортежа, представляющего окружность:

```
distance = distance_from_origin(*circle[:2])
distance = edge_distance_from_origin(*circle)
```

В обоих случаях предполагается, что кортеж имеет вид `(x, y, radius)`. Эту проблему можно решить, зная порядок следования элементов и используя операцию распаковывания с применением именованного кортежа:

```
import collections
Circle = collections.namedtuple("Circle", "x y radius")
circle = Circle(13, 84, 9)
distance = distance_from_origin(circle.x, circle.y)
```

Такой подход позволяет создавать трехэлементные кортежи типа `Circle` с именованными атрибутами, что делает вызов функций более простым и понятным, поскольку для обращения к элементам используются их имена. К сожалению, сама проблема при этом не исчезает. Например, ничто не мешает созданию окружности с ошибочными значениями:

```
circle = Circle(33, 56, -5)
```

Окружность с отрицательным радиусом – это сущая бессмыслица, но в данном случае именованный кортеж `circle` будет создан без возбуждения исключения, как если бы радиус был представлен обычной пе-

ременной, в которой оказалось отрицательное значение. Ошибка будет обнаружена только при вызове функции `edge_distance_from_origin()` и только если эта функция проверяет значение радиуса на отрицательное значение. Такая невозможность выполнить проверку данных в момент создания объекта является, пожалуй, самым негативным аспектом исключительно процедурного подхода.

Если нам потребуется изменять окружности, например перемещать их, изменяя координаты центра, или изменять их размеры, изменяя радиус, мы сможем воспользоваться методом `collections.namedtuple_replace()`:

```
circle = circle._replace(radius=12)
```

Но, так же, как и при создании кортежа типа `Circle`, ничто не предохранит (и даже не предупредит) от установки ошибочных значений.

Если в программе предусматривается возможность частого изменения параметров окружностей, мы могли бы ради удобства использовать такой изменчивый тип данных как список:

```
circle = [36, 77, 8]
```

Но и это решение не обеспечивает никакой защиты от ошибочных данных, а лучшее, что можно сделать для доступа к элементам по именам, это создать несколько констант, чтобы иметь возможность записывать более осмысленные инструкции, такие как `circle[RADIUS] = 5`. Но использование списков несет дополнительные проблемы, например, мы, на вполне законных основаниях, сможем вызвать метод `circle.sort()`! Как вариант, можно было бы использовать словарь, например, `circle = dict(x=36, y=77, radius=8)`, но и в этом случае нет никакой возможности проконтролировать значение радиуса и нет никакой защиты от вызова методов, неприменимых к окружностям.

Объектно-ориентированные концепции и терминология

Все, что нам необходимо, – это способ упаковать данные, представляющие окружность, и некоторый способ ограничить круг методов, которые могут применяться к данным, чтобы возможны были только допустимые операции. Обе поставленные задачи могут быть решены за счет создания собственного типа данных `Circle`. Далее в этом разделе мы увидим, как создать тип данных `Circle`, но сначала нам необходимо познакомиться с некоторыми начальными сведениями и терминологией. Не стоит беспокоиться, если сначала термины покажутся вам незнакомыми, они станут более понятными, когда мы перейдем к изучению примеров.

Мы будем использовать термины *класс*, *тип* и *тип данных* как взаимозаменяемые. В языке Python мы можем создавать собственные классы, полностью интегрированные в язык, которые могут использо-

ваться как любые встроенные типы данных. Мы уже сталкивались со многими классами, например, `dict`, `int` и `str`. Мы будем использовать термин *объект* и иногда *экземпляр*, для обозначения экземпляра определенного класса. Например, значение `5` – это объект класса `int`, а `"oblong"` – это объект класса `str`.

Большинство классов инкапсулируют не только данные, но и методы, применяемые к этим данным. Например, класс `str` хранит строки символов Юникода в виде данных и поддерживает методы, такие как `str.upper()`. Многие классы также поддерживают дополнительные особенности, например, мы можем объединить две строки (или любые две последовательности), используя оператор `+`, и определить длину последовательности с помощью встроенной функции `len()`. Такие особенности реализуются при помощи *специальных методов*, которые представляют собой самые обычные методы, за исключением того, что их имена всегда начинаются и заканчиваются двумя символами подчеркивания и являются предопределенными. Например, если нам потребуется создать класс, поддерживающий операцию конкатенации с использованием оператора + и функцию `len()`, мы сможем обеспечить такую поддержку, реализовав в нашем классе специальные методы `__add__()` и `__len__()`. Напротив, мы никогда не должны использовать имена, начинающиеся и заканчивающиеся двумя символами подчеркивания, если они не являются предопределенными именами специальных методов и не соответствуют назначению класса. Тем самым гарантируется, что мы никогда не вступим в конфликты с последующими версиями Python, даже если появятся новые предопределенные специальные методы.

Как правило, объекты имеют атрибуты, методы – это вызываемые атрибуты, а другие атрибуты – это данные. Например, объект `complex` имеет атрибуты `imag` и `real` и множество методов, включая специальные методы, такие как `__add__()` и `__sub__()` (поддержка двухместных операторов `+` и `-`), и обычные методы, такие как `conjugate()`. Атрибуты данных (часто их называют просто «атрибутами») обычно реализуются как *переменные экземпляра*, то есть переменные, уникальные для каждого конкретного объекта. Мы еще увидим примеры обычных атрибутов, а также примеры реализации атрибутов данных в виде *свойств*. Свойство – это элемент данных объекта, доступ к которым оформляется как доступ к переменной экземпляра, но само обращение неявно обслуживается методами доступа. Как будет показано ниже, использование свойств упрощает проверку корректности данных.

Внутри метода (который является обычной функцией, получающей в виде первого аргумента конкретный экземпляр класса, в контексте которого выполняются действия) потенциально доступны несколько разновидностей переменных. К переменным экземпляра можно обращаться посредством квалификации их имен самим экземпляром. Внутри методов могут создаваться локальные переменные – доступ

к ним осуществляется без квалификации имени. Доступ к переменным класса (иногда они называются статическими переменными) может осуществляться посредством квалификации их имен именем класса. Доступ к глобальным переменным, то есть к переменным модуля, осуществляется без квалификации их имен.

В некоторых книгах, посвященных языку Python, используется понятие *пространства имен* – отображения имен на объекты. Модули – это пространства имен. Например, выполнив инструкцию `import math`, мы получаем возможность обращаться к объектам в модуле `math`, квалифицируя их именем пространства имен (например, `math.pi` или `math.sin()`). Точно так же классы и объекты являются пространствами имен. Например, если представить, что была выполнена инструкция `z = complex(1, 2)`, то пространство имен объекта `z` будет содержать два доступных нам атрибута (`z.real` и `z.imag`).

Одно из преимуществ объектно-ориентированного подхода состоит в том, что если у нас имеется класс, мы можем *специализировать* его. Это означает, что можно создать новый класс, наследующий все атрибуты (данные и методы) из оригинального класса, и добавить в него или заместить некоторые методы, или добавить дополнительные переменные экземпляра. Мы можем создать *подкласс* (другое название *специализации*) любого класса Python, будь то встроенный класс, класс из стандартной библиотеки[1] или один из наших собственных классов. Возможность специализации – одно из важнейших преимуществ объектно-ориентированного программирования, поскольку она упрощает использование существующих классов, с опробованными и проверенными функциональными возможностями, в качестве основы для новых классов, расширяющих оригинал, добавляя новые атрибуты данных или новые функциональные возможности простым и понятным способом. Более того, имеется возможность передавать объекты новых классов функциям и методам, которые были написаны для работы с оригинальным классом, и при этом они будут работать вполне корректно.

Мы будем использовать термин *базовый класс* для обозначения наследуемого класса. Базовым классом может быть как прямой предок, так и любой другой класс, расположенный выше в дереве наследования. Другой термин, обозначающий базовый класс, – *суперкласс*. Мы будем использовать термины *подкласс*, *порожденный класс* и *дочерний класс* для обозначения класса, наследующего (то есть специализирующего) другой класс. В языке Python все встроенные и библиотечные классы, а также все созданные нами классы прямо или косвенно на-

[1] Некоторые библиотечные классы, реализованные на языке C, не могут быть специализированы. Такая особенность этих классов обязательно подчеркивается в документации.

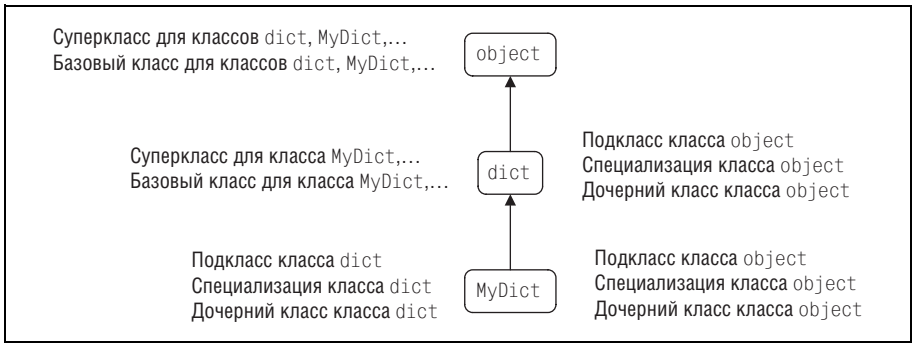

Рис. 6.1. *Некоторые термины, используемые при описании механизма наследования*

следуют единый базовый класс `object`. Рис. 6.1 иллюстрирует некоторые термины, используемые при описании механизма наследования.

Любой метод можно переопределить, то есть повторно реализовать в подклассе, как в языке Java (за исключением методов со спецификатором `final`).[1] Если предположить, что имеется объект класса `MyDict` (наследующего класс `dict`) и производится вызов метода, который определяется обоими классами `dict` и `MyDict`, интерпретатор корректно вызовет версию метода для класса `MyDict`. Этот механизм называется *динамическим связыванием методов* или *полиморфизмом*. Если возникнет необходимость вызвать версию метода базового класса внутри одноименного метода подкласса, сделать это можно с помощью встроенной функции `super()`.

Кроме того, в языке Python используется механизм *грубого определения типа* (так называемая *утиная типизация*) – «если это ходит как утка и крякает как утка, значит, это утка». Говоря другими словами, если нам необходимо вызвать определенный метод объекта, то неважно, к какому классу относится этот объект, главное, чтобы он имел метод, который предполагается вызвать. В предыдущей главе мы видели, что, когда возникала необходимость в объекте файла, мы могли получить его, вызвав функцию `open()` или создав объект `io.StringIO`, который имеет тот же самый API (Application Programming Interface – прикладной программный интерфейс), то есть обладает теми же самыми методами, что и объект, возвращаемый функцией `open()`, открывающей файл в текстовом режиме.

Механизм наследования используется для моделирования отношений типа «является», то есть отношения, когда объекты одного класса по

[1] В терминологии языка C++ все методы классов в языке Python являются виртуальными.

существу являются теми же самыми, что и объекты какого-то другого класса, но с некоторыми отличиями, такими как дополнительные атрибуты данных или дополнительные методы. Другой подход основан на использовании механизма *агрегирования* (или *композиции*) – когда класс включает одну или более переменных экземпляра, являющихся экземплярами других классов. Механизм агрегирования используется для моделирования отношений типа «имеет». В языке Python при создании любых классов используется механизм наследования, потому что все классы в конечном итоге имеют единый базовый класс `object`, и, кроме того, в большинстве классов используется механизм агрегирования, потому что в большинстве классов имеются переменные экземпляров различных типов.

Некоторые объектно-ориентированные языки программирования обладают двумя особенностями, отсутствующими в языке Python. Первая особенность – это перегрузка, то есть возможность иметь в одном и том же классе несколько методов с одинаковыми именами, но с различными списками входных параметров. Благодаря наличию в языке Python очень гибкого механизма передачи аргументов отсутствие возможности перегрузки практически не является ограничением. Вторая особенность – управление доступом; в языке Python не существует абсолютно надежных механизмов защиты частных данных. Однако если мы создаем атрибуты (переменные экземпляра или методы), имена которых начинаются двумя символами подчеркивания, интерпретатор будет предотвращать неумышленные попытки доступа к ним, так что эти атрибуты можно считать частными. (Делается это посредством подмены имен, как будет показано на примере в главе 8.)

Аналогично тому, как мы первый символ имени наших собственных модулей писали в верхнем регистре, мы будем поступать и при именовании наших собственных классов. Мы можем определить любое число классов, как в самой программе, так и в модулях. Имена классов не обязательно должны соответствовать именам модулей, и модули могут содержать столько определений классов, сколько нам потребуется.

Теперь, когда мы рассмотрели некоторые проблемы, которые могут быть решены с помощью классов, познакомились с необходимыми терминами и некоторыми основами, – можно приступать к созданию собственных классов.

Собственные классы

В предыдущих главах нам уже приходилось создавать собственные классы: наши собственные исключения. Ниже приводится синтаксис, используемый при создании собственных классов:

```
class className:
    suite
```

```
class className(base_classes):
    suite
```

Поскольку при создании подклассов исключений мы не добавляли никаких новых атрибутов (данных экземпляра или методов), в качестве блока кода (suite) мы использовали инструкцию `pass` (то есть ничего не добавляли), а так как блок кода состоял из единственной инструкции, мы помещали его в одной строке с инструкцией `class`. Обратите внимание: как и инструкция `def`, инструкция `class` является самой обычной инструкцией, что дает возможность создавать классы динамически, когда в этом возникнет необходимость. Методы класса создаются с помощью инструкций `def` внутри блока кода класса. Экземпляры класса создаются посредством обращения к имени класса, как к функции, которой передаются все необходимые аргументы. Например, инструкция `x = complex(4, 8)` создаст комплексное число и запишет ссылку на него в переменную `x`.

Атрибуты и методы

Начнем с очень простого класса `Point`, который хранит координаты точки (x, y). Определение класса находится в файле *Shape.py*, а ниже приводится его полная реализация (за исключением строк документирования):

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return math.hypot(self.x, self.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __repr__(self):
        return "Point({0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

Поскольку базовый класс не был указан явно, класс `Point` является прямым наследником класса `object`, как если бы было записано определение `class Point(object)`. Прежде чем приступать к обсуждению всех его методов, рассмотрим несколько примеров их использования:

```
import Shape
a = Shape.Point()
repr(a)                     # вернет: 'Point(0, 0)'
b = Shape.Point(3, 4)
str(b)                      # вернет: '(3, 4)'
b.distance_from_origin()    # вернет: 5.0
b.x = -19
```

```
str(b)                      # вернет: '(-19, 4)'
a == b, a != b              # вернет: (False, True)
```

Класс `Point` имеет два атрибута данных, `self.x` и `self.y`, и пять методов (не считая унаследованных методов), из которых четыре являются специальными методами – они показаны на рис. 6.2. После импортирования модуля `Shape` появляется возможность использовать класс `Point`, как любой другой класс. Доступ к атрибутам можно осуществлять непосредственно (например, `y = a.y`), а сам класс отлично интегрируется со всеми остальными классами языка Python, обеспечивая поддержку оператора равенства (`==`) и представления класса в репрезентативной и строковой формах. Интерпретатор Python достаточно умен, чтобы обеспечить поддержку оператора неравенства (`!=`) на основе имеющейся поддержки оператора равенства. (Однако имеется возможность реализовать поддержку каждого оператора в отдельности, если потребуется обеспечить полный контроль, когда, к примеру, один оператор не является полной противоположностью другому.)

При вызове метода интерпретатор автоматически передает ему первый аргумент – ссылку на сам объект (в языках C++ и Java она имеет имя *this*). В соответствии с соглашениями мы обязаны включать этот параметр в список под именем `self`. Все атрибуты объекта (данные и методы) должны квалифицироваться именем `self`. При этом потребуется вводить с клавиатуры чуть больше, чем в других языках программирования, но в этом есть свое преимущество – полная ясность: мы всегда точно знаем, что обращаемся к атрибуту объекта, если квалифицируем его именем `self`.

Чтобы создать объект, необходимо выполнить два действия. Сначала необходимо создать неинициализированную заготовку объекта, а затем необходимо подготовить объект к использованию, инициализировав

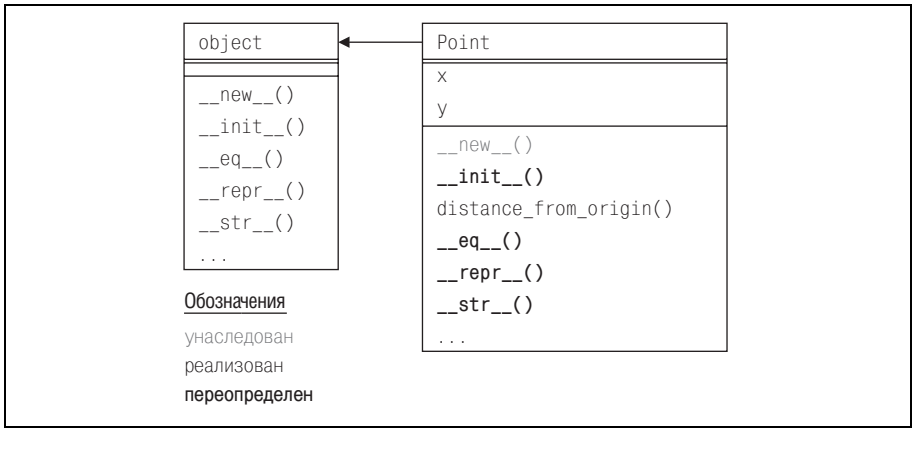

Рис. 6.2. *Дерево наследования класса Point*

его. В некоторых языках программирования (таких как C++ и Java) эти два действия объединены в одно, но в языке Python они выполняются отдельно друг от друга. Когда создается объект (например, `p = Shape.Point()`), то сначала вызывается специальный метод `__new__()`, который создает объект, а затем выполняется инициализация объекта вызовом специального метода `__init__()`.

Альтернативный тип FuzzyBool, стр. 300

В языке Python при создании практически любого класса нам будет необходимо переопределять только метод `__init__()`, поскольку имеющейся реализации метода `object.__new__()` почти всегда достаточно, и к тому же он вызывается автоматически, если мы не предусматриваем собственную реализацию метода `__new__()`. (Ниже в этой главе мы увидим пример одного из редких случаев, когда возникает необходимость переопределить метод `__new__()`.) Отсутствие необходимости переопределять методы в подклассе – это еще одно преимущество объектно-ориентированного программирования. Если метод базового класса удовлетворяет нашим потребностям, мы можем не переопределять его в своем классе. Если при обращении к методу объекта окажется, что класс объекта не реализует его, интерпретатор автоматически попытается отыскать его в базовых классах объекта, а затем в их базовых классах, и так до тех пор, пока не найдет требуемый метод, а если метод не будет обнаружен, он возбудит исключение `AttributeError`.

Например, если попробовать выполнить инструкцию `p = Shape.Point()`, интерпретатор начнет поиск метода `Point.__new__()`. Поскольку мы не переопределяли этот метод, интерпретатор попытается отыскать этот метод в базовом классе класса `Point`. В данном случае существует всего один базовый класс, `object`, который имеет требуемый метод, поэтому интерпретатор вызовет метод `object.__new__()` и создаст неинициализированную заготовку объекта. Затем интерпретатор приступит к поиску метода инициализации, `__init__()`, и поскольку мы предусмотрели его реализацию, интерпретатору не потребуется искать его в базовых классах и он вызовет метод `Point.__init__()`. В заключение интерпретатор запишет в переменную `p` ссылку на вновь созданный и инициализированный объект типа `Point`.

Поскольку методы очень короткие и к тому же они приводились за несколько страниц отсюда, для удобства мы приведем снова каждый метод перед обсуждением.

```
def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
```

В методе инициализации создаются две переменные экземпляра, `self.x` и `self.y`, которым присваиваются значения параметров `x` и `y`. Поскольку

ку при создании нового объекта класса Point интерпретатор сразу же обнаружит этот метод, он не будет автоматически вызывать метод object.__init__(). Как только интерпретатор обнаруживает необходимый метод, он сразу же вызывает его, прекращая дальнейшие поиски.

Пуристы объектно-ориентированного программирования могли бы начать реализацию своего метода с вызова метода __init__() базового класса, обращением к super().__init__(). Действие вызова такой функции super() заключается в вызове метода __init__() базового класса. Для классов, порожденных непосредственно от класса object, в этом нет никакой необходимости, и в этой книге мы будем вызывать методы базовых классов, только когда это действительно нужно – например, при создании классов, которые должны будут наследоваться, или при создании классов, не являющихся непосредственными наследниками класса object. Отчасти это вопрос стиля, но тем не менее совершенно разумно – всегда начинать метод __init__() своего класса с вызова super().__init__().

```
def distance_from_origin(self):
    return math.hypot(self.x, self.y)
```

Это обычный метод, выполняющий вычисления на основе переменных экземпляра объекта. Для методов весьма характерно иметь небольшой размер и получать в виде параметров только объект, в контексте которого они вызываются, поскольку нередко все данные, необходимые методу, доступны внутри объекта.

```
def __eq__(self, other):
    return self.x == other.x and self.y == other.y
```

Имена методов не должны начинаться и заканчиваться двумя символами подчеркивания, если они не являются предопределенными специальными методами. В языке Python каждому оператору сравнения соответствует свой специальный метод, как показано в табл. 6.1.

Таблица 6.1. Специальные методы сравнивания

Специальный метод	Пример использования	Описание
__lt__(self, other)	x < y	Возвращает True, если x меньше, чем y
__le__(self, other)	x <= y	Возвращает True, если x меньше или равно y
__eq__(self, other)	x == y	Возвращает True, если x равно y
__ne__(self, other)	x != y	Возвращает True, если x не равно y
__ge__(self, other)	x >= y	Возвращает True, если x больше или равно y
__gt__(self, other)	x > y	Возвращает True, если x больше, чем y

Все экземпляры классов по умолчанию поддерживают оператор == и операция сравнения всегда возвращает False. Мы можем переопределить это поведение, реализовав специальный метод __eq__(), как это было сделано в данном случае. Интерпретатор Python будет автоматически подставлять метод __ne__() (not equal – не равно), реализующий действие оператора неравенства (!=), если в классе присутствует реализация метода __eq__(), но отсутствует реализация метода __ne__().

Тип FuzzyBool, стр. 292

По умолчанию все экземпляры классов являются хешируемыми, поэтому для них можно вызывать функцию hash(), использовать их в качестве ключей словаря и сохранять в множествах. Но если будет реализован метод __eq__(), экземпляры перестанут быть хешируемыми. Как исправить это положение, будет показано при обсуждении класса FuzzyBool ниже.

Реализовав этот специальный метод, мы получаем возможность сравнивать объекты Point, но при попытке сравнить объект Point с объектом другого типа, например, int, будет возбуждено исключение AttributeError (поскольку объекты класса int не имеют атрибута x). С другой стороны, мы *можем* сравнивать объекты Point с другими объектами совместимых типов, у которых имеется атрибут x (благодаря грубому определению типов в языке Python), но это может приводить к неожиданным результатам.

Если необходимо избежать сравнения в случаях, когда это не имеет смысла, можно использовать несколько подходов. Один из них состоит в использовании инструкции assert, например, assert isinstance(other, Point). Другой состоит в том, чтобы возбуждать исключение TypeError для обозначения попытки сравнения с неподдерживаемым типом, например, if not isinstance(other, Point): raise TypeError(). Третий способ (который, с точки зрения языка Python, является наиболее правильным) заключается в следующем: if not isinstance(other, Point): return NotImplemented. В этом третьем случае, когда метод возвращает NotImplemented, интерпретатор попытается вызвать метод other.__eq__(self), чтобы определить, поддерживает ли тип other сравнение с типом Point, и если в этом типе не будет обнаружен такой метод или он также возвращает NotImplemented, интерпретатор возбудит исключение TypeError. (Обратите внимание, что значение NotImplemented может вернуть только переопределенный специальный метод сравнения – из тех, что перечислены в табл. 6.1.)

Встроенная функция isinstance() принимает объект и класс (или кортеж классов) и возвращает True, если объект принадлежит данному классу (или одному из классов, перечисленных в кортеже) или одному из базовых классов указанного класса (или одного из классов, перечисленных в кортеже).

```
    def __repr__(self):
        return "Point({0.x!r}, {0.y!r})".format(self)
```

Встроенная функция repr() вызывает специальный метод __repr__() указанного объекта и возвращает его результат. Возвращаемая строка может быть одного из двух видов. Один вид – когда возвращаемая строка с помощью функции eval() может быть преобразована в объект, эквивалентный тому, что был передан функции repr(). Второй вид используется, когда такое преобразование невозможно. Примеры таких ситуаций будут показаны позднее. Ниже показано, как можно выполнить преобразование объекта Point в строку и обратно – в объект Point:

Метод str.format(), стр. 100

```
p = Shape.Point(3, 9)
repr(p)                                  # вернет: 'Point(3, 9)'
q = eval(p.__module__ + "." + repr(p))
repr(q)                                  # вернет: 'Point(3, 9)'
```

При вызове функции eval() мы должны передать имя модуля, если использовалась инструкция import Shape. (Это не требуется, если импортирование выполнялось иным способом, например, from Shape import Point.) Каждому объекту интерпретатор Python присваивает несколько частных атрибутов, один из которых __module__ – строка, хранящая имя модуля объекта, в данном случае "Shape".

Инструкция import, стр. 230

После выполнения этого фрагмента в нашем распоряжении будет два объекта класса Point, p и q, с одинаковыми значениями атрибутов, поэтому операция сравнения говорит о том, что они равны. Функция eval() возвращает результат выполнения переданной ей строки, которая должна содержать допустимую инструкцию языка Python.

Динамическое выполнение программного кода, стр. 400

```
def __str__(self):
    return "({0.x!r}, {0.y!r})".format(self)
```

Встроенная функция str() работает точно так же, как функция repr(), за исключением того, что она вызывает специальный метод __str__() объекта. Результатом работы этого метода должна быть строка, предназначенная для восприятия человеком и которую не предполагается передавать функции eval(). Если продолжить предыдущий пример, вызов str(p) (или str(q)) вернул бы строку '(3, 9)'.

Мы закончили рассмотрение простого класса Point, а также некоторых подробностей, которые важно знать, но не обязательно применять на практике. Класс Point хранит координаты (x, y) – важную часть данных, необходимых для представления окружностей, с которых мы начали эту главу. В следующем подразделе будет показано, как создать собственный класс Circle, наследующий класс Point, чтобы нам не

приходилось дублировать программный код, создающий атрибуты x и y или метод distance_from_origin().

Наследование и полиморфизм

Класс Circle построен на основе класса Point, с использованием механизма наследования. Класс Circle добавляет один атрибут данных (radius) и три новых метода. Кроме того, он переопределяет несколько методов класса Point. Ниже приводится полное определение класса:

```
class Circle(Point):

    def __init__(self, radius, x=0, y=0):
        super().__init__(x, y)
        self.radius = radius

    def edge_distance_from_origin(self):
        return abs(self.distance_from_origin() - self.radius)

    def area(self):
        return math.pi * (self.radius ** 2)

    def circumference(self):
        return 2 * math.pi * self.radius

    def __eq__(self, other):
        return self.radius == other.radius and super().__eq__(other)

    def __repr__(self):
        return "Circle({0.radius!r}, {0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return repr(self)
```

Наследование реализуется просто, посредством перечисления класса (или классов), который должен быть унаследован нашим классом, в строке с инструкцией class.[1] В данном случае мы наследуем класс Point – иерархия дерева наследования класса Circle приводится на рис. 6.3.

Внутри метода __init__() мы используем функцию super() для вызова метода __init__() базового класса – он создает и инициализирует атрибуты self.x и self.y. Пользователи класса могут попытаться определить недопустимое значение радиуса, например –2. В следующем подразделе мы покажем, как предотвратить появление этой проблемы, для повышения устойчивости атрибутов используя свойства.

Методы area() и circumference() достаточно очевидны. Метод edge_distance_from_origin() в ходе производимых вычислений вызывает метод distance_from_origin(). Так как класс Circle не реализует свой метод

[1] Множественное наследование, абстрактные типы данных и другие, более сложные приемы объектно-ориентированного программирования рассматриваются в главе 8.

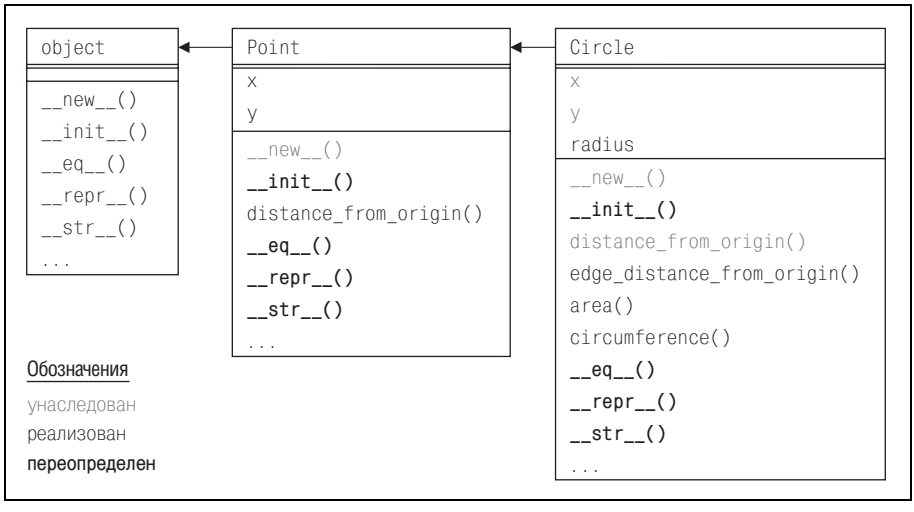

Рис. 6.3. *Дерево наследования класса Circle*

`distance_from_origin()`, интерпретатор найдет и будет использовать метод базового класса `Point`. Сравните это с переопределением метода `__eq__()`. Этот метод сравнивает радиус окружности с радиусом другой окружности, и если они равны, то при помощи функции `super()` явно вызывается метод `__eq__()` базового класса. Если бы мы не использовали функцию `super()`, мы могли бы попасть в бесконечную рекурсию, поскольку метод `Circle.__eq__()` продолжал бы вызывать сам себя. Обратите также внимание на то, что в вызов `super()` мы не передаем аргумент `self`, потому что интерпретатор сделает это автоматически.

Ниже приводится пара примеров использования:

```
p = Shape.Point(28, 45)
c = Shape.Circle(5, 28, 45)
p.distance_from_origin()       # вернет: 53.0
c.distance_from_origin()       # вернет: 53.0
```

Мы можем вызвать метод `distance_from_origin()` как для экземпляра класса `Point`, так и для экземпляра класса `Circle`, потому что класс `Circle` наследует класс `Point`.

Полиморфизм подразумевает, что любой объект данного класса может использоваться, как если бы это был объект любого из базовых его классов. По этой причине, когда создается подкласс, нам требуется реализовать только необходимые дополнительные методы и переопределить только те существующие методы, которые нам хотелось бы заменить. Переопределяя методы, мы можем в случае необходимости использовать реализацию базовых классов, применяя функцию `super()` внутри переопределяемых методов.

В случае с классом Circle мы реализовали дополнительные методы, такие как area() и circumference(), и переопределили методы, которые необходимо было изменить. Переопределить методы __repr__() и __str__() было необходимо потому, что без этого использовались бы методы базового класса, возвращающие строки с представлением класса Point, а не Circle. Переопределить методы __init__() и __eq__() было необходимо потому, что нам необходимо было учесть тот факт, что класс Circle имеет один дополнительный атрибут; и в обоих случаях была использована реализация базового класса, чтобы минимизировать объем работы, которую необходимо было выполнить.

Поверхностное и глубокое копирование, стр. 173

Классы Point и Circle можно считать полными, поскольку они соответствуют нашим требованиям. Мы могли добавить в них дополнительные методы, например, другие специальные методы сравнения, если бы нам было необходимо упорядочивать объекты классов Point и Circle. Еще можно было бы реализовать в классах Point и Circle метод копирования. В большинстве классов Python отсутствует метод copy() (за исключением dict.copy() и set.copy()). Если нам потребуется скопировать экземпляр класса Point или Circle, мы легко можем сделать это, импортировав модуль copy и использовав функцию copy.copy(). (В случае с объектами классов Point и Circle нет необходимости использовать функцию copy.deepcopy(), потому что они содержат только неизменяемые переменные экземпляра.)

Использование свойств для управления доступом к атрибутам

В предыдущем подразделе класс Point поддерживал метод distance_from_origin(), а класс Circle – методы area(), circumference() и edge_distance_from_origin(). Все эти методы возвращают единственное значение типа float, поэтому, с точки зрения пользователя классов, они точно так же могли бы быть атрибутами данных, но доступными только для чтения. В файле *ShapeAlt.py* представлена альтернативная реализация классов Point и Circle, где все упомянутые методы представляются как свойства. Это позволяет нам писать программный код, как показано ниже:

```
circle = Shape.Circle(5, 28, 45) # предполагается, что модуль ShapeAlt
                                 # был импортирован под именем Shape
circle.radius                    # вернет: 5
circle.edge_distance_from_origin # вернет: 48.0
```

Ниже приводится реализация методов чтения для свойств area и edge_distance_from_origin класса ShapeAlt.Circle:

```
    @property
    def area(self):
        return math.pi * (self.radius ** 2)

    @property
    def edge_distance_from_origin(self):
        return abs(self.distance_from_origin - self.radius)
```

Если мы реализуем только методы чтения, как это было сделано здесь, свойства будут доступны только для чтения. Программный код реализации свойства `area` остался тем же самым, что и в реализации метода `area()`. Программный код реализации свойства `edge_distance_from_origin` несколько изменился, потому что теперь он обращается к свойству `distance_from_origin` базового класса, а не к методу `distance_from_origin()`. Самое заметное отличие между реализациями заключается в наличии *декоратора* `property`. Декоратор – это функция, которая в качестве аргумента принимает функцию или метод и возвращает «декорированную» версию, то есть версию функции или метода, измененную некоторым способом. Декоратор обозначается первым символом «@» в имени. Пока просто воспринимайте декораторы как элемент синтаксиса – в главе 8 будет показано, как можно создавать собственные декораторы.

Функция-декоратор `property()` – это встроенная функция, и она может принимать до четырех аргументов: функцию чтения, функцию записи, функцию удаления и строку документирования. Фактически использование имени `@property` равносильно вызову функции `property()` с единственным аргументом – функцией чтения. Мы могли бы создать свойство `area`, как показано ниже:

```
    def area(self):
        return math.pi * (self.radius ** 2)
    area = property(area)
```

Мы редко используем такой синтаксис, потому что использование декоратора выглядит короче и понятнее.

В предыдущем подразделе мы отмечали отсутствие проверки значений, записываемых в атрибут `radius` класса `Circle`. Мы можем реализовать такую проверку, преобразовав атрибут `radius` в свойство. Для этого не потребуется изменять реализацию метода `Circle.__init__()`; любой другой программный код, обращающийся к атрибуту `Circle.radius`, будет продолжать корректно работать, только теперь значения будут проходить проверку при записи.

Как правило, программисты, создающие программы на языке Python, используют свойства, а не явные методы чтения и записи (например, `getRadius()` и `setRadius()`), которые обычно используются в других языках программирования. Это обусловлено тем, что атрибут данных очень легко можно превратить в свойство, что никак не скажется на программном коде, использующем класс.

Чтобы превратить атрибут в свойство, доступное для чтения и записи, нам необходимо создать частный атрибут, который будет являться фактическим хранилищем данных и будет использоваться методами чтения и записи. Ниже приводится полная реализация методов чтения и записи, а также строка документирования:

```
@property
def radius(self):
    """Радиус окружности

    >>> circle = Circle(-2)
    Traceback (most recent call last):
    ...
    AssertionError: radius must be nonzero and non-negative
    >>> circle = Circle(4)
    >>> circle.radius = -1
    Traceback (most recent call last):
    ...
    AssertionError: radius must be nonzero and non-negative
    >>> circle.radius = 6
    """
    return self.__radius

@radius.setter
def radius(self, radius):
    assert radius > 0, "radius must be nonzero and non-negative"
    self.__radius = radius
```

Чтобы убедиться, что записываемое значение радиуса больше нуля, используется инструкция `assert`; после проверки значение радиуса сохраняется в частном атрибуте `self.__radius`. Примечательно, что методы чтения и записи (и метод удаления, если бы он нам потребовался) имеют одно и то же имя – они отличаются только декораторами, и декораторы соответствующим образом переименовывают методы, чтобы исключить конфликты имен.

Декоратор метода записи может показаться немного необычным на первый взгляд. Каждое создаваемое свойство имеет атрибут `getter`, `setter` или `deleter`, поэтому, как только свойство `radius` будет создано, появятся атрибуты `radius.getter`, `radius.setter` и `radius.deleter`. В атрибут `radius.getter` декоратором `@property` записывается ссылка на метод чтения. Другие два атрибута устанавливаются интерпретатором так, что они ничего не делают (поэтому в атрибут ничего нельзя записать или удалить его), если они не были использованы как декораторы; тогда они замещаются декорируемыми ими методами.

Метод инициализации `Circle.__init__()` содержит инструкцию `self.radius = radius`. При выполнении она превратится в вызов метода записи для свойства `radius`, поэтому, если при создании объекта `Circle` будет указано недопустимое значение, будет возбуждено исключение `AssertionError`. Точно так же, если будет произведена попытка установить недопустимое значение свойства `radius` у существующего объекта

класса Circle, снова будет вызван метод записи, который возбудит исключение. Строка документирования включает в себя доктесты, проверяющие корректное возбуждение исключений в этих случаях.

Типы Point и Circle являются нашими собственными типами данных, обладающими достаточным объемом функциональных возможностей, чтобы быть полезными. Большинство типов данных, которые нам придется создавать, будут похожи на эти типы данных, но иногда будет возникать необходимость в самостоятельном создании собственного полного типа данных. Пример такого типа данных мы увидим в следующем подразделе.

Создание полных и полностью интегрированных типов данных

В создании полного типа данных можно пойти двумя путями. Первый состоит в том, чтобы создать тип данных с самого начала. Хотя тип данных будет наследовать класс object (как и любой другой класс Python), тем не менее придется реализовать все атрибуты данных и методы (за исключением метода __new__()). Другой путь состоит в том, что наследовать существующий тип данных, напоминающий тот, что мы собираемся создать. В этом случае основная работа обычно связана с переопределением тех методов, поведение которых необходимо изменить, и с «ликвидацией» тех методов, которые вообще являются нежелательными.

В следующем подразделе мы реализуем тип данных FuzzyBool, начав с нуля, а в подразделе, следующем за ним, мы реализуем тот же самый тип данных, но при этом воспользуемся механизмом наследования, чтобы уменьшить объем работы, которую необходимо выполнить. Встроенный тип bool имеет два возможных значения (True и False), но в некоторых областях ИИ (искусственный интеллект) используется нечеткая логика, опирающаяся на значения, соответствующие понятиям «истина» и «ложь», а также на промежуточные между ними. В наших реализациях мы будем использовать значения с плавающей точкой, где 0.0 будет соответствовать значению False, а 1.0 – значению True. В этой системе координат значение 0.5 будет обозначать 50-процентную истинность, 0.25 – 25-процентную истинность и т. д. Ниже приводятся несколько примеров использования (они работают совершенно одинаково с любой из двух реализаций):

```
a = FuzzyBool.FuzzyBool(.875)
b = FuzzyBool.FuzzyBool(.25)
a >= b                              # вернет: True
bool(a), bool(b)                    # вернет: (True, False)
~a                                  # вернет: FuzzyBool(0.125)
a & b                               # вернет: FuzzyBool(0.25)
b |= FuzzyBool.FuzzyBool(.5)        # теперь b имеет значение: FuzzyBool(0.5)
"a={0:.1%} b={1:.0%}".format(a, b) # вернет: 'a=87.5% b=50%'
```

Нам необходимо, чтобы тип FuzzyBool поддерживал полный набор операторов сравнения (<, <=, ==, !=, >=, >) и три основные логические операции: НЕ (~), И (&) и ИЛИ (|). В дополнение к логическим операциям нам необходимо реализовать пару других логических методов – conjunction() и disjunction(), способных принимать произвольное число значений типа FuzzyBool и возвращающих соответствующие результаты типа FuzzyBool. И для полноты типа нам потребуется реализовать возможность преобразования в типы bool, int, float и str, а также обеспечить получение репрезентативной формы, совместимой с функцией eval(). Наконец, тип FuzzyBool должен поддерживать спецификаторы формата метода str.format(), он должен иметь возможность использоваться в качестве ключей словаря или членов множеств, значения типа FuzzyBool должны быть неизменяемыми, при условии поддержки комбинированных операторов присваивания (&= и |=), чтобы обеспечить дополнительные удобства в использовании.

В табл. 6.1 (стр. 283) перечислены специальные методы операций сравнения, в табл. 6.2 (стр. 294) перечислены фундаментальные специальные методы и в табл. 6.3 (стр. 296) перечислены арифметические специальные методы, включая методы реализации битовых операторов (~, & и |), которые применительно к типу FuzzyBool играют роль логических операторов, а также арифметические операторы + и -, которые в типе FuzzyBool не будут реализованы, как не имеющие смысла.

Создание типов данных с нуля

Создание типа данных FuzzyBool с нуля означает, что мы должны создать атрибут для хранения значения типа FuzzyBool и все необходимые методы. Ниже приводится инструкция class и метод инициализации, взятые из файла *FuzzyBool.py*:

```
class FuzzyBool:

    def __init__(self, value=0.0):
        self.__value = value if 0.0 <= value <= 1.0 else 0.0
```

Свойство radius класса ShapeAlt.Circle, стр. 289

Мы сделали атрибут частным, потому что нам необходимо, чтобы тип FuzzyBool вел себя как неизменяемый объект, для которого было бы неправильно разрешать прямой доступ к атрибуту. Кроме того, если в аргументе value получено число, находящееся вне диапазона допустимых значений, мы принудительно замещаем его значением по умолчанию 0.0 (ложь). В предыдущем подразделе, в классе ShapeAlt.Circle, мы использовали политику строгого ограничения, возбуждая исключение при получении недопустимых значений радиуса во время создания нового объекта Circle. Дерево наследования класса FuzzyBool приводится на рис. 6.4.

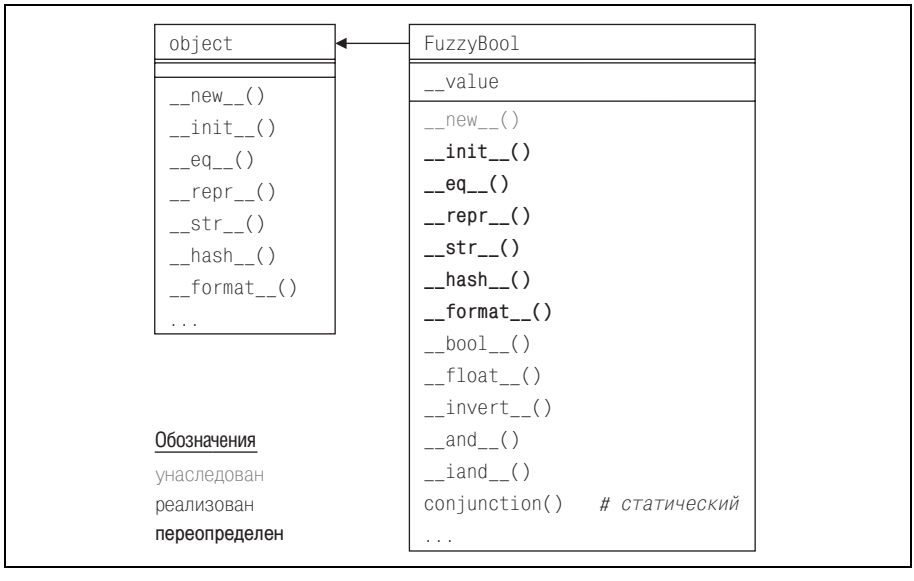

Рис. 6.4. *Дерево наследования класса FuzzyBool*

Простейшим логическим оператором является логическое НЕ, в качестве которого мы будем использовать битовый оператор инверсии (~):

```
def __invert__(self):
    return FuzzyBool(1.0 - self.__value)
```

Битовый и логический оператор И (&) реализуется специальным методом `__and__()`, а соответствующий ему комбинированный оператор присваивания (&=) – методом `__iand__()`:

```
def __and__(self, other):
    return FuzzyBool(min(self.__value, other.__value))

def __iand__(self, other):
    self.__value = min(self.__value, other.__value)
    return self
```

Логический оператор И возвращает новый объект `FuzzyBool`, основываясь на значениях объектов `self` и `other`, тогда как комбинированный оператор присваивания изменяет значение частного атрибута. Строго говоря, такое поведение не совсем свойственно неизменяемым объектам, но оно совпадает с поведением некоторых других неизменяемых типов языка Python, таких как `int`. Например, при использовании оператора += создается впечатление, что изменяется операнд слева, но в действительности выполняется перепривязка ссылки на новый объект `int`, хранящий результат операции сложения; хотя в случае `FuzzyBool` перепривязку выполнять не требуется, так как мы действительно изменяем сам объект. Причина, по которой метод возвращает значе-

ние self, заключается в необходимости обеспечения возможности объединять операции в цепочку.

Таблица 6.2. Фундаментальные специальные методы

Специальный метод	Пример использования	Описание
`__bool__(self)`	`bool(x)`	Если реализован, возвращает значение истинности для x. Удобно, если используются конструкции вида `if x: ...`
`__format__(self, format_spec)`	`"{0}".format(x)`	Обеспечивает поддержку метода `str.format()` для классов
`__hash__(self)`	`hash(x)`	Если реализован, x сможет использоваться как ключ словаря или храниться в множестве
`__init__(self, args)`	`x = X(args)`	Вызывается при инициализации объекта
`__new__(cls, args)`	`x = X(args)`	Вызывается при создании объекта Переопределение метода __new__(), стр. 300
`__repr__(self)`	`repr(x)`	Возвращает строку с репрезентативной формой представления x, которая обеспечивает равенство `eval(repr(x)) == x`
`__repr__(self)`	`ascii(x)`	Возвращает строку с репрезентативной формой представления x с использованием только символов набора ASCII Метод str.format(), стр. 103
`__str__(self)`	`str(x)`	Возвращает строковое представление x, пригодное для восприятия человеком

Мы могли бы также реализовать метод `__rand__()`. Этот метод вызывается в случае, когда объекты self и other принадлежат разным типам, а метод `__and__()` для данной пары типов не реализован.[1] В классе FuzzyBool в этом нет никакой необходимости. Для большинства двухместных операторов имеются специальные методы двух версий: «i» (in-place – изменяется сам объект) и «r» (reflect, то есть производится обмен операндов местами).

Мы не показываем реализации методов `__or__()`, соответствующий логическому оператору |, и `__ior__()`, соответствующий комбинированному оператору присваивания |=, потому что они полностью эквивалентны методам реализации операции И, за исключением того, что в результате возвращается не минимальное, а максимальное значение пары self и other.

[1] То есть возвращает значение NotImplemented. – *Прим. перев.*

```
    def __repr__(self):
        return ("{0}({1})".format(self.__class__.__name__,
```

Мы предусмотрели реализацию метода `__repr__()`, воспроизводящего репрезентативную форму представления. Например, последовательность инструкций `f = FuzzyBool.FuzzyBool(.75); repr(f)` будет воспроизводить строку `'FuzzyBool(0.75)'`.

Все объекты имеют ряд специальных атрибутов, автоматически создаваемых интерпретатором, один из которых называется `__class__` и содержит ссылку на класс объекта. Все классы обладают частным атрибутом `__name__`, который также создается автоматически. Мы используем эти атрибуты для получения имени класса в репрезентативной форме представления. Это означает, что если от класса `FuzzyBool` будет порожден дочерний класс, добавляющий дополнительные методы, унаследованный метод `__repr__()` будет продолжать корректно работать и в контексте подкласса, потому что он будет получать имя класса для этого подкласса.

```
    def __str__(self):
        return str(self.__value)
```

Специальный метод __del__()

Специальный метод `__del__(self)` вызывается при уничтожении объекта – по крайней мере в теории. На практике метод `__del__()` может не вызываться никогда, даже при завершении программы. Более того, когда выполняется инструкция `del x`, все, что происходит при этом, – удаляется ссылка на объект `x` и уменьшается счетчик ссылок, указывающих на объект `x`. Только когда этот счетчик достигает значения 0, есть вероятность, что метод `__del__()` будет вызван, но интерпретатор Python не дает никаких гарантий, что этот метод будет когда-нибудь вызван. По этой причине метод `__del__()` очень редко переопределяется – он не переопределяется ни в одном из примеров в этой книге, и он не должен использоваться для освобождения ресурсов, для закрытия файлов, сетевых соединений или подключений к базам данных.

Язык Python предоставляет два отдельных механизма, при использовании которых можно реализовать корректное освобождение ресурсов. Один из них заключается в использовании блоков `try ... finally`, как это было показано ранее и как это будет еще показано в главе 7. Другой механизм основан на использовании контекста объекта в соединении с инструкцией `with`, о которой будет рассказываться в главе 8.

Таблица 6.3. Арифметические и битовые специальные методы

Специальный метод	Пример использования	Специальный метод	Пример использования
`__abs__(self)`	`abs(x)`	`__complex__(self)`	`complex(x)`
`__float__(self)`	`float(x)`	`__int__(self)`	`int(x)`
`__index__(self)`	`bin(x) oct(x) hex(x)`	`__round__(self, digits)`	`round(x, digits)`
`__pos__(self)`	`+x`	`__neg__(self)`	`-x`
`__add__(self, other)`	`x + y`	`__sub__(self, other)`	`x - y`
`__iadd__(self, other)`	`x += y`	`__isub__(self, other)`	`x -= y`
`__radd__(self, other)`	`y + x`	`__rsub__(self, other)`	`y - x`
`__mul__(self, other)`	`x * y`	`__mod__(self, other)`	`x % y`
`__imul__(self, other)`	`x *= y`	`__imod__(self, other)`	`x %= y`
`__rmul__(self, other)`	`y * x`	`__rmod__(self, other)`	`y % x`
`__floordiv__(self, other)`	`x // y`	`__truediv__(self, other)`	`x / y`
`__ifloordiv__(self, other)`	`x //= y`	`__itruediv__(self, other)`	`x /= y`
`__rfloordiv__(self, other)`	`y // x`	`__rtruediv__(self, other)`	`y / x`
`__divmod__(self, other)`	`divmod(x, y)`	`__rdivmod__(self, other)`	`divmod(y, x)`
`__pow__(self, other)`	`x ** y`	`__and__(self, other)`	`x & y`
`__ipow__(self, other)`	`x **= y`	`__iand__(self, other)`	`x &= y`
`__rpow__(self, other)`	`y ** x`	`__rand__(self, other)`	`y & x`
`__xor__(self, other)`	`x ^ y`	`__or__(self, other)`	`x | y`
`__ixor__(self, other)`	`x ^= y`	`__ior__(self, other)`	`x |= y`
`__rxor__(self, other)`	`y ^ x`	`__ror__(self, other)`	`y | x`
`__lshift__(self, other)`	`x << y`	`__rshift__(self, other)`	`x >> y`
`__ilshift__(self, other)`	`x <<= y`	`__irshift__(self, other)`	`x >>= y`
`__rlshift__(self, other)`	`y << x`	`__rrshift__(self, other)`	`y >> x`
		`__invert__(self)`	`~x`

В качестве строковой формы представления мы просто возвращаем значение с плавающей точкой, преобразованное в строку. Мы не использовали функцию `super()`, чтобы избежать попадания в бесконечную рекурсию, и вызываем функцию `str()`, передавая ей атрибут `self.__value`, а не сам экземпляр объекта.

```
    def __bool__(self):
        return self.__value > 0.5

    def __int__(self):
        return round(self.__value)

    def __float__(self):
        return self.__value
```

Специальный метод `__bool__()` преобразует экземпляр в тип `bool`, то есть он всегда должен возвращать либо `True`, либо `False`. Специальный метод `__int__()` реализует преобразование в целое число. Мы использовали здесь встроенную функцию `round()`, потому что функция `int()` просто усекает дробную часть (поэтому для любого значения `FuzzyBool`, кроме `1.0`, метод всегда возвращал бы значение `0`). Преобразование в число с плавающей точкой выполняется очень просто, потому что само значение уже является числом с плавающей точкой.

```
    def __lt__(self, other):
        return self.__value < other.__value

    def __eq__(self, other):
        return self.__value == other.__value
```

Чтобы обеспечить полную поддержку всех операторов сравнения (`<`, `<=`, `==`, `!=`, `>=`, `>`), необходимо реализовать хотя бы три из них: `<`, `<=` и `==`, потому что интерпретатор сможет вывести действие оператора `>` из оператора `<`, `!=` – из `==` и `>=` – из `<=`. Мы привели реализацию лишь двух методов, потому что все они очень похожи между собой.[1]

Полная поддержка всего набора операторов сравнения, стр. 439

```
    def __hash__(self):
        return hash(id(self))
```

По умолчанию экземпляры наших собственных классов поддерживают оператор `==` (который всегда возвращает `False`) и являются хешируемыми (поэтому они могут использоваться в качестве ключей словаря или добавляться в множества). Но если реализовать специальный метод `__eq__()`, выполняющий корректную проверку на равенство, экземпляры перестанут быть хешируемыми. Это можно исправить, реализовав специальный метод `__hash__()`, что мы и сделали.

Язык Python предоставляет функцию хеширования строк, чисел, фиксированных множеств и других классов. Здесь мы просто воспользовались встроенной функцией `hash()` (которая может работать с любым типами данных, имеющими специальный метод `__hash__()`) и передаем ей уникальный идентификатор объекта, на основании которого вычисляется хеш-значение. (Мы не можем использовать частный

[1] В действительности мы реализовали лишь два метода, `__lt__()` и `__eq__()`, приведенные здесь, – остальные методы сравнения генерируются автоматически, как будет показано в главе 8.

атрибут `self.__value`, потому что он может изменяться комбинированными операторами присваивания, а хеш-значение никогда не должно изменяться.)

Встроенная функция `id()` возвращает уникальное целое число для объекта, который передается в виде аргумента. Обычно этим целым числом является адрес объекта в памяти, однако мы можем только предполагать, что в программе не может существовать двух объектов с одинаковыми числовыми идентификаторами. Функция `id()` используется внутри реализации оператора `is`, который определяет, указывают ли две ссылки на один и тот же объект.

```
def __format__(self, format_spec):
    return format(self.__value, format_spec)
```

Встроенная функция `format()` – единственная действительно необходимая функция в объявлениях классов. Она принимает единственный объект и необязательную спецификацию формата и возвращает строку с объектом, отформатированным соответствующим образом.

Примеры использования объектов типа FuzzyBool, стр. 291

Когда объект используется в строке формата, вызывается метод `__format__()` объекта с самим объектом и спецификацией формата в виде аргументов. Метод возвращает строку с экземпляром, отформатированным соответствующим образом, как было показано ранее.

Все встроенные классы имеют соответствующие методы `__format__()`. В данном случае мы использовали метод `float.__format__()`, передавая значение с плавающей точкой и полученную строку формата. Того же эффекта можно было бы добиться другим способом:

```
def __format__(self, format_spec):
    return self.__value.__format__(format_spec)
```

При использовании встроенной функции `format()` немного уменьшается объем ввода с клавиатуры, и программный код выглядит более очевидно. Никто не заставляет нас использовать функцию `format()`, поэтому мы могли бы изобрести свой собственный язык форматирования и интерпретировать его внутри метода `__format__()`; главное – чтобы он возвращал строку.

```
@staticmethod
def conjunction(*fuzzies):
    return FuzzyBool(min([float(x) for x in fuzzies]))
```

Встроенная функция `staticmethod()` предназначена для использования в качестве декоратора, как видно из этого объявления. Статические методы – это обычные методы, которые *не* получают аргумент `self` или любой другой первый аргумент, который автоматически передавался бы интерпретатором Python.

Оператор `&` допускает объединение в цепочки так, чтобы, например, значения `f`, `g` и `h` типа `FuzzyBool` могли быть объединены в одном выра-

жении `f & g & h`. Такой способ удобно использовать при небольшом числе объектов `FuzzyBool`, но когда в выражении участвует десяток операндов или более, такой порядок вычислений становится слишком неэффективным, поскольку вызов функции производится для каждого оператора `&`. Благодаря методу, который определен здесь, мы можем получить тот же результат, используя единственный вызов функции `FuzzyBool.FuzzyBool.conjunction(f, g, h)`. Этот вызов можно переписать более кратко, использовав экземпляр `FuzzyBool`, но поскольку статические методы не получают аргумент `self`, то при вызове метода относительно экземпляра и в выражении участвует сам экземпляр, мы должны передавать его явно, например, `f.conjunction(f, g, h)`.

Мы не показали соответствующую реализацию метода `disjunction()`, так как он отличается только именем и тем, что вместо функции `min()` использует функцию `max()`.

Некоторые программисты считают использование статических методов несвойственным языку Python и используют их только при переносе программ с других языков программирования (таких как C++ или Java) или если метод не использует аргумент `self`. В языке Python вместо статических методов лучше создавать функции модуля, как будет показано в следующем подразделе, или методы класса, как будет показано в последнем разделе.

Точно так же, когда переменная создается в пределах класса, но за пределами какого-либо метода, она становится статической переменной (переменной класса). В качестве констант обычно более удобно использовать частные глобальные переменные модуля, а переменные класса часто бывают полезны, когда необходимо хранить информацию, общую для всех экземпляров класса.

Мы завершили реализацию класса `FuzzyBool` «с нуля». Нам пришлось переопределить 15 методов (17, если бы мы выполнили минимум по всем четырем операторам сравнения) и реализовать два статических метода. В следующем подподразделе мы покажем альтернативную реализацию, на этот раз унаследовав класс `float`. В этом случае нам придется переопределить всего восемь методов и реализовать две функции модуля, а также «исключить реализацию» 32 методов.

В большинстве объектно-ориентированных языков программирования наследование используется, чтобы создать новый класс, обладающий всеми методами и атрибутами родительского класса, а также дополнительными методами и атрибутами. Язык Python целиком и полностью поддерживает эту парадигму, позволяя добавлять новые методы или переопределять унаследованные методы, чтобы изменить их поведение. Но, помимо этого, язык Python позволяет исключать реализации методов, то есть определять новый класс так, как если бы он вообще не имел некоторых унаследованных методов. Такой прием может вызвать протесты со стороны пуристов объектно-ориентированного программирования, так как он искажает идею полиморфизма, но

в языке Python, по крайней мере иногда, этот прием может быть полезен.

Создание типов данных из других типов данных

Реализация класса `FuzzyBool`, которая обсуждается в этом подподразделе, находится в файле *FuzzyBoolAlt.py*. Одно из основных отличий от предыдущей версии состоит в том, что статические методы `conjunction()` и `disjunction()` в этой версии реализованы как функции модуля. Например:

```
def conjunction(*fuzzies):
    return FuzzyBool(min(fuzzies))
```

На этот раз программный код получился намного проще, чем прежде, потому что класс `FuzzyBoolAlt.FuzzyBool` наследует класс `float`, и потому объекты класса `FuzzyBool` могут использоваться непосредственно, без необходимости выполнять какие-либо преобразования. (Дерево наследования приводится на рис. 6.5.) Порядок обращения к функции теперь также выглядит более понятным, чем прежде. Вместо того чтобы указывать имя модуля и имя класса (или использовать экземпляр класса), после выполнения инструкции `import FuzzyBoolAlt` мы можем производить вызовы как `FuzzyBoolAlt.conjunction()`.

Ниже приводится инструкция `class`, объявляющая класс `FuzzyBool`, и реализация метода `__new__()`:

```
class FuzzyBool(float):

    def __new__(cls, value=0.0):
        return super().__new__(cls,
                value if 0.0 <= value <= 1.0 else 0.0)
```

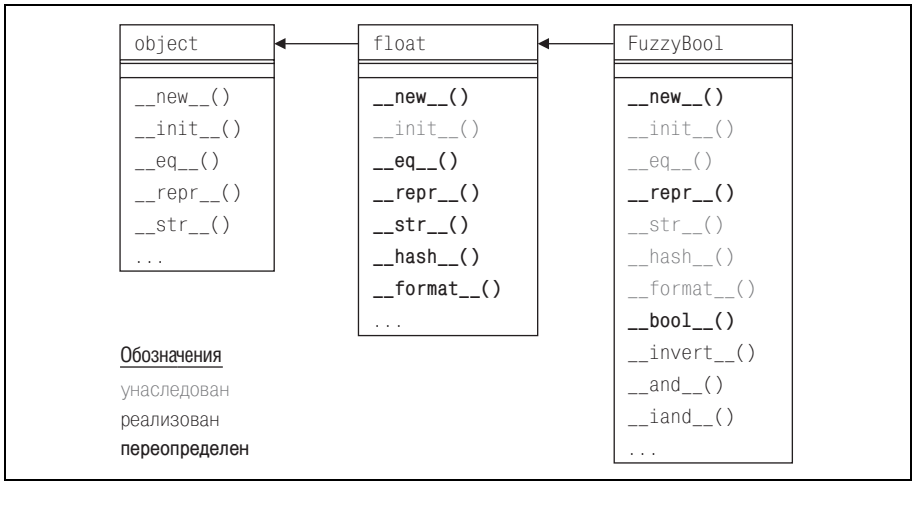

Рис. 6.5. *Дерево наследования альтернативного класса FuzzyBool*

При создании нового класса изменяемых объектов мы обычно полагаемся на метод `object.__new__()`, который создает неинициализированную заготовку объекта. Но в случае создания классов неизменяемых объектов нам необходимо выполнить создание и инициализацию за один шаг, потому что неизменяемый объект не может изменяться после того, как будет создан.

Метод `__new__()` вызывается еще до того, как объект будет создан (так, именно созданием объекта и занимается метод `__new__()`), поэтому метод не получает объект в виде аргумента `self`, в конце концов, объект еще не существует. В действительности метод `__new__()` – это метод класса, он похож на обычный метод класса за исключением того, что он вызывается в контексте класса, а не в контексте экземпляра, и в первом аргументе ему передается класс. Имя аргумента `cls`, в котором передается класс, определяется только соглашениями, так же как и имя `self` используется для обозначения самого объекта исключительно в соответствии с общепринятыми соглашениями.

Итак, когда мы записываем инструкцию `f = FuzzyBool(0.7)`, за кулисами интерпретатор вызывает метод `FuzzyBool.__new__(FuzzyBool, 0.7)`, чтобы создать новый объект – например, *fuzzy*, а затем метод *fuzzy*`.__init__()`, чтобы выполнить его инициализацию, и в результате возвращает ссылку на объект *fuzzy* – ту самую ссылку, которая будет записана в переменную `f`. Большая часть работы метода `__new__()` выполняется реализацией базового класса `object.__new__()`, а все, что делаем мы, – это гарантируем попадание значения в заданный диапазон.

Методы класса определяются с помощью встроенной функции `classmethod()`, используемой как декоратор. Но нам совсем необязательно брать на себя труд писать `@classmethod` перед инструкцией `def __new__()`, потому что интерпретатор уже знает, что этот метод всегда является методом класса. Декоратор необходимо использовать, только когда создаются другие методы класса, как будет показано в последнем разделе главы.

Теперь, когда мы познакомились с методами класса, можно познакомиться с другими разновидностями методов, существующими в языке Python. Методы класса получают в первом аргументе, который передается интерпретатором Python автоматически, свой класс. Обычные методы получают в первом аргументе, который передается интерпретатором Python автоматически, экземпляр класса, относительно которого был вызван метод. Статические методы не имеют первого аргумента, добавляемого автоматически. Все перечисленные разновидности методов могут получать любое число аргументов (в виде второго и последующих аргументов – в случае методов класса и обычных методов, и в виде первого и последующих аргументов – в случае статических методов).

```
def __invert__ (self):
    return FuzzyBool(1.0 - float(self))
```

Этот метод реализует поддержку битового оператора НЕ (~), как и прежде. Примечательно, что теперь вместо обращения к частному атрибуту, в котором хранилось значение FuzzyBool, мы используем сам объект self. Это стало возможным благодаря наследованию класса float, а это означает, что экземпляр FuzzyBool может использоваться везде, где ожидается объект типа float, естественно, за исключением методов, реализация которых была «исключена» из класса FuzzyBool.

```
def __and__(self, other):
    return FuzzyBool(min(self, other))

def __iand__(self, other):
    return FuzzyBool(min(self, other))
```

Реализация логических операций также не изменилась (хотя программный код претерпел некоторые изменения). Здесь так же, как и в методе __invert__(), можно непосредственно использовать объекты self и other, как если бы они были объектами типа float. Мы опустили методы реализации операции ИЛИ, так как они отличаются только именами (__or__() и __ior__()) и тем, что вместо функции min() используют функцию max().

```
def __repr__(self):
    return ("{0}({1})".format(self.__class__.__name__,
                              super().__repr__()))
```

Нам потребовалось переопределить реализацию метода __repr__(), потому что метод float.__repr__() просто возвращает число в виде строки, тогда как нам необходимо, чтобы было указано имя класса, чтобы репрезентативную форму представления можно было использовать в вызове функции eval(). Мы не можем просто передать методу str.format() объект self во втором аргументе, так как это приведет к бесконечной рекурсии метода __repr__(), поэтому мы вызываем реализацию базового класса.

Нам не требуется переопределять метод __str__(), потому что вполне достаточно версии базового класса float.__str__(), которая и будет использоваться в отсутствие метода FuzzyBool.__str__().

```
def __bool__(self):
    return self > 0.5

def __int__(self):
    return round(self)
```

Когда объект типа float используется в логическом контексте, он будет рассматриваться как False, когда его значение равно 0.0, и True – в противном случае. Это не соответствует поведению объектов класса FuzzyBool, поэтому мы переопределили этот метод. Точно так же функция int(self) будет просто отбрасывать дробную часть, превращая в 0 любое значение, кроме 1.0, поэтому здесь мы используем функцию

round(), чтобы округлять до 0 любое значение, меньшее или равное 0.5, и до 1 – значения, большие 0.5.

Мы не стали переопределять методы __hash__(), __format__() и те методы, которые обеспечивают поддержку операторов сравнения, поскольку реализации этих методов в классе float корректно работают применительно к классу FuzzyBool.

Методы, которые мы переопределили, обеспечивают полную реализацию класса FuzzyBool, и для этого потребовалось меньше программного кода, чем было представлено в предыдущем подподразделе. Однако этот новый класс FuzzyBool наследует более 30 методов, которые для него не имеют смысла. Например, ни один из арифметических операторов или операторов сдвига (+, -, *, /, <<, >> и т. д.) неприменим к объектам класса FuzzyBool. Ниже показано, как «исключается» реализация операции сложения:

```
def __add__(self, other):
    raise NotImplementedError()
```

Мы могли бы написать точно такой же программный код для методов __iadd__() и __radd__(), чтобы полностью исключить возможность сложения. (Обратите внимание, что NotImplementedError – это стандартное исключение и оно полностью отличается от объекта NotImplemented.) Чтобы более точно имитировать поведение встроенных классов языка Python, вместо исключения NotImplementedError можно возбуждать исключение TypeError. Ниже показано, как можно было бы заставить метод FuzzyBool.__add__() вести себя так же, как встроенные классы, которые используются в недопустимой операции:

```
def __add__(self, other):
    raise TypeError("unsupported operand type(s) for +: "
                    "'{0}' and '{1}'".format(
                self.__class__.__name__, other.__class__.__name__))
```

Реализация одноместных операций, которые требуется исключить, так чтобы они имитировали поведение встроенных типов, выглядит немного проще:

```
def __neg__(self):
    raise TypeError("bad operand type for unary -: '{0}'".format(
                    self.__class__.__name__))
```

При реализации операторов сравнения используется намного более простой прием. Например, исключить поддержку оператора == мы могли бы так:

```
def __eq__(self, other):
    return NotImplemented
```

Если метод, реализующий оператор сравнения (<, <=, ==, !=, >=, >), возвращает встроенный объект NotImplemented, то при попытке использовать этот метод интерпретатор сначала попытается выполнить сравни-

вание, поменяв операнды местами (на случай, если у объекта other имеется подходящий метод сравнения), а затем, если этот прием не поможет, возбудит исключение TypeError с сообщением, поясняющим, что данная операция не поддерживается для операндов этих типов. Но во всех остальных методах, не имеющих отношения к сравнению и которые требуется исключить, мы должны возбуждать либо исключение NotImplementedError, либо исключение TypeError, как это было сделано в методах __add__() и __neg__() выше.

Однако было бы слишком утомительно исключать каждый нежелательный метод, как было показано выше, хотя такой подход работает и в программном коде выглядит понятнее. Теперь мы рассмотрим более совершенную методику исключения методов – она используется в модуле FuzzyBoolAlt, но для вас, пожалуй, было бы лучше перейти к следующему разделу (стр. 306) и вернуться сюда, если потребуется взглянуть на практический пример.

Ниже приводится программный код, выполняющий исключение двух ненужных нам одноместных операций:

```
for name, operator in (("__neg__", "-"),
                       ("__index__", "index()")):
    message = ("bad operand type for unary {0}: '{{self}}'"
               .format(operator))
    exec("def {0}(self): raise TypeError(\"{1}\".format("
         "self=self.__class__.__name__))".format(name, message))
```

Динамическое программирование, стр. 406

Встроенная функция exec() динамически выполняет программный код из передаваемого ей объекта. В данном случае – это строка, но это могут быть объекты и некоторых других типов. По умолчанию программный код выполняется в объемлющем контексте, в данном случае – внутри определения класса FuzzyBool, поэтому выполняемые инструкции def будут создавать методы класса FuzzyBool, что нам и требуется. Программный код будет выполняться всего один раз, во время импортирования модуля FuzzyBoolAlt. Ниже приводится программный код, который будет сгенерирован первым кортежем ("__neg__", "-"):

```
def __neg__(self):
    raise TypeError("bad operand type for unary -: '{self}'"
                    .format(self=self.__class__.__name__))
```

Здесь мы возбуждаем исключение с текстом сообщения, соответствующим тому, которое используется интерпретатором Python для своих собственных типов. Программный код, обрабатывающий двухместные методы и *n*-местные функции (такие как pow()), следует тому же шаблону, но с другим сообщением об ошибке. Для полноты картины ниже приводится программный код, который мы использовали:

```
for name, operator in (("__xor__", "^"), ("__ixor__", "^="),
        ("__add__", "+"), ("__iadd__", "+="), ("__radd__", "+"),
        ("__sub__", "-"), ("__isub__", "-="), ("__rsub__", "-"),
        ("__mul__", "*"), ("__imul__", "*="), ("__rmul__", "*"),
        ("__pow__", "**"), ("__ipow__", "**="),
        ("__rpow__", "**"), ("__floordiv__", "//"),
        ("__ifloordiv__", "//="), ("__rfloordiv__", "//"),
        ("__truediv__", "/"), ("__itruediv__", "/="),
        ("__rtruediv__", "/"), ("__divmod__", "divmod()"),
        ("__rdivmod__", "divmod()"), ("__mod__", "%"),
        ("__imod__", "%="), ("__rmod__", "%"),
        ("__lshift__", "<<"), ("__ilshift__", "<<="),
        ("__rlshift__", "<<"), ("__rshift__", ">>"),
        ("__irshift__", ">>="), ("__rrshift__", ">>")):
    message = ("unsupported operand type(s) for {0}: "
               "'{{self}}'{{join}} {{args}}".format(operator))
    exec("def {0}(self, *args):\n"
         "    types = [\"'\" + arg.__class__.__name__ + \"'\" "
         "for arg in args]\n"
         "    raise TypeError(\"{1}\".format("
         "self=self.__class__.__name__, "
         "join=(\" and\" if len(args) == 1 else \",\"),"
         "args=\", \".join(types)))".format(name, message))
```

Программный код получился немного более сложным, чем прежде, потому что для двухместных операторов мы должны выводить сообщения, где перечислены оба типа, как *type1 and type2*, а в случае трех и более типов, мы должны выводить их как *type1, type2, type3*, чтобы имитировать поведение встроенных типов. Ниже приводится программный код, сгенерированный для первого кортежа (`"__xor__", "^"`):

```
def __xor__(self, *args):
    types = ["'" + arg.__class__.__name__ + "'" for arg in args]
    raise TypeError("unsupported operand type(s) for ^: "
                    "'{self}'{join} {args}".format(
                    self=self.__class__.__name__,
                    join=(" and" if len(args) == 1 else ","),
                    args=", ".join(types)))
```

Два цикла `for ... in`, которые мы использовали здесь, можно просто копировать и затем добавлять или удалять одноместные операторы и методы в первом цикле и двухместные и *n*-местные операторы и методы – во втором, чтобы исключать реализации ненужных методов.

Теперь, благодаря этому последнему фрагменту программного кода, если бы у нас имелись два объекта `FuzzyBool`, `f` и `g`, и мы попытались сложить их, используя выражение `f + g`, то получили бы исключение `TypeError` с сообщением «unsupported operand type(s) for +: 'FuzzyBool' and 'FuzzyBool'» (неподдерживаемые типы операндов для оператора +: 'FuzzyBool' и 'FuzzyBool'), то есть именно то, что нам и требуется.

Способ создания классов, который мы использовали первым при реализации класса `FuzzyBool`, распространен намного шире и пригоден для создания практически любых типов. Однако если требуется создать класс неизменяемых объектов, основной способ такой реализации заключается в переопределении метода `object.__new__()`, наследовании одного из неизменяемых типов языка Python, таких как `float`, `int`, `str` или `tuple`, с последующей реализацией всех необходимых методов. Недостаток такого подхода заключается в том, что может потребоваться исключить реализации некоторых методов, а это приведет к нарушению полиморфизма; поэтому в большинстве случаев вариант на основе агрегирования, использованного в первой реализации класса `FuzzyBool`, является намного более предпочтительным.

Собственные классы коллекций

В подразделах этого раздела мы рассмотрим порядок создания классов, способных хранить большие объемы данных. Первый класс, который будет рассмотрен, – это класс `Image`, один из тех, что способны хранить изображения. Этот класс является типичным представителем многих классов, используемых для хранения данных, в том смысле, что он не только обеспечивает доступ к данным в памяти, но также имеет методы сохранения данных на диск и загрузки данных с диска. Второй и третий классы, которые мы изучим, `SortedList` и `SortedDict`, предназначены, чтобы восполнить редкий, вызывающий удивление недостаток – отсутствие изначально отсортированных коллекций в стандартной библиотеке Python.

Создание классов, включающих коллекции

Простейшим представлением 2-мерных цветных изображений является двухмерный массив, каждый элемент которого хранит значение цвета. То есть чтобы представить изображение размером 100×100, нам придется хранить 10 000 значений цвета. При создании класса `Image` (в файле *Image.py*) мы выбрали потенциально более эффективный подход. Класс `Image` хранит одно значение цвета для фона плюс те пиксели изображения, цвет которых отличается от цвета фона. Реализовано это с помощью словаря, своего рода разреженного массива, каждый ключ которого представляет координаты (x, y), а значение определяет цвет в точке с этими координатами. Если представить изображение размером 100×100, в котором половина пикселей имеют цвет фона, то нам потребуется хранить всего 5 000 + 1 значение цвета, что дает существенную экономию занимаемой памяти.

Модуль pickle, стр. 341

Модуль *Image.py* следует уже знакомому нам шаблону: он начинается со строки «shebang», далее следует информация об авторских правах в виде комментариев, затем – строка документирования модуля, и затем – инст-

рукции импорта, в данном случае импортируются модули os и pickle. Мы коротко опишем модуль pickle, когда будем обсуждать вопросы сохранения и загрузки изображений. Вслед за инструкциями импорта следуют объявления наших собственных исключений:

```
class ImageError(Exception): pass
class CoordinateError(ImageError): pass
```

Мы показали только первые два класса исключений, остальные (LoadError, SaveError, ExportError и NoFilenameError) создаются точно так же и наследуют исключение ImageError. Пользователи класса Image могут выбирать между обработкой конкретных исключений и обработкой базового исключения ImageError.

Остальную часть модуля занимает определение класса Image, и в самом конце находятся три стандартные строки, запускающие выполнение доктестов модуля. Прежде чем перейти к знакомству с классом и его методами, посмотрим, как его можно использовать:

```
border_color = "#FF0000"    # красный
square_color = "#0000FF"    # синий
width, height = 240, 60
midx, midy = width // 2, height // 2
image = Image.Image(width, height, "square_eye.img")
for x in range(width):
    for y in range(height):
        if x < 5 or x >= width - 5 or y < 5 or y >= height - 5:
            image[x, y] = border_color
        elif midx - 20 < x < midx + 20 and midy - 20 < y < midy + 20:
            image[x, y] = square_color
image.save()
image.export("square_eye.xpm")
```

Обратите внимание, что при установке значений цвета в изображении допускается использовать оператор доступа к элементам ([]). Квадратные скобки могут также использоваться для получения и удаления (фактически для установки в цвет фона) значений цвета в точках с определенными координатами (x, y). Координаты передаются в виде единого кортежа (благодаря оператору запятой), как если бы мы записали обращение в виде: image[(x, y)]. В языке Python легко можно добиться такого вида интеграции с синтаксисом – достаточно лишь реализовать соответствующие специальные методы, которыми в данном случае являются методы реализации оператора доступа к элементу: __getitem__(), __setitem__() и __delitem__().

Для представления цветов в классе Image используются строки шестнадцатеричных цифр. Цвет фона должен определяться при создании изображения, в противном случае по умолчанию используется белый цвет. Класс Image может сохранять на диск и загружать с диска изображения в своем собственном формате, но он также может экспортировать изображения в формат *.xpm*, с которым способны работать многие

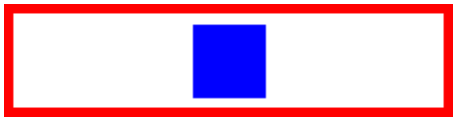

***Рис. 6.6.** Изображение square_eye.xpm*

приложения, предназначенные для работы с графикой. Изображение *.xpm*, которое воспроизводит данный фрагмент программного кода, приводится на рис. 6.6.

Теперь приступим к знакомству с методами класса `Image`, начав с инструкции `class` и метода инициализации:

```
class Image:

    def __init__(self, width, height, filename="",
                 background="#FFFFFF"):
        self.filename = filename
        self.__background = background
        self.__data = {}
        self.__width = width
        self.__height = height
        self.__colors = {self.__background}
```

При создании экземпляра класса `Image` пользователь (то есть программист – пользователь класса) должен указать ширину и высоту изображения, а имя файла и цвет фона являются необязательными, потому что эти параметры имеют значения по умолчанию. Ключами словаря `self.__data` являются координаты (x, y), а его значениями – строки, обозначающие цвет. Множество `self.__colors` инициализируется значением цвета фона – в нем будут храниться все уникальные значения цвета, присутствующие в изображении.

Все атрибуты класса, за исключением `filename`, являются частными, поэтому нам необходимо реализовать средства доступа к ним. Это легко можно сделать с помощью свойств.[1]

```
@property
def background(self):
    return self.__background

@property
def width(self):
    return self.__width

@property
```

[1] В главе 8 будет представлен совершенно иной подход к обеспечению доступа к атрибутам – с использованием таких методов, как `__getattr__()` и `__setattr__()`, что в некоторых случаях может быть очень удобно.

```
    def height(self):
        return self.__height

    @property
    def colors(self):
        return set(self.__colors)
```

Копирование коллекций, стр. 173

При возвращении атрибута объектом нам необходимо понимать разницу между изменяемыми и неизменяемыми типами. Всегда безопасно возвращать атрибуты неизменяемых типов, поскольку их невозможно изменить, но в случае с изменяемыми атрибутами нам следует рассмотреть некоторые их особенности. Возврат ссылки на изменяемый атрибут выполняется очень быстро, потому что при этом не происходит его копирование, но это также означает, что вызывающий программный код получает доступ к внутреннему состоянию объекта и может изменить его так, что объект окажется испорчен. Для предотвращения такой возможности можно взять за правило всегда возвращать копии атрибутов изменяемых типов данных, если это не оказывает существенного влияния на производительность. (В этом случае вместо хранения множества уникальных цветов можно было бы возвращать результат выражения `set(self.__data.values()) | {self.__background}`, когда вызывающей программе потребуется множество уникальных цветов.)

```
    def __getitem__(self, coordinate):
        assert len(coordinate) == 2, "coordinate should be a 2-tuple"
        if (not (0 <= coordinate[0] < self.width) or
            not (0 <= coordinate[1] < self.height)):
            raise CoordinateError(str(coordinate))
        return self.__data.get(tuple(coordinate), self.__background)
```

Этот метод возвращает цвет пикселя с заданными координатами, когда вызывающая программа использует оператор доступа к элементам (`[]`). Специальный метод реализации оператора доступа к элементам и некоторые другие специальные методы, имеющие отношение к коллекциям, перечислены в табл. 6.4.

Мы будем применять две тактики для организации доступа к элементам. Первая тактика состоит в том, чтобы предварительно проверить, является ли аргумент `coordinate` последовательностью из двух элементов (обычно кортеж из двух элементов), для чего используется инструкция `assert`. Вторая тактика состоит в том, что принимаются любые координаты, но если они выходят за пределы изображения, возбуждается наше собственное исключение.

Для получения значения цвета из указанных координат мы использовали метод `dict.get()` со значением по умолчанию, равным цвету фона. Это гарантирует, что если для данной пары координат цвет никогда не

устанавливался, вместо возбуждения исключения KeyError будет возвращен цвет фона.

```
def __setitem__(self, coordinate, color):
    assert len(coordinate) == 2, "coordinate should be a 2-tuple"
    if (not (0 <= coordinate[0] < self.width) or
        not (0 <= coordinate[1] < self.height)):
        raise CoordinateError(str(coordinate))
    if color == self.__background:
        self.__data.pop(tuple(coordinate), None)
    else:
        self.__data[tuple(coordinate)] = color
        self.__colors.add(color)
```

Если пользователь устанавливает значение цвета, равное значению цвета фона, мы просто удаляем соответствующий элемент словаря, поскольку отсутствие тех или иных координат в словаре означает, что пиксели с этими координатами имеют цвет фона. Вместо инструкции del следует использовать метод dict.pop() и передавать ему фиктивный второй аргумент, чтобы избежать возбуждения исключения KeyError, если указанный ключ (координаты) отсутствует в словаре.

Таблица 6.4. Специальные методы коллекций

Специальный метод	Пример использования	Описание
`__contains__(self, x)`	`x in y`	Возвращает `True`, если `x` присутствует в последовательности `y` или `x` является ключом отображения `y`
`__delitem__(self, k)`	`del y[k]`	Удаляет `k`-й элемент последовательности `y` или элемент с ключом `k` в отображении `y`
`__getitem__(self, k)`	`y[k]`	Возвращает `k`-й элемент последовательности `y` или значение элемента с ключом `k` в отображении `y`
`__iter__(self)`	`for x in y:` `  pass`	Возвращает итератор по элементам последовательности `y` или по ключам отображения `y`
`__len__(self)`	`len(y)`	Возвращает число элементов в `y`
`__reversed__(self)`	`reversed(y)`	Возвращает итератор, выполняющий обход элементов последовательности `y` или ключей отображения `y` в обратном порядке
`__setitem__(self, k, v)`	`y[k] = v`	Устанавливает `k`-й элемент последовательности `y` или значение элемента с ключом `k` в отображении `y`

Если цвет отличается от цвета фона, мы устанавливаем его как значение элемента с заданными координатами и добавляем его в множество уникальных цветов, используемых в изображении.

```
def __delitem__(self, coordinate):
    assert len(coordinate) == 2, "coordinate should be a 2-tuple"
    if (not (0 <= coordinate[0] < self.width) or
        not (0 <= coordinate[1] < self.height)):
        raise CoordinateError(str(coordinate))
    self.__data.pop(tuple(coordinate), None)
```

Когда удаляется значение цвета для заданных координат, фактически происходит назначение цвета фона для этих координат. Здесь для удаления элемента также используется метод `dict.pop()`, потому что он корректно работает, даже когда в словаре отсутствует элемент с указанными координатами.

Мы не предусматриваем реализацию метода `__len__()`, поскольку он не имеет смысла для двухмерного объекта. Кроме того, мы не предусматриваем реализацию метода получения репрезентативной формы, поскольку объект `Image` невозможно сформировать полностью единственным вызовом `Image()`; в связи с этим в классе отсутствует реализация метода `__repr__()` (и `__str__()`). Если пользователь вызовет функцию `repr()` или `str()` для объекта `Image`, метод `object.__repr__()` базового класса вернет строку вида `'<Image.Image object at 0x9c794ac>'`. Это стандартный формат представления объектов, которые не могут быть созданы функцией `eval()`. Шестнадцатеричное число – это числовой идентификатор объекта, который является уникальным (обычно это адрес объекта в памяти), но не фиксированным значением.

Нам необходимо предоставить пользователям класса `Image` возможность сохранять изображения на диск и загружать их с диска, поэтому мы предусмотрели реализацию двух методов – `save()` и `load()`, выполняющие эти действия.

Чтобы сохранить данные на диск, мы будем выполнять их *консервирование*. На языке Python под консервированием понимается способ сериализации (преобразования в последовательность байтов или в строку) объектов. В консервировании особенно ценно то, что имеется возможность консервировать коллекции, такие как списки или словари, и даже объекты, внутри которых имеются другие объекты (включая коллекции, которые в свою очередь могут включать другие коллекции, и т. д.) – весь объем данных будет законсервирован, причем без дублирования повторяющихся объектов.

Законсервированный объект можно прочитать прямо в переменную Python – нам не потребуется выполнять тот или иной вид парсинга. То есть консервирование объектов идеально подходит для сохранения и загрузки специализированных коллекций данных, особенно в маленьких программах и в программах, разрабатываемых для личного пользования. Однако

при консервировании не используются механизмы обеспечения безопасности (данные не шифруются и электронная подпись не добавляется), поэтому загрузка законсервированных объектов из непроверенных источников может быть опасным делом. Ввиду этого в программах, предназначенных не только для личного пользования, лучше выработать собственный формат файлов, – специфичный для программы. В главе 7 будет показано, как читать и писать файлы с двоичными данными, с текстом и с данными в формате XML.

```
def save(self, filename=None):
    if filename is not None:
        self.filename = filename
    if not self.filename:
        raise NoFilenameError()

    fh = None
    try:
        data = [self.width, self.height, self.__background,
                self.__data]
        fh = open(self.filename, "wb")
        pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)
    except (EnvironmentError, pickle.PicklingError) as err:
        raise SaveError(str(err))
    finally:
        if fh is not None:
            fh.close()
```

Первая часть функции просто проверяет наличие имени файла. Если объект `Image` был создан без указания имени файла и после этого имя файла не было установлено, то при вызове метода `save()` необходимо явно указывать имя файла (в этом случае он выполняет операцию «сохранить как» и устанавливает значение атрибута `filename`). Если имя файла не было указано при вызове метода, то используется текущее значение атрибута `filename`, а если текущее имя файла не задано, то возбуждается исключение.

Затем создается список, в который добавляются данные объекта для сохранения, включая словарь `self.__data` с элементами координаты-цвет, но исключая множество уникальных цветов, поскольку эти данные легко реконструируются. Далее открывается файл для записи в двоичном режиме и вызывается функция `pickle.dump()`, которая записывает данные объекта в файл. Вот и все!

Модуль `pickle` может сериализовать данные с использованием разных форматов (в документации они называются *протоколами*). Формат определяется третьим аргументом функции `pickle.dump()`. Протокол 0 – это ASCII, его удобно использовать во время отладки. Мы использовали протокол 3 (`pickle.HIGHEST_PROTOCOL`) – компактный двоичный формат, именно поэтому файл был открыт в режиме записи двоичных данных. При чтении файлов с законсервированными объектами протокол

не указывается — функция pickle.load() автоматически определяет используемый протокол.

```
def load(self, filename=None):
    if filename is not None:
        self.filename = filename
    if not self.filename:
        raise NoFilenameError()

    fh = None
    try:
        fh = open(self.filename, "rb")
        data = pickle.load(fh)
        (self.__width, self.__height, self.__background,
         self.__data) = data
        self.__colors = (set(self.__data.values()) |
                         {self.__background})
    except (EnvironmentError, pickle.UnpicklingError) as err:
        raise LoadError(str(err))
    finally:
        if fh is not None:
            fh.close()
```

Эта функция, так же как и функция save(), начинается с определения имени файла, который требуется загрузить. Файл должен быть открыт для чтения в двоичном режиме, а сама операция чтения выполняется единственной инструкцией data = pickle.load(fh). Объект data — это точная реконструкция сохранявшегося объекта. То есть в данном случае это список, содержащий целочисленные значения ширины и высоты, строку с цветом фона и словарь с элементами координаты-цвет. Для присваивания каждого элемента списка data соответствующей переменной выполняется операция распаковывания кортежа, поэтому любые имевшиеся данные, хранившиеся в объекте Image, будут (корректно) потеряны.

Множество уникальных цветов реконструируется посредством создания множества всех цветов, хранящихся в словаре, после чего в множество добавляется цвет фона.

```
def export(self, filename):
    if filename.lower().endswith(".xpm"):
        self.__export_xpm(filename)
    else:
        raise ExportError("unsupported export format: " +
                          os.path.splitext(filename)[1])
```

Мы реализовали один универсальный метод экспорта, где по расширению файла определяется имя вызываемого метода или возбуждается исключение, если запрошенный формат экспорта не поддерживается. В данном случае поддерживается экспорт только в файлы *.xpm* (и только для изображений, насчитывающих не более 8 930 цветов). Мы не приводим реализацию метода __export_xpm(), потому что он никак не

связан с темой главы, но в исходных текстах примеров к этой книге он, конечно же, присутствует.

На этом мы завершаем рассмотрение класса `Image`. Этот класс является типичным представителем типов данных, используемых для хранения специфических данных в программах, обеспечивая возможность доступа к элементам содержащихся в нем данных, возможность сохранения на диск и загрузки с диска своих данных и предоставляя только необходимые методы. В следующих двух подразделах мы увидим, как создать два типа коллекций, обладающие полным API.

Создание классов коллекций посредством агрегирования

В этом подразделе мы создадим полный тип коллекции `SortedList`, который представляет собой список, хранящий свои элементы в порядке сортировки. Сортировка выполняется с помощью оператора «меньше чем» (`<`), действие которого реализуется специальным методом `__lt__()`, или с помощью функции, которая передается в виде аргумента `key`. Класс стремится соответствовать API встроенного класса `list`, чтобы максимально упростить его освоение, но некоторые методы не могут быть реализованы по вполне объяснимым причинам, например, использование оператора конкатенации (`+`) может привести к нарушению порядка сортировки элементов, поэтому мы не реализуем его.

Как всегда, когда создается свой собственный класс, у нас есть выбор – наследовать класс, подобный тому, что создается, или создать новый класс с нуля, агрегируя в него экземпляры других, необходимых классов. Для класса `SortedList`, рассматриваемого в этом подразделе, мы будем использовать подход на основе агрегирования элементов данных, а в следующем подразделе, где рассматривается создание класса `SortedDict`, мы будем использовать не только агрегирование, но и наследование.

В главе 8 мы узнаем, что классы могут брать на себя обязательства поддерживать определенные типы API. Например, класс `list` поддерживает API `MutableSequence`, то есть поддерживает оператор `in`, встроенные функции `iter()` и `len()`, оператор доступа к элементам (`[]`) для получения, установки и удаления элементов, а также метод `insert()`. Класс `SortedList`, представленный здесь, не поддерживает возможность изменения значений элементов и не имеет метода `insert()`, поэтому он не поддерживает API `MutableSequence`. Если бы мы создавали класс `SortedList`, наследуя класс `list`, окончательная реализация заявляла бы о себе как об изменяемой последовательности, но не имела бы полного API. Поэтому класс `SortedList` не наследует класс `list` и не делает никаких утверждений о своем API. С другой стороны, класс `SortedDict`, рассматриваемый в следующем подразделе, реализует полный API `MutableMapping`, который поддерживается классом `dict`, поэтому мы смогли создать его как подкласс класса `dict`.

Ниже приводятся несколько типичных примеров использования класса SortedList:

```
letters = SortedList.SortedList(("H", "c", "B", "G", "e"), str.lower)
# str(letters) == "['B', 'c', 'e', 'G', 'H']"
letters.add("G")
letters.add("f")
letters.add("A")
# str(letters) == "['A', 'B', 'c', 'e', 'f', 'G', 'G', 'H']"
letters[2] # вернет: 'c'
```

Объект класса SortedList представляет собой *агрегат* (композицию) из двух частных атрибутов — функции self.__key() (ссылка на объект self.__key) и списка self.__list.

Лямбда-функции, стр. 215

Ключевая функция передается во втором аргументе (или в виде именованного аргумента key, если начальная последовательность не задана). Если ключевая функция задана, используется частная функция модуля:

```
_identity = lambda x: x
```

Это функция тождественности: она просто возвращает свой аргумент, не изменяя его, поэтому, когда она используется в качестве ключевой функции, это означает, что ключом сортировки объектов в списке являются сами объекты.

Тип SortedList не поддерживает возможность изменения элементов с помощью оператора доступа к элементам ([]) (и поэтому не реализует специальный метод __setitem__()), а также не имеет методов append() или extend(), поскольку они могут нарушать порядок сортировки. Единственный способ добавить элементы в сортированный список состоит в том, чтобы передать последовательность при создании объекта SortedList, или добавлять их позднее, с помощью метода SortedList.add(). С другой стороны, мы безопасно можем использовать оператор доступа к элементам для получения значения или удаления элемента в указанной позиции, потому что ни одна из этих операций не оказывает влияния на порядок сортировки, поэтому были реализованы оба специальных метода __getitem__() и __delitem__().

Теперь мы перейдем к последовательному рассмотрению методов класса и, как обычно, начнем с инструкции class и метода инициализации:

```
class SortedList:

    def __init__(self, sequence=None, key=None):
        self.__key = key or _identity
        assert hasattr(self.__key, "__call__")
        if sequence is None:
            self.__list = []
        elif (isinstance(sequence, SortedList) and
              sequence.key == self.__key):
```

```
            self.__list = sequence.__list[:]
        else:
            self.__list = sorted(list(sequence), key=self.__key)
```

Поскольку имя функции является ссылкой на объект (то есть на функцию), мы можем хранить функции в переменных как ссылки на любые другие объекты. Здесь частная переменная `self.__key` хранит ссылку на ключевую функцию, передаваемую методу, или функцию идентичности. Первая инструкция метода использует тот факт, что оператор ИЛИ возвращает первый операнд, если в логическом контексте он оценивается как `True` (то есть когда аргумент `key` имеет значение, отличное от `None`), или второй операнд – в противном случае. Немного более длинный, но более очевидный вариант: `self.__key = key if key is not None else _identity`.

Теперь, когда мы определились с ключевой функцией, мы используем инструкцию `assert`, чтобы убедиться, что эту функцию можно вызывать. Встроенная функция `hasattr()` возвращает `True`, если объект, переданный в первом аргументе, имеет атрибут, имя которого передается во втором аргументе. Имеются соответствующие функции `setattr()` и `delattr()`, которые будут описываться в главе 8. Все вызываемые объекты, такие как функции или методы, имеют атрибут `__call__`.

Чтобы порядок создания сделать максимально похожим на создание объектов типа `list`, мы определили необязательный аргумент `sequence`, соответствующий единственному аргументу функции `list()`. Класс `SortedList` включает в себя коллекцию типа `list` в виде частной переменной `self.__list` и сохраняет в этой переменной элементы в отсортированном порядке, используя заданную ключевую функцию.

Предложение `elif` проверяет, является ли заданная последовательность объектом типа `SortedList`, и если это так, то проверяет, не используется ли для этой последовательности та же самая ключевая функция. Если последовательность удовлетворяет обоим критериям, то просто создается поверхностная копия последовательности без ее сортировки. Если большинство ключевых функций создается «на лету», с помощью инструкции `lambda`, при сравнивании они не будут расцениваться как эквивалентные, даже если содержат один и тот же программный код, поэтому такая оптимизация на практике может не давать существенного эффекта.

```
    @property
    def key(self):
        return self.__key
```

После создания сортированного списка его ключевая функция должна быть зафиксирована, поэтому мы сохраняем ее в частной переменной, чтобы предотвратить возможность ее изменения. Но у некоторых пользователей может возникнуть потребность получить ссылку на ключевую функцию (как будет показано в следующем подразделе), по-

этому мы создаем свойство, доступное только для чтения, обеспечивающее такую возможность.

```
def add(self, value):
    index = self.__bisect_left(value)
    if index == len(self.__list):
        self.__list.append(value)
    else:
        self.__list.insert(index, value)
```

Когда вызывается этот метод, он должен вставить указанное значение в частный список `self.__list` в нужную позицию, соблюдая порядок сортировки списка. Частный метод `SortedList.__bisect_left()` возвращает индекс требуемой позиции, как будет показано чуть ниже. Если новое значение больше любого другого значения в списке, то его следует добавить в конец списка и в этом случае индекс позиции будет равен длине списка (номера позиций в списке начинаются с 0 и заканчиваются значением `len(L) - 1`). В этом случае добавление нового элемента должно выполняться методом `append()`. В противном случае производится вставка нового значения в найденную позицию, которая может иметь порядковый номер 0, если новое значение меньше любого другого значения в списке.

```
def __bisect_left(self, value):
    key = self.__key(value)
    left, right = 0, len(self.__list)
    while left < right:
        middle = (left + right) // 2
        if self.__key(self.__list[middle]) < key:
            left = middle + 1
        else:
            right = middle
    return left
```

Этот частный метод вычисляет номер позиции в списке, куда должно быть вставлено новое значение. Он вычисляет ключ для нового значения, используя ключевую функцию, и сравнивает с вычисленными ключами проверяемых элементов. Алгоритм, используемый методом, называется алгоритмом *двоичного поиска* (или поиск методом половинного деления), который обладает высокой скоростью даже в случае очень больших списков. Например, чтобы отыскать местоположение нового значения в списке, насчитывающем 1 000 000 элементов, требуется выполнить не более 21 сравнения.[1] Сравните это с простым, несортированным списком, для которого используется алгоритм линейного поиска, требующий выполнить в среднем 500 000 сравнений,

[1] Модуль `bisect`, входящий в состав стандартной библиотеки языка Python, содержит функцию `bisect.bisect_left()` и ряд других функций, но к моменту написания этих строк ни одна из функций в модуле `bisect` не обеспечивала возможность использования ключевых функций.

а в самом тяжелом случае – до 1 000 000 сравнений, чтобы отыскать местоположение нового значения в списке, насчитывающем 1 000 000 элементов.

```
def remove(self, value):
    index = self.__bisect_left(value)
    if index < len(self.__list) and self.__list[index] == value:
        del self.__list[index]
    else:
        raise ValueError("{0}.remove(x): x not in list".format(
                         self.__class__.__name__))
```

Этот метод используется, чтобы удалить первое вхождение заданного значения. Он использует метод `SortedList.__bisect_left()`, чтобы отыскать позицию заданного значения, после чего проверяет, находится ли найденный индекс внутри списка и действительно ли в найденной позиции находится искомое значение. Если условия выполняются, производится удаление элемента; в противном случае возбуждается исключение `ValueError` (что полностью соответствует поведению метода `list.remove()` в аналогичной ситуации).

```
def remove_every(self, value):
    count = 0
    index = self.__bisect_left(value)
    while (index < len(self.__list) and
           self.__list[index] == value):
        del self.__list[index]
        count += 1
    return count
```

Этот метод похож на метод `SortedList.remove()` и является расширением API списка. Сначала он отыскивает в списке номер позиции первого вхождения заданного значения и затем в цикле, пока значение индекса находится в пределах списка и значение элемента в данной позиции совпадает с указанным значением, производит удаление элементов. В этом программном коде имеется очень тонкий момент – так как в каждой итерации происходит удаление элемента списка, то после удаления элемента в позиции с данным индексом оказывается элемент, следовавший за удаленным.

```
def count(self, value):
    count = 0
    index = self.__bisect_left(value)
    while (index < len(self.__list) and
           self.__list[index] == value):
        index += 1
        count += 1
    return count
```

Этот метод возвращает число вхождений заданного значения в список (которое может быть равно 0). Он использует очень похожий алго-

ритм, что и метод SortedList.remove_every(), только здесь в каждой итерации необходимо увеличивать номер позиции.

```
def index(self, value):
    index = self.__bisect_left(value)
    if index < len(self.__list) and self.__list[index] == value:
        return index
    raise ValueError("{0}.index(x): x not in list".format(
                     self.__class__.__name__))
```

Так как объект типа SortedList представляет собой отсортированный список, мы можем использовать метод половинного деления, чтобы отыскать (или не отыскать) заданное значение в списке.

```
def __delitem__(self, index):
    del self.__list[index]
```

Специальный метод __delitem__() обеспечивает поддержку синтаксиса del *L*[*n*], где *L* – это отсортированный список, а *n* – целое число, определяющее номер позиции в списке. Мы не выполняем проверку на выход индекса за пределы списка, поскольку в этом случае вызов self.__list[index] возбудит исключение IndexError, что нам и требуется.

```
def __getitem__(self, index):
    return self.__list[index]
```

Этот метод обеспечивает поддержку синтаксиса x = *L*[*n*], где *L* – это отсортированный список, а *n* – целое число, определяющее номер позиции в списке.

```
def __setitem__(self, index, value):
    raise TypeError("use add() to insert a value and rely on "
                    "the list to put it in the right place")
```

Нам требуется предотвратить возможность изменения пользователем элемента списка в заданной позиции (то есть запретить возможность выполнения операции *L*[*n*] = x), так как в этом случае может нарушиться порядок сортировки элементов в списке. Как правило, чтобы показать, что операция не поддерживается тем или иным типом данных, используется исключение TypeError.

```
def __iter__(self):
    return iter(self.__list)
```

Этот метод легко реализовать, потому что мы можем просто вернуть итератор частного списка, используя встроенную функцию iter(). Этот метод используется для поддержки синтаксиса for *value* in *iterable*.

Обратите внимание: когда объект интерпретируется как последовательность, то используется именно этот метод. Так, чтобы преобразовать объект *L* типа SortedList в простой список, можно вызвать функцию list(*L*), в результате чего интерпретатор Python вызовет метод SortedList.__iter__(*L*), чтобы получить последовательность, необходимую функции list().

```
def __reversed__(self):
    return reversed(self.__list)
```

Этот метод обеспечивает поддержку встроенной функции `reversed()`, благодаря чему мы можем записать, например, `for` *`value`* `in reversed(`*`iterable`*`)`.

```
def __contains__(self, value):
    index = self.__bisect_left(value)
    return (index < len(self.__list) and
            self.__list[index] == value)
```

Метод `__contains__()` обеспечивает поддержку оператора `in`. И снова мы можем использовать быстрый алгоритм двоичного поиска вместо медленного алгоритма линейного поиска, используемого классом `list`.

```
def clear(self):
    self.__list = []

def pop(self, index=-1):
    return self.__list.pop(index)

def __len__(self):
    return len(self.__list)

def __str__(self):
    return str(self.__list)
```

Метод `SortedList.clear()` отбрасывает существующий список и замещает его новым пустым списком. Метод `SortedList.pop()` удаляет элемент из указанной позиции и возвращает его или возбуждает исключение `IndexError`, если индекс находится за пределами списка. В методах `pop()`, `__len__()` и `__str__()` мы просто перекладываем работу на объект `self.__list`.

Мы не предусматриваем переопределение специального метода `__repr__()`, поэтому, когда для объекта *`L`* типа `SortedList` пользователь вызовет функцию `repr(`*`L`*`)`, будет использоваться метод `object.__repr__()` базового класса. Он воспроизведет строку `'<SortedList.SortedList object at 0x97e7cec>'`, но, конечно, с другим значением числового идентификатора. Мы не можем предоставить иную реализацию метода `__repr__()`, потому что для этого пришлось бы представить в строке ключевую функцию, но у нас нет возможности создать репрезентативное представление ссылки на объект-функцию в виде строки, которую можно было бы передать функции `eval()`.

Мы не предусматриваем реализацию методов `insert()`, `reverse()` или `sort()`, потому что ни один из них не соответствует понятию сортированного списка. Если попытаться вызвать какой-либо из них, будет возбуждено исключение `AttributeError`.

Если мы скопируем сортированный список, используя прием *`L`*`[:]`, в результате будет получен объект типа `list`, а не типа `SortedList`. Простейший способ получить копию сортированного списка состоит

в том, чтобы импортировать модуль copy и воспользоваться функцией copy.copy() – она достаточно интеллектуальна, чтобы скопировать сортированный список (и экземпляры большинства других классов) без какой-либо помощи. Однако мы решили реализовать явный метод copy():

```
def copy(self):
    return SortedList(self, self.__key)
```

Передавая в первом аргументе сам объект self, мы гарантируем, что будет создана лишь поверхностная копия self.__list вместо копирования и пересортировки. (Благодаря тому, что в методе __init__() присутствует предложение elif, выполняющее проверку типа.) Теоретически высокая скорость копирования таким способом недостижима для функции copy.copy(), однако мы легко можем исправить этот недостаток, добавив строку:

```
__copy__ = copy
```

Когда вызывается функция copy.copy(), она сначала пытается использовать специальный метод __copy__() объекта и только в случае его отсутствия выполняет свой собственный программный код. Благодаря этой строке функция copy.copy() теперь сможет при работе с отсортированными списками использовать метод SortedList.copy(). (То же самое возможно в случае реализации специального метода __deepcopy__(), но это немного сложнее – электронная документация модуля copy содержит все необходимые подробности.)

Теперь мы завершили реализацию класса SortedList. В следующем подразделе мы будем использовать объект класса SortedList для хранения ключей класса SortedDict.

Создание классов коллекций посредством наследования

Класс SortedDict, который демонстрируется в этом подразделе, стремится максимально имитировать поведение класса dict. Основное отличие между ними состоит в том, что ключи класса SortedDict всегда упорядочены в соответствии с заданной ключевой функцией или в соответствии с функцией идентичности. Класс SortedDict предоставляет тот же API, что и класс dict (за исключением поддержки функции repr(), обеспечивающей возможность создания репрезентативной формы представления, пригодной для передачи функции eval()), плюс два дополнительных метода, которые имеют смысл только для упорядоченной коллекции.[1]

[1] Класс SortedDict, представленный здесь, отличается от аналогичного класса из написанной этим же автором книги «Rapid GUI Programming with Python and Qt», ISBN 0132354187, и от похожего класса в каталоге пакетов Python (Python Package Index).

Ниже приводятся несколько примеров, дающих представление о том, как работает SortedDict:

```
d = SortedDict.SortedDict(dict(s=1, A=2, y=6), str.lower)
d["z"] = 4
d["T"] = 5
del d["y"]
d["n"] = 3
d["A"] = 17
str(d) # вернет: "{'A': 17, 'n': 3, 's': 1, 'T': 5, 'z': 4}"
```

В реализации класса SortedDict используются оба механизма – агрегирование и наследование. Сортированный список ключей агрегирован в виде переменной экземпляра, тогда как сам класс SortedDict наследует встроенный класс dict. Наше знакомство с классом мы начнем с рассмотрения инструкции class и метода инициализации, а затем поочередно рассмотрим все остальные методы.

```
class SortedDict(dict):

    def __init__ (self, dictionary=None, key=None, **kwargs):
        dictionary = dictionary or {}
        super().__init__(dictionary)
        if kwargs:
            super().update(kwargs)
        self.__keys = SortedList.SortedList(super().keys(), key)
```

В инструкции class указан базовый класс dict. Метод инициализации пытается имитировать функцию dict(), но при этом имеет второй дополнительный аргумент, в котором передается ключевая функция. Вызов super().__init__() использует для инициализации объекта класса SortedDict метод dict.__init__() базового класса. Точно так же, если методу были переданы именованные аргументы, вызывается метод dict.update() базового класса, чтобы добавить их в словарь. (Обратите внимание, что принимается только одно вхождение любого именованного аргумента, поэтому ни один из ключей среди именованных аргументов kwargs не может быть «dictionary» или «key».)

Копии всех ключей словаря сохраняются в сортированном списке – в переменной self.__keys. При инициализации сортированного списка ему передается список ключей словаря с помощью метода базового класса dict.keys() – мы не можем использовать метод SortedDict.keys(), потому что он опирается на использование переменной self.__keys, которая появится только *после* того, как будет создан сортированный список SortedList ключей.

```
def update(self, dictionary=None, **kwargs):
    if dictionary is None:
        pass
    elif isinstance(dictionary, dict):
        super().update(dictionary)
    else:
```

```
            for key, value in dictionary.items():
                super().__setitem__(key, value)
        if kwargs:
            super().update(kwargs)
        self.__keys = SortedList.SortedList(super().keys(),
                                            self.__keys.key)
```

Этот метод используется для добавления в словарь элементов другого словаря, или именованных аргументов, или и того и другого. Элементы, существующие только в другом словаре, добавляются в данный словарь, а если элементы с одинаковыми ключами присутствуют в обоих словарях, значения элементов другого словаря заместят значения элементов данного словаря. Мы несколько расширили поведение словаря, так как сохраняем исходную ключевую функцию словаря, даже если другой словарь является объектом класса `SortedDict`.

Добавление элементов выполняется в два этапа. Сначала выполняется добавление элементов словаря. Если другой словарь является объектом подкласса, наследующего класс `dict` (что, безусловно, относится и к классу `SortedDict`), добавление выполняется вызовом метода `dict.update()` базового класса – здесь очень важно использовать метод базового класса, потому что в случае вызова `SortedDict.update()` он попадет в бесконечную рекурсию. Если словарь не является объектом подкласса, наследующего класс `dict`, то выполняются итерации по его элементам, и каждая пара ключ-значение добавляется отдельно. (Если словарь не является объектом подкласса, наследующего класс `dict`, и не имеет метода `items()`, совершенно справедливо будет возбуждено исключение `AttributeError`.) Если были переданы именованные аргументы, точно так же вызывается метод `update()` базового класса, чтобы добавить их в словарь.

В результате добавления элементов список `self.__keys` становится недействительным, поэтому мы замещаем его новым списком типа `SortedList`, образованным из ключей словаря (опять же используя метод базового класса, потому что метод `SortedDict.keys()` опирается на использование списка `self.__keys`, который находится в процессе обновления), используя оригинальную ключевую функцию сортированного списка.

```
    @classmethod
    def fromkeys(cls, iterable, value=None, key=None):
        return cls({k: value for k in iterable}, key)
```

Интерфейс класса `dict` включает метод класса `dict.fromkeys()`. Этот метод используется для создания нового словаря на основе итерируемого объекта. Каждый элемент итерируемого объекта становится ключом, а значением каждого ключа становится `None` или значение аргумента `value`.

Так как это метод класса, первый его аргумент автоматически передается интерпретатором Python и является классом. Объекту класса `dict`

будет передан класс dict, а объекту класса SortedDict будет передан класс SortedDict. Возвращаемое значение – словарь заданного класса. Например:

```
class MyDict(SortedDict.SortedDict): pass
d = MyDict.fromkeys("VEINS", 3)
str(d) # returns: "{'E': 3, 'I': 3, 'N': 3, 'S': 3, 'V': 3}"
d.__class__.__name__ # returns: 'MyDict'
```

То есть при вызове метода дочернего класса в переменной cls будет установлен корректный класс, точно так же, как при вызове обычных методов в переменной self будет передана ссылка на текущий объект.

Функции-генераторы

Функция-генератор, или *метод-генератор* – это функция, или метод, содержащая выражение yield. В результате обращения к функции-генератору возвращается итератор. Значения из итератора извлекаются по одному, с помощью его метода __next__(). При каждом вызове метода __next__() он возвращает результат вычисления выражения yield. (Если выражение отсутствует, возвращается значение None.) Когда функция-генератор завершается или выполняет инструкцию return, возбуждается исключение StopIteration.

На практике очень редко приходится вызывать метод __next__() или обрабатывать исключение StopIteration. Обычно функция-генератор используется в качестве итерируемого объекта. Ниже приводятся две практически эквивалентные функции. Функция слева возвращает список, а функция справа возвращает генератор.

```
# Создает и возвращает список
def letter_range(a, z):
    result = []
     while ord(a) < ord(z):
        result.append(a)
        a = chr(ord(a) + 1)
     return result
```

```
# Возвращает каждое
# значение по требованию
def letter_range(a, z):
    while ord(a) < ord(z):
        yield a
        a = chr(ord(a) + 1)
```

Результаты, воспроизводимые обеими функциями, можно обойти с помощью цикла for, например for letter in letter_range("m", "v"):. Однако когда требуется получить список символов с помощью функции слева, достаточно просто вызвать ее как letter_range("m", "v"), а для функции справа необходимо выполнить преобразование: list(letter_range("m", "v")).

Функции-генераторы и методы-генераторы (а также выражения-генераторы) более полно рассматриваются в главе 8.

Методы класса отличаются от статических методов и удобнее в использовании, потому что статические методы привязаны к определенному классу и не различают ситуации, когда они вызываются в контексте оригинального класса, а когда – в контексте подкласса.

```
def __setitem__(self, key, value):
    if key not in self:
        self.__keys.add(key)
    return super().__setitem__(key, value)
```

Этот метод обеспечивает поддержку синтаксиса `d[key] = value`. Если ключ `key` отсутствует в словаре, он добавляется в список ключей, – с использованием возможностей класса `SortedList`, чтобы поместить его в позицию с корректным номером. Затем вызывается метод базового класса, результат которого возвращается вызывающей программе, чтобы обеспечить поддержку объединения операций в цепочку, например `x = d[key] = value`.

Обратите внимание, что условная инструкция `if` проверяет наличие ключа в словаре типа `SortedList`, используя выражение `not in self`. Так как класс `SortedDict` наследует класс `dict`, объект класса `SortedDict` может использоваться везде, где ожидается объект класса `dict`, и в данном случае объект `self` является экземпляром класса `SortedDict`. При переопределении методов класса `dict` в классе `SortedDict`, когда для выполнения каких-либо действий необходимо вызвать реализацию базового класса, мы не должны забывать использовать функцию `super()`, как это сделано в последней инструкции данного метода. Делается это для того, чтобы предотвратить вызов методом самого себя и попадание в бесконечную рекурсию.

Мы не стали переопределять метод `__getitem__()`, так как версия метода в базовом классе прекрасно справляется с работой и не оказывает влияния на порядок сортировки ключей.

```
def __delitem__(self, key):
    try:
        self.__keys.remove(key)
    except ValueError:
        raise KeyError(key)
    return super().__delitem__(key)
```

Этот метод обеспечивает поддержку синтаксиса `del d[key]`. Если ключ `key` отсутствует в словаре, вызов метода `SortedDict.remove()` возбуждает исключение `ValueError`. Если это происходит, исключение обрабатывается и возбуждается исключение `KeyError`, что соответствует поведению класса `dict`. В противном случае возвращается результат вызова реализации базового класса, которая удаляет элемент с заданным ключом из словаря.

```
def setdefault(self, key, value=None):
    if key not in self:
```

```
            self.__keys.add(key)
        return super().setdefault(key, value)
```

Этот метод возвращает значение для заданного ключа, если этот ключ присутствует в словаре. В противном случае он создает новый элемент с заданным ключом и значением и возвращает значение. В случае со словарем типа `SortedDict` ключ должен быть добавлен в список ключей, если этого ключа еще нет в словаре.

```
def pop(self, key, *args):
    if key not in self:
        if len(args) == 0:
            raise KeyError(key)
        return args[0]
    self.__keys.remove(key)
    return super().pop(key, args)
```

Если в словаре имеется указанный ключ, этот метод возвращает соответствующее ему значение и удаляет элемент ключ-значение из словаря. Кроме того, ключ также удаляется из списка ключей.

Реализация этого не так проста, потому что метод должен поддерживать два различных типа поведения, чтобы соответствовать реализации метода `dict.pop()`. Во-первых, реализация должна поддерживать синтаксис `d.pop(k)` и возвращать значение ключа `k` или, если ключ не существует, возбуждать исключение `KeyError`. Во-вторых, поддерживать синтаксис `d.pop(k, value)` и возвращать значение ключа `k` или, если ключ не существует, возвращать значение `value` (которое может быть объектом `None`). В любом случае, если ключ существует, соответствующий элемент словаря удаляется.

```
def popitem(self):
    item = super().popitem()
    self.__keys.remove(item[0])
    return item
```

Метод `dict.popitem()` удаляет и возвращает случайный элемент ключ-значение из словаря. Сначала необходимо вызвать метод базового класса, так как заранее неизвестно, какой элемент будет удален. Затем ключ элемента удаляется из списка ключей, и элемент возвращается вызывающей программе.

```
def clear(self):
    super().clear()
    self.__keys.clear()
```

Здесь удаляются все элементы словаря и все элементы списка ключей.

```
def values(self):
    for key in self.__keys:
    yield self[key]

def items(self):
```

```
        for key in self.__keys:
        yield (key, self[key])

    def __iter__(self):
        return iter(self.__keys)

    keys = __iter__
```

У словарей имеется четыре метода, возвращающие итераторы: `dict.values()` — для значений, `dict.items()` — для элементов ключ-значение, `dict.keys()` — для ключей и специальный метод `__iter__()`, обеспечивающий поддержку функции `iter()`, возвращающей итератор по ключам словаря. (В действительности версии методов базового класса возвращают представления словаря, но итераторы, реализованные здесь, во многом обладают тем же поведением.)

Поскольку методы `__iter__()` и `keys()` обладают идентичным поведением, то вместо того чтобы создавать отдельную реализацию метода `keys()`, мы просто создали переменную с именем `keys` и записали в нее ссылку на метод `__iter__()`. Теперь пользователи класса `SortedDict` смогут вызывать метод `d.keys()` или функцию `iter(d)`, чтобы получить итератор, позволяющий выполнить обход ключей словаря. Точно так же они смогут вызывать метод `d.values()`, чтобы получить итератор, позволяющий выполнить обход значений словаря.

Методы `values()` и `items()` являются методами-генераторами (краткое описание методов-генераторов приводится во врезке «Функции-генераторы» на стр. 324). В обоих случаях они выполняют итерации по сортированному списку ключей, благодаря чему всегда возвращают итераторы, выполняющие итерации в порядке сортировки ключей (согласно ключевой функции, которая определяется на этапе создания словаря). В методах `items()` и `values()` значения извлекаются из словаря с использованием синтаксиса `d[k]` (то есть с помощью метода `__getitem__()`), благодаря тому, что у нас сохраняется возможность интерпретировать `self` как объект класса `dict`.

Методы-генераторы, стр. 397

```
    def __repr__(self):
        return object.__repr__(self)

    def __str__(self):
        return ("{" + ", ".join(["{0!r}: {1!r}".format(k, v)
                                 for k, v in self.items()]) + "}")
```

Мы не в состоянии предоставить репрезентативную форму представления объекта класса `SortedDict`, пригодную для передачи функции `eval()`, потому что мы не можем реализовать репрезентативную форму представления ключевой функции. Поэтому мы переопределяем метод `__repr__()` и вместо метода `dict.__repr__()` вызываем метод `object.__repr__()` всеобщего базового класса. Он воспроизводит строку репре-

зентативной формы представления, непригодную для передачи функции `eval()`, например `'<SortedDict.SortedDict object at 0xb71fff5c>'`.

Мы предусмотрели собственную реализацию метода `__str__()`, так как нам необходимо, чтобы элементы выводились в порядке сортировки ключей. Однако этот метод можно было бы реализовать несколько иначе:

```
items = []
for key, value in self.items():
    items.append("{0!r}: {1!r}".format(key, value))
return "{" + ", ".join(items) + "}"
```

Однако использование генератора списков позволило получить более короткую реализацию и избавиться от временной переменной `items`.

Методы `dict.get()` и `dict.__getitem__()` базового класса (для поддержки синтаксиса `v = d[k]`), `dict.__len__()` (для поддержки `len(d)`) и `dict.__contains__()` (для поддержки `x in d`) прекрасно справляются со своей работой и не зависят от порядка сортировки, поэтому нам не требуется переопределять их.

Последний метод класса `dict`, который нам необходимо переопределить, – это метод `copy()`.

```
def copy(self):
    d = SortedDict()
    super(SortedDict, d).update(self)
    d.__keys = self.__keys.copy()
    return d
```

Этот метод можно было бы реализовать просто: `def copy(self): return SortedDict(self)`. Но мы выбрали немного более сложное решение, чтобы избежать повторной сортировки уже отсортированного списка ключей. Здесь создается пустой отсортированный словарь, затем в него с помощью метода `dict.update()` базового класса, с целью избежать вызова переопределенной версии метода `SortedDict.update()`, переписываются элементы оригинального отсортированного словаря, и наконец список ключей вновь созданного словаря замещается поверхностной копией списка ключей `self.__keys` оригинального словаря.

Когда функция `super()` вызывается без аргументов, она работает с объектом `self` и с его базовым классом. Но мы легко можем заставить ее работать с любым классом и с любым объектом, явно передавая ей имя класса и объект. При таком использовании функция `super()` работает с классом, который является *базовым* по отношению к указанному, поэтому в данном случае программный код имеет тот же эффект, как (который вполне можно было бы использовать) `dict.update(d, self)`.

Благодаря тому, что в языке Python используется весьма эффективный алгоритм сортировки, который оптимизирован для работы с частично отсортированными списками, наша реализация едва ли даст заметный эффект, разве только при работе с огромными словарями. Тем

не менее она демонстрирует, что собственная реализация метода `copy()` в принципе может быть эффективнее, чем прием `copy_of_x = ClassOfX(x)`, который используется во встроенных типах языка Python. Точно так же, как и в случае с классом `SortedList`, мы определили ссылку `__copy__ = copy`, чтобы функция `copy.copy()` могла использовать нашу реализацию, а не свою собственную.

```
def value_at(self, index):
    return self[self.__keys[index]]

def set_value_at(self, index, value):
    self[self.__keys[index]] = value
```

Эти два метода расширяют API класса `dict`. Поскольку в отличие от обычного класса `dict` ключи в классе `SortedDict` упорядочены, отсюда следует, что к объектам этого класса применимо понятие номера позиции. Например, первый элемент словаря имеет номер позиции 0, а последний элемент – номер позиции `len(d) - 1`. Оба эти метода оперируют элементом словаря, ключ которого находится в позиции `index` внутри отсортированного списка ключей. Благодаря механизму наследования мы можем отыскивать значения в объекте `SortedDict` с помощью оператора доступа к элементу (`[]`), применяя его непосредственно к объекту `self`, так как в этом отношении `self` можно считать объектом класса `dict`. Если указанное значение индекса находится за пределами списка, методы возбуждают исключение `IndexError`.

На этом мы завершили реализацию класса `SortedDict`. Не так часто возникает необходимость создавать универсальные классы коллекций, подобные этому классу, но когда такая необходимость возникает, специальные методы позволяют полностью интегрировать наш класс в язык Python, чтобы пользователи могли работать с ними, как с любыми другими встроенными классами или с классами из стандартной библиотеки.

В заключение

В этой главе были рассмотрены все фундаментальные основы поддержки объектно-ориентированного программирования в языке Python. Мы начали с демонстрации некоторых недостатков, присущих применению исключительно процедурного стиля программирования, и описали, как их можно избежать, используя объектно-ориентированный подход. Затем были описаны некоторые термины, используемые в объектно-ориентированном программировании, включая множество «совпадающих по значению» терминов, таких как *базовый класс* и *суперкласс*.

Мы увидели, как можно создавать простые классы с атрибутами данных и собственными методами. Кроме того, мы увидели, как наследовать классы и как добавлять дополнительные атрибуты данных и ме-

тоды, а также, как «исключать» реализации нежелательных методов. Исключение методов необходимо при наследовании классов, когда возникает потребность ограничить круг методов, предоставляемых нашим подклассом; однако этот прием следует использовать с большой осторожностью, потому что он не соответствует ожиданиям, что подкласс может использоваться везде, где может использоваться базовый класс, то есть он нарушает принцип полиморфизма.

Собственные классы могут быть интегрированы в язык Python так, чтобы они поддерживали те же синтаксические конструкции, что и встроенные классы Python или классы из стандартной библиотеки. Достигается это за счет реализации специальных методов. Мы показали, как можно реализовать специальные методы поддержки операций сравнения, как обеспечить представление объектов в репрезентативной и строковой формах и как обеспечить преобразование в другие типы данных, такие как `int` и `float`, когда это имеет смысл. Мы также показали, как реализовать метод `__hash__()`, чтобы экземпляры нестандартных классов могли использоваться в качестве ключей словаря или членов множества.

Атрибуты данных сами по себе не обеспечивают механизм, гарантирующий установку корректных значений. Мы увидели, насколько просто атрибуты данных замещаются свойствами, что позволяет создавать атрибуты, доступные только для чтения, а для свойств, доступных для записи, легко реализовать проверку корректности записываемых данных.

В большинстве случаев классы, которые мы создаем, являются «неполными», потому что мы стремимся реализовать только те методы, которые действительно необходимы. Это вполне оправданный прием, но у нас имеется возможность создавать полностью собственные реализации классов, которые предоставляют все соответствующие методы. Мы увидели, как создаются классы, хранящие одиночные значения, путем агрегирования других классов или более компактным способом – за счет использования наследования. Мы также увидели, как создаются классы, хранящие множество значений (коллекции). Нестандартные классы коллекций могут обеспечивать те же возможности, что и встроенные классы коллекций, включая поддержку оператора `in`, функций `len()`, `iter()`, `reversed()` и оператор доступа к элементам (`[]`).

Мы узнали, что создание объекта и его инициализация – это разные операции и что язык Python позволяет контролировать выполнение обеих операций, хотя практически всегда нам требуется предусматривать собственную реализацию только для операции инициализации. Мы также узнали, что можно безопасно возвращать объекты неизменяемых атрибутов данных, но в случае изменяемых атрибутов данных почти всегда желательно возвращать их копии, чтобы избежать непреднамеренного изменения объекта и приведения его в недопустимое состояние.

В языке Python имеются обычные методы, статические методы, методы классов и функции модуля. Мы узнали, что в большинстве своем методы относятся к категории обычных методов. Иногда бывает полезно реализовать методы класса. Статические же методы используются редко, так как методы класса или функции модуля представляют собой более привлекательную альтернативу.

Встроенная функция `repr()` вызывает специальный метод `__repr__()` объекта. Желательно, чтобы выполнялось `eval(repr(x)) == x`, и мы увидели, как реализовать поддержку такой возможности. Когда отсутствует возможность создать строку репрезентативной формы, которую можно передавать функции `eval()`, мы используем метод `object.__repr__()` базового класса для воспроизведения репрезентативной формы в стандартном формате, несовместимом с функцией `eval()`.

Проверка типа объекта достаточно эффективно может быть выполнена с помощью встроенной функции `isinstance()`, хотя пуристы объектно-ориентированного стиля почти наверняка предпочли бы избегать ее использования. Доступ к методам базового класса выполняется посредством вызова встроенной функции `super()`, которая является основным способом избежать попадания в бесконечную рекурсию, когда возникает необходимость вызвать метод базового класса в переопределенной версии этого же метода в подклассе.

Функции-генераторы и методы-генераторы обеспечивают средство выполнения отложенных вычислений. Они возвращают (посредством выражения `yield`) значения по одному за запрос и возбуждают исключение `StopIteration`, когда (если это происходит) заканчиваются возвращаемые значения. Генераторы могут использоваться везде, где могут использоваться итераторы; и для конечных генераторов все их значения могут быть извлечены в кортеж или в список, если передать итератор, возвращаемый генератором, функции `tuple()` или `list()`.

Объектно-ориентированный подход практически всегда упрощает программный код в сравнении с исключительно процедурным подходом. Создавая свои классы, мы можем гарантировать доступность только допустимых операций (так как мы реализуем только соответствующие методы) и гарантировать, что ни одна из операций не переведет объект в недопустимое состояние (например, используя свойства для проверки присваиваемых значений). Начав использовать объектно-ориентированный стиль, мы практически наверняка уйдем от использования глобальных структур данных и глобальных функций, оперирующих этими данными, создавая свои классы и реализуя методы, применяемые к ним. Объектно-ориентированный подход позволяет упаковывать в единую структуру данные и методы для работы с ними. Это помогает не запутаться во всех наших данных и функциях и упрощает создание программ, простых в сопровождении, так как функциональность хранится отдельно, в виде самостоятельных классов.

Упражнения

Первые два упражнения связаны с модификацией классов, о которых рассказывалось в этой главе. Последние два упражнения связаны с созданием новых классов с самого начала.

1. Измените класс `Point` (из модуля `Shape.py` или `ShapeAlt.py`) так, чтобы обеспечить поддержку следующих операций, где `p`, `q` и `r` являются объектами типа `Point`, а `n` – число.

```
p = q + r   # Point.__add__()
p += q      # Point.__iadd__()
p = q - r   # Point.__sub__()
p -= q      # Point.__isub__()
p = q * n   # Point.__mul__()
p *= n      # Point.__imul__()
p = q / n   # Point.__truediv__()
p /= n      # Point.__itruediv__()
p = q // n  # Point.__floordiv__()
p //= n     # Point.__ifloordiv__()
```

   Каждый из методов реализации комбинированных инструкций присваивания будет состоять всего из четырех строк программного кода, а все остальные методы – из двух, включая строку с инструкцией `def`, и, конечно же, все они очень просты и похожи между собой. С минимальным описанием и доктестом для каждого из них всего добавится порядка ста тридцати новых строк. Пример решения приводится в файле *Shape_ans.py*, аналогичное решение также приводится в файле *ShapeAlt_ans.py*.

2. Измените класс в файле *Image.py* так, чтобы в нем появился метод `resize(width, height)`. Если новая ширина или высота меньше текущего значения, все цвета, оказавшиеся за пределами новых границ изображения, должны удаляться. Если в качестве нового значения ширины или высоты передается `None`, соответствующее значение ширины или высоты должно оставаться без изменений. Наконец, не забудьте воссоздавать множество `self.__colors`. Возвращаемое логическое значение должно свидетельствовать о том, были ли произведены изменения размеров или нет. Всю реализацию метода можно уместить в 20 строк (в 35, включая строку документирования и простейший доктест). Пример решения приводится в файле *Image_ans.py*.

3. Создайте класс `Transaction`, который хранит сумму, дату, валюту (по умолчанию «USD» – доллар США), курс валюты по отношению к доллару (по умолчанию 1) и описание (по умолчанию `None`). Все атрибуты данных должны быть частными. Реализуйте следующие свойства, доступные только для чтения: `amount`, `date`, `currency`, `usd_conversion_rate`, `description` и `usd` (вычисляется, как `amount * usd_conversion_rate`). Реализацию класса можно уместить в шестьдесят строк программного кода, включая несколько простейших док-

тестов. Пример решения (этого упражнения и следующего) приводится в файле *Account.py*.

4. Реализуйте класс `Account`, который хранил бы номер счета, название счета и список транзакций (объектов класса `Transaction`). Номер счета должен быть реализован в виде свойства, доступного только для чтения. Название счета должно быть реализовано в виде свойства, доступного для чтения и для записи с проверкой длины названия, которое должно содержать не менее четырех символов. Класс должен поддерживать встроенную функцию `len()` (возвращая число транзакций) и содержать два вычисляемых свойства, доступных только для чтения: `balance`, возвращающее баланс счета в долларах США, и `all_usd`, возвращающее `True`, если все транзакции выполнялись в долларах США, или `False` — в противном случае. Добавьте три дополнительных метода: `apply()` для добавления транзакции, `save()` и `load()`. Методы `save()` и `load()` должны сохранять и загружать объекты в двоичном формате, в файле, имя которого совпадает с номером счета и с расширением *.acc*. Они должны сохранять и загружать номер счета, название счета и все транзакции. Реализацию класса можно уместить в девяносто строк программного кода вместе с несколькими простейшими доктестами, включающими проверку операций сохранения и загрузки с помощью такого программного кода, как `name = os.path.join(tempfile.gettempdir(), account_name)`, который позволяет получить подходящее имя временного файла. Требуется удалить временные файлы по завершении доктестов. Пример решения приводится в файле *Account.py*.

7

- Запись и чтение двоичных данных
- Запись и синтаксический анализ текстовых файлов
- Запись и синтаксический анализ файлов XML
- Произвольный доступ к двоичным данным в файлах

Работа с файлами

В большинстве программ возникает необходимость сохранять информацию (например, данные или информацию о состоянии) в файлах и загружать ее из файлов. В языке Python имеется множество различных способов выполнять эти действия. В главе 3 мы уже коротко рассматривали вопросы работы с текстовыми файлами, а в предыдущей главе обсуждали вопрос «консервирования» объектов. В этой главе мы более детально рассмотрим работу с файлами.

Все приемы, представленные в этой главе, не зависят от типа используемой платформы. Это означает, что файл, сохраненный любым из примеров программ в одной операционной системе и аппаратной архитектуре, может быть загружен в другой операционной системе и на другой аппаратной архитектуре. Это утверждение может быть справедливым и для ваших программ тоже, если вы будете использовать те же приемы, что и в представленных здесь примерах программ.

В первых трех разделах главы рассматриваются общие случаи сохранения на диске и загрузки с диска целых коллекций данных. В первом разделе будет показано, как это можно реализовать с использованием двоичных форматов файлов, причем в первом подразделе описывается применение модуля `pickle` (с возможным сжатием), а во втором разделе демонстрируется, как ту же работу можно выполнить вручную. Во втором разделе рассматриваются приемы работы с текстовыми файлами. Запись информации в файлы выполняется очень просто, но обратное чтение может оказаться непростым делом, особенно, когда приходится иметь дело с не текстовыми данными, такими как числа и даты. Мы рассмотрим два подхода к синтаксическому разбору текста: вручную и с использованием регулярных выражений. Третий раздел рассказывает, как читать и писать файлы в формате XML. В этом разделе

будет показано, как писать и читать такие файлы с применением деревьев элементов, объектной модели документа (Document Object Model, DOM), а также как выполнять запись вручную и анализировать файлы с использованием парсера SAX (Simple API for XML – упрощенный API для работы с XML).

Четвертый раздел демонстрирует, как можно организовать произвольный доступ к данным в двоичных файлах. Это удобно, когда все элементы данных имеют одинаковый размер и когда количество элементов в файле больше, чем нам требуется (или возможно) хранить в памяти.

Какой формат файлов является более предпочтительным для хранения целых коллекций – двоичный, текстовый или XML? Как лучше работать с каждым из форматов? Ответы на эти вопросы слишком сильно зависят от конкретной ситуации, чтобы на них можно было дать единственный категоричный ответ, тем более что каждый формат имеет свои достоинства и недостатки, так же как и каждый из способов работы с ними. Мы рассмотрим каждый из них, чтобы вы могли принимать обоснованные решения в зависимости от ситуации.

При использовании двоичных форматов обычно достигается очень высокая скорость сохранения и загрузки, а, кроме того, они могут быть очень компактными. Двоичные данные не требуется анализировать, потому что каждый тип данных сохраняется в своем естественном представлении. Двоичные данные не могут читаться или редактироваться человеком, а без точного знания формата невозможно создать отдельные инструменты для работы с двоичными данными.

Текстовые форматы легко могут читаться и редактироваться человеком, что упрощает их обработку отдельными инструментами или изменение с помощью текстового редактора. Парсинг текстовых форматов может оказаться далеко не простым делом, и не всегда просто бывает выдать сообщение об ошибке, если формат текстового файла нарушен (например, в результате небрежного редактирования).

Файлы в формате XML могут читаться и редактироваться человеком, хотя они содержат большой объем служебной информации, что приводит к увеличению объемов файлов. Подобно текстовому формату, формат XML может обрабатываться отдельными инструментами. Парсинг файлов XML выполняется достаточно просто (при условии, что парсинг выполняется с помощью парсера XML, а не вручную), и некоторые парсеры способны выдавать весьма информативные сообщения об ошибках. Однако парсеры XML могут быть очень медленными, поэтому чтение очень больших файлов XML может занимать значительно больше времени, чем чтение эквивалентных им двоичных или текстовых файлов. Формат XML содержит такие метаданные, как информацию о кодировке символов (явно или неявно), которые нечасто встретишь в текстовых файлах, и это обеспечивает более высокую переносимость для файлов XML, чем для текстовых файлов.

Текстовые форматы обычно более удобны для конечного пользователя, но иногда значительные проблемы производительности делают двоичный формат единственным разумным выбором. Тем не менее всегда полезно предусмотреть возможность импорта/экспорта для формата XML, что обеспечит возможность обработки файлов инструментами сторонних разработчиков и не помешает использованию текстового или двоичного формата в процессе нормальной работы самой программы.

В трех первых разделах этой главы будет использоваться одна и та же коллекция данных: множество записей об авиационных инцидентах. В табл. 7.1 показаны имена и типы данных полей, а также ограничения, накладываемые на записи об авиационных инцидентах. В действительности совершенно неважно, какие данные мы обрабатываем. Для нас сейчас важно научиться обрабатывать фундаментальные типы данных, включая строки, целые числа, числа с плавающей точкой, логические значения и даты – научившись обрабатывать эти типы данных, мы без труда сможем обрабатывать любые другие типы данных.

Таблица 7.1. Содержимое одной записи в файле данных авиационных инцидентов

Имя	Тип данных	Примечания
report_id	str	Минимальная длина 8 символов, без пробельных символов
date	datetime.date	
airport	str	Непустое, без символов перевода строки
aircraft_id	str	Непустое, без символов перевода строки
aircraft_type	str	Непустое, без символов перевода строки
pilot_percent_hours_on_type	float	В диапазоне от 0.0 до 100.0
pilot_total_hours	int	Положительное и ненулевое значения
midair	bool	
narrative	str	Многострочный текст

Используя один и тот же набор данных об авиационных инцидентах и сохраняя его в двоичном, текстовом и XML форматах, мы получаем возможность сравнить различные форматы и объем программного кода, необходимого для работы с ними. В табл. 7.2 приводится число строк программного кода, необходимого для реализации операций чтения и записи в каждом из форматов.

Таблица 7.2. Сравнение средств чтения/записи для формата файлов с данными об авиационных инцидентах

Формат	Средство	Чтение+запись строк кода	Всего строк кода	Размер выходного файла (~Кбайт)
Двоичный	Модуль `pickle` (со сжатием gzip)	20 + 16 =	36	160
Двоичный	Модуль `pickle`	20 + 16 =	36	416
Двоичный	Вручную (со сжатием gzip)	60 + 34 =	94	132
Двоичный	Вручную	60 + 34 =	94	356
Текст	Чтение с использованием регулярных выражений, запись вручную	39 + 28 =	67	436
Текст	Вручную	53 + 28 =	81	436
XML	Дерево элементов	37 + 27 =	64	460
XML	DOM	44 + 36 =	80	460
XML	Чтение с использованием парсера SAX, запись вручную	55 + 37 =	92	464

В таблице приводятся приблизительные размеры файлов, содержащих записи о 596 авиационных инцидентах.[1] Размеры сжатых файлов с теми же данными, сохраненными под различными именами, могут отличаться на несколько байтов, так как имена файлов, которые могут иметь разную длину, включаются в состав сжатых данных. Точно так же могут отличаться размеры файлов XML, потому что одни средства записи в формате XML замещают кавычки в тексте сущностями (`&quot;` для " и `&apos;` для '), а другие – нет.

Во всех трех первых разделах рассматривается программный код одной и той же программы: *convert-incidents.py*. Эта программа используется для чтения информации об инцидентах в одном формате и для записи в другом формате. Ниже приводится справочный текст, который выводится программой в консоли. (Мы немного отформатировали текст, чтобы уместить его в ширину книжной страницы.)

```
Usage: convert-incidents.py [options] infile outfile

Reads aircraft incident data from infile and writes the data to
outfile. The data formats used depend on the file extensions:
.aix is XML, .ait is text (UTF-8 encoding), .aib is binary,
.aip is pickle, and .html is HTML (only allowed for the outfile).
All formats are platform-independent.
```

[1] В примерах используются реальные данные об авиационных инцидентах, которые можно найти на сайте FAA (Федеральное авиационное управление правительства США, *www.faa.gov*)

```
Options:
  -h, --help      show this help message and exit
  -f, --force     write the outfile even if it exists [default: off]
  -v, --verbose   report results [default: off]
  -r READER, --reader=READER
                  reader (XML): 'dom', 'd', 'etree', 'e', 'sax', 's'
                  reader (text): 'manual', 'm', 'regex', 'r'
                  [default: etree for XML, manual for text]
  -w WRITER, --writer=WRITER
                  writer (XML): 'dom', 'd', 'etree', 'e',
                  'manual', 'm' [default: manual]
  -z, --compress  compress .aib/.aip outfile [default: off]
  -t, --test      execute doctests and exit (use with -v for verbose)
(Перевод:
Порядок использования: convert-incidents.py [параметры] infile outfile

Читает данные об авиационных инцидентах из файла infile и записывает их
в файл outfile. Форматы файлов определяются по их расширениям:
.aix - XML, .ait - текст (в кодировке UTF-8), .aib - двоичный,
.aip - формат модуля pickle и .html - HTML (только для выходного
файла outfile).
Все форматы являются платформонезависимыми.

Параметры:
  -h, --help     вывести текст этого сообщения и выйти
  -f, --force    выполнять запись в outfile, даже если он существует
                 [по умолчанию запись в существующий файл не производится]
  -v, --verbose  вывести результаты [по умолчанию: отключено]
  -r READER, --reader=READER
                  средство чтения (XML): 'dom', 'd', 'etree', 'e', 'sax', 's'
                 средство чтения (текст): 'manual', 'm', 'regex', 'r'
                 [по умолчанию: etree для XML, manual для текста]
  -w WRITER, --writer=WRITER
                 средство записи (XML): 'dom', 'd', 'etree', 'e',
                 'manual', 'm' [по умолчанию: manual]
  -z, --compress  сжатие для выходных файлов .aib/.aip
                 [по умолчанию: отключено]
  -t, --test     выполнить доктесты и выйти (для вывода подробного
                 отчета о прохождении тестов используйте
                 параметр -v) конец перевода)
```

Параметры, используемые программой, более сложные, чем обычно могло бы потребоваться конечному пользователю, которого мало беспокоит, какое средство чтения или записи используется для любого из поддерживаемых форматов. В более реалистичной версии программы параметры, управляющие выбором средств чтения и записи, отсутствовали бы, и мы просто использовали бы по одному средству чтения и одному средству записи для каждого из форматов. Точно так же в окончательной версии программы отсутствовал бы параметр тестирования, который предоставляется исключительно для того, чтобы мы могли протестировать программный код.

Программа определяет собственное исключение:

```
class IncidentError(Exception): pass
```

Информация об авиационных инцидентах хранится в виде объектов класса `Incident`. Ниже приводится строка с инструкцией `class` и метод инициализации:

```
class Incident:

    def __init__(self, report_id, date, airport, aircraft_id,
                 aircraft_type, pilot_percent_hours_on_type,
                 pilot_total_hours, midair, narrative=""):
        assert len(report_id) >= 8 and len(report_id.split()) == 1, \
                  "invalid report ID"
        self.__report_id = report_id
        self.date = date
        self.airport = airport
        self.aircraft_id = aircraft_id
        self.aircraft_type = aircraft_type
        self.pilot_percent_hours_on_type = pilot_percent_hours_on_type
        self.pilot_total_hours = pilot_total_hours
        self.midair = midair
        self.narrative = narrative
```

При создании объекта `Incident` проверяется идентификатор отчета и делается доступным только для чтения в виде свойства `repotr_id`. Все остальные атрибуты данных представляют собой свойства, доступные для чтения и для записи. Например, ниже приводится программный код объявления свойства `date`:

```
@property
def date(self):
    return self.__date

@date.setter
def date(self, date):
    assert isinstance(date, datetime.date), "invalid date"
    self.__date = date
```

Все остальные свойства объявляются точно так же, отличаясь только некоторыми особенностями инструкции `assert`, поэтому мы не будем приводить их здесь. Поскольку для проверки мы используем инструкции `assert`, программа будет завершаться аварийно при любой попытке создать объект `Incident` с недопустимыми данными или при попытке записать недопустимое значение в любое из свойств, доступных для чтения/записи. Такой бескомпромиссный подход был выбран потому, что нам необходимо гарантировать допустимость загружаемых и сохраняемых данных, а в случае ошибки нам требуется, чтобы программа сообщала о ней и завершала свою работу, вместо того чтобы просто продолжать работу.

Коллекция данных об инцидентах хранится в объекте типа `IncidentCollection`. Этот класс наследует класс `dict`, благодаря чему мы получа-

ем в свое распоряжение массу функциональных возможностей, таких как поддержка оператора доступа к элементам ([ ]) для получения, создания и удаления отдельных записей об инцидентах. Ниже приводится строка с инструкцией class и несколько методов класса:

```
class IncidentCollection(dict):

    def values(self):
        for report_id in self.keys():
            yield self[report_id]

    def items(self):
        for report_id in self.keys():
            yield (report_id, self[report_id])

    def __iter__(self):
        for report_id in sorted(super().keys()):
            yield report_id

    keys = __iter__
```

Нам не потребовалось переопределять специальный метод инициализации, потому что вполне достаточно функциональности унаследованного метода dict.__init__(). Ключами словаря являются идентификаторы отчетов, а значениями – объекты Incident. Мы переопределили методы values(), items() и keys() так, чтобы возвращаемые ими итераторы обеспечивали выполнение итераций в порядке сортировки идентификаторов отчетов. Такое поведение обусловлено тем, что методы values() и items() используют итератор по ключам, возвращаемый методом IncidentCollection.keys(), а этот метод (который имеет еще одно имя: IncidentCollection.__iter__()) выполняет в порядке сортировки итерации по ключам, возвращаемым методом dict.keys() базового класса.

Дополнительно класс IncidentCollection имеет методы export() и import_(). (Мы использовали завершающий символ подчеркивания, чтобы обеспечить отличие имени метода от встроенной инструкции import.) Методу export() передается имя файла и в виде необязательных аргументов – средство записи и флаг сжатия, а он на основе имени файла и средства записи передает управление более конкретному методу, такому как export_xml_dom() или export_xml_etree(). Метод import_() принимает имя файла и средство чтения в виде необязательного аргумента и работает похожим образом. Методам импортирования, работающим с двоичными форматами, не передается информация о том, был ли сжат файл – как ожидается, они сами будут определять это и работать соответственно.

Запись и чтение двоичных данных

Двоичные форматы даже без сжатия обычно являются более компактными и, как правило, обеспечивают более высокую скорость сохране-

ния и загрузки. Наиболее простой способ заключается в использовании модуля pickle, хотя при обработке двоичных данных вручную обычно получаются файлы меньшего размера.

Консервирование с возможным сжатием

Консервирование является наиболее простым подходом к выполнению операций сохранения и загрузки данных в программах на языке Python, но, как уже отмечалось в предыдущей главе, процедура консервирования не имеет механизмов обеспечения безопасности (шифрование, цифровая подпись), поэтому загрузка законсервированных объектов из непроверенных источников может оказаться опасной. Проблема безопасности обусловлена тем, что законсервированные объекты могут импортировать произвольные модули и вызывать произвольные функции, то есть можно создать такой законсервированный объект, который, к примеру, после загрузки будет заставлять интерпретатор выполнять неблаговидные действия. Тем не менее консервирование часто является идеальным средством для работы с узкоспециализированными данными, особенно в программах, предназначенных для личного пользования.

Обычно намного проще сначала выработать формат файла и написать программный код, выполняющий сохранение, а потом написать программный код, выполняющий загрузку, поэтому мы начнем с того, что рассмотрим реализацию консервирования коллекции записей об инцидентах.

```
def export_pickle(self, filename, compress=False):
    fh = None
    try:
        if compress:
            fh = gzip.open(filename, "wb")
        else:
            fh = open(filename, "wb")
        pickle.dump(self, fh, pickle.HIGHEST_PROTOCOL)
        return True
    except (EnvironmentError, pickle.PicklingError) as err:
        print("{0}: export error: {1}".format(
              os.path.basename(sys.argv[0]), err))
        return False
    finally:
        if fh is not None:
            fh.close()
```

Если было запрошено сжатие, для открытия файла используется функция gzip.open() из модуля gzip, в противном случае используется встроенная функция open(). При консервировании данных в двоичном формате мы должны использовать двоичный режим записи ("wb").

(В Python 3.0 константа `pickle.HIGHEST_PROTOCOL` обозначает протокол 3, соответствующий компактному двоичному формату.[1])

Менеджеры контекста, стр. 428

В случае появления ошибок мы предпочитаем сразу же сообщать о них пользователю и возвращать вызывающей программе логическое значение, свидетельствующее об успехе или неудаче. В методе используется блок `finally`, чтобы обеспечить закрытие файла независимо от наличия ошибки. В главе 8 будет представлен более компактный способ закрытия файлов, в котором не используется блок `finally`.

Этот программный код очень напоминает то, что мы уже видели в предыдущей главе, но здесь есть один тонкий момент, о котором необходимо упомянуть. Консервированию подвергается объект `self` класса `dict`. Но значениями словаря являются объекты класса `Incident`, то есть объекты нашего собственного класса. Модуль `pickle` достаточно интеллектуален, чтобы сохранять объекты почти любых наших классов без нашего вмешательства.

Специальный метод `__dict__()`, стр. 422

Вообще, консервироваться могут логические значения, числа и строки, а также экземпляры классов, включая нестандартные классы, предоставляющие частный атрибут `__dict__`. Кроме того, консервироваться могут любые встроенные типы коллекций (кортежи, списки, множества, словари), если они содержат только объекты, допускающие возможность консервирования (включая коллекции, то есть поддерживаются рекурсивные структуры). Имеется также возможность консервировать другие типы объектов или экземпляры нестандартных классов, которые обычно не могут консервироваться (например, потому что они имеют атрибуты, не допускающие возможность консервирования), для чего достаточно или оказать некоторую помощь модулю `pickle`, или реализовать функции сохранения и загрузки. Все необходимые подробности вы найдете в электронной документации к модулю `pickle`.

Чтобы прочитать законсервированные данные, необходимо определить – были ли они сжаты или нет. Любой файл, сжатый с использованием алгоритма gzip, начинается с *сигнатуры файла* (*magic number*). Сигнатура – это последовательность из одного или более байтов в начале файла, используемая для обозначения типа файла. Для обозначения файлов, сжатых с использованием алгоритма gzip, используется

[1] Протокол 3 впервые появился только в Python 3. Если необходимо создавать файлы, доступные для чтения и записи программам, работающим под управлением Python 2 и Python 3, необходимо использовать протокол 2.

сигнатура из двух байтов 0x1F 0x8B, которые мы сохраняем в переменной типа bytes:

```
GZIP_MAGIC = b"\x1F\x8B"
```

Подробнее о типе данных bytes рассказывается во врезке «Типы данных bytes и bytearray» (стр. 344), а в табл. 7.3 (стр. 345–347) перечисляются их методы.

Ниже приводится программный код, выполняющий чтение законсервированных данных:

```
def import_pickle(self, filename):
    fh = None
    try:
        fh = open(filename, "rb")
        magic = fh.read(len(GZIP_MAGIC))
        if magic == GZIP_MAGIC:
            fh.close()
            fh = gzip.open(filename, "rb")
        else:
            fh.seek(0)
        self.clear()
        self.update(pickle.load(fh))
        return True
    except (EnvironmentError, pickle.UnpicklingError) as err:
        print("{0}: import error: {1}".format(
              os.path.basename(sys.argv[0]), err))
        return False
    finally:
        if fh is not None:
            fh.close()
```

Мы не знаем заранее, был файл сжат или нет. В любом случае, мы начинаем с того, что открываем файл для чтения в двоичном режиме, а затем читаем первые два байта. Если эти два байта представляют сигнатуру gzip, файл закрывается и создается новый объект файла вызовом функции gzip.open(). Если файл не был сжат, используется объект файла, созданный функцией open(); вызовом его метода seek() указатель позиции в файле перемещается в начало, чтобы следующая операция чтения (выполняемая внутри функции pickle.load()) начала чтение файла с самого начала.

Мы не можем выполнить прямое присваивание объекту self, так как это приведет к уничтожению используемого объекта типа IncidentCollection, поэтому сначала мы удаляем все элементы словаря и затем с помощью метода dict.update() заполняем словарь объектами с информацией об инцидентах из словаря типа IncidentCollection, загруженного из файла.

Обратите внимание, что порядок следования байтов в машинном слове для данной аппаратной архитектуры не имеет никакого значения, по-

тому что при чтении сигнатуры мы читаем два отдельных байта, а когда модуль pickle читает основные данные, он сам заботится о порядке следования байтов.

Типы данных bytes и bytearray

В языке Python имеется два типа данных, которые используются для работы с обычными байтами: тип bytes – неизменяемый и тип bytearray – изменяемый. Оба типа хранят последовательности из нуля или более 8-битовых беззнаковых целых чисел (байтов), где каждый байт может представлять число в диапазоне 0...255.

Метод str.translate(), стр. 99

Оба типа очень похожи на строки и предоставляют практически те же методы, включая поддержку срезов. Кроме того, тип данных bytearray предоставляет несколько методов, напоминающих методы класса list, позволяющих производить изменения внутри объекта bytearray. Все методы этих двух типов данных перечислены в табл. 7.3 (стр. 345–347).

Несмотря на то, что операция извлечения среза для объектов bytes и bytearray возвращает объект того же самого типа, тем не менее оператор доступа к элементу ([]) возвращает объект типа int – значение заданного байта. Например:

```
word = b"Animal"
x = b"A"
word[0] == x     # вернет: False  # word[0] == 65;    x == b"A"
word[:1] == x    # вернет: True   # word[:1] == b"A"; x == b"A"
word[0] == x[0]  # вернет: True   # word[0] == 65;    x[0] == 65
```

Ниже приводятся еще несколько примеров использования объектов типа bytes и bytearray:

```
data = b"5 Hills \x35\x20\x48\x69\x6C\x6C\x73"
data.upper()                          # вернет: b'5 HILLS 5 HILLS'
data.replace(b"ill", b"at")           # вернет: b'5 Hats 5 Hats'
bytes.fromhex("35 20 48 69 6C 6C 73") # вернет: b'5 Hills'
bytes.fromhex("352048696C6C73")       # вернет: b'5 Hills'
data = bytearray(data)                # теперь data имеет тип bytearray
data.pop(10)                          # вернет: 72 (ord("H"))
data.insert(10, ord("B"))             # data == b'5 Hills 5 Bills'
```

Методы, имеющие смысл только применительно к строкам, такие как bytes.upper(), предполагают, что байты соответствуют символам из набора ASCII. Метод класса bytes.fromhex() игнорирует пробелы и интерпретирует каждую подстроку из двух цифр как шестнадцатеричное число, то есть строка "35" будет преобразована в байт со значением 0x35, и т. д.

Таблица 7.3. Методы объектов типа bytes и bytearray

Синтаксис	Описание
`ba.append(i)`	Добавляет целое число `i` (в диапазоне 0...255) в объект `ba` типа `bytearray`
`b.capitalize()`	Возвращает копию объекта `b` типа `bytes` или `bytearray`, с первым символом в верхнем регистре (если это символ ASCII)
`b.center(width, byte)`	Возвращает копию объекта `b`, отцентрированную в поле шириной `width`. Недостающие символы по умолчанию заполняются пробелами или символами, в соответствии с необязательным аргументом *byte*
`b.count (x, start, end)`	Возвращает число вхождений объекта `x` типа `bytes` или `bytearray`, в объект `b` типа `bytes` или `bytearray` (или в срез `b[start:end]`)
`b.decode (encoding, error)`	Возвращает объект типа `str`, представляющий результат декодирования байтов с использованием кодировки UTF-8 или кодировки, определяемой аргументом *encoding*, с обработкой ошибок, определяемой необязательным аргументом *err*
`b.endswith (x, start, end)`	Возвращает `True`, если `b` (или срез `b[start:end]`) оканчивается содержимым объекта `x` типа `bytes` или `bytearray` или любым из объектов типа `bytes` или `bytearray` в кортеже `x`; в противном случае возвращает `False`
`b.expandtabs (size)`	Возвращает копию объекта `b`, в котором символы табуляции замещены пробелами с шагом 8 или в соответствии со значением необязательного аргумента *size*
`ba.extend(seq)`	Дополняет объект `ba` типа `bytearray` целыми числами из последовательности `seq`. Все целые числа должны находиться в диапазоне 0...255
`b.find (x, start, end)`	Возвращает позицию самого первого (крайнего слева) вхождения объекта `x` типа `bytes/bytearray` в объект `b` (или в срез `b[start:end]`); если объект `x` не найден, возвращается -1. Для поиска самого последнего (крайнего справа) вхождения следует использовать метод `rfind()`
`b.fromhex(h)`	Возвращает объект типа `bytes`, который содержит байты, соответствующие шестнадцатеричным значениям в строке `h`
`b.index (x, start, end)`	Возвращает позицию самого первого (крайнего слева) вхождения объекта `x` в объект `b` (или в срез строки `b[start:end]`); если объект `x` не найден, возбуждается исключение `ValueError`. Для поиска самого последнего (крайнего справа) вхождения следует использовать метод `rindex()`
`ba.insert(p, i)`	Вставляет целое число `i` (в диапазоне 0...255) в позицию `p` в объекте `ba`

Кодировки символов, стр. 112

Таблица 7.3 (продолжение)

Синтаксис	Описание
`b.isalnum()`	Возвращает `True`, если объект `b` типа `bytes/bytearray` не пустой и содержит только алфавитно-цифровые символы ASCII
`b.isalpha()`	Возвращает `True`, если объект `b` типа `bytes/bytearray` не пустой и содержит только алфавитные символы ASCII
`b.isdigit()`	Возвращает `True`, если объект `b` типа `bytes/bytearray` не пустой и содержит только цифровые символы ASCII
`b.islower()`	Возвращает `True`, если объект `b` типа `bytes/bytearray` содержит хотя бы один символ ASCII, который может быть представлен в нижнем регистре, и все такие символы находятся в нижнем регистре
`b.isspace()`	Возвращает `True`, если объект `b` типа `bytes/bytearray` не пустой и содержит только пробельные символы из набора ASCII
`b.istitle()`	Возвращает `True`, если объект `b` не пустой и имеет формат заголовка
`b.isupper()`	Возвращает `True`, если объект `b` типа `bytes/bytearray` содержит хотя бы один символ ASCII, который может быть представлен в верхнем регистре, и все такие символы находятся в верхнем регистре
`b.join(seq)`	Объединяет все элементы типа `bytes/bytearray` в последовательности `seq`, вставляя между ними объект `b` (который может быть пустым)
`b.ljust (width, byte)`	Возвращает копию объекта `b` типа `bytes/bytearray` выровненной по левому краю в поле шириной `width`. Недостающие символы по умолчанию заполняются пробелами или символами, в соответствии с необязательным аргументом *byte*. Для выравнивания по правому краю используйте метод `rjust()`
`b.lower()`	Возвращает копию объекта `b` типа `bytes/bytearray`, в котором все символы ASCII приведены к нижнему регистру
`b.partition(sep)`	Возвращает кортеж с тремя объектами типа `bytes`: часть `b` перед самым первым вхождением содержимого объекта `sep`, сам объект `sep` и часть `b` после самого первого вхождения содержимого объекта `sep`. Если содержимое объекта `sep` не будет найдено, возвращается объект `b` и два пустых объекта `bytes`. Для деления объекта `b` по самому правому вхождению содержимого объекта `sep` используйте метод `rpartition()`
`ba.pop(p)`	Удаляет и возвращает целое число, находящееся в объекте `ba` в позиции `p`
`ba.remove(i)`	Удаляет первое вхождение целого числа `i` из объекта `ba` типа `bytearray`

Синтаксис	Описание
`b.replace (x, y, n)`	Возвращает копию объекта `b`, в котором каждое (но не более *n*, если этот аргумент определен) вхождение объекта `x` типа `bytes/bytearray` замещается объектом `y`
`ba.reverse()`	Переставляет в памяти элементы объекта `ba` типа `bytearray` в обратном порядке
`b.split(x, n)`	Возвращает список объектов типа `bytes`, выполняя разбиение объекта `b` не более чем *n* раз по содержимому объекта `x`. Если число *n* не задано, разбиение выполняется по всем найденным вхождениям объекта `x`. Если объект `x` не задан, разбиение выполняется по пробельным символам. Для выполнения разбиения строки, начиная с правого края, используйте метод `rsplit`.
`b.splitlines(f)`	Возвращает список строк, выполняя разбиение объекта `b` по символам перевода строки, удаляя их, если в аргументе `f` не задано значение `True`
`b.startswith (x start, end)`	Возвращает `True`, если объект `b` типа `bytes/bytearray` (или срез `b[start:end]`) начинается содержимым объекта `x` типа `bytes/bytearray` или содержимым любого объекта `x` типа `bytes/bytearray`, если `x` – это кортеж; в противном случае возвращает `False`
`b.strip(x)`	Возвращает копию объекта `b`, из которого удалены начальные и завершающие пробельные символы (или байты, входящие в объект `x` типа `bytes/bytearray`). Метод `lstrip()` выполняет удаление только начальных символов (или байтов), а метод `rstrip()` – конечных
`b.swapcase()`	Возвращает копию объекта `b`, в котором все символы ASCII верхнего регистра преобразованы в символы нижнего регистра, а все символы ASCII нижнего регистра – в символы верхнего регистра
`b.title()`	Возвращает копию объекта `b`, в котором первые символы ASCII каждого слова преобразованы в символы верхнего регистра, а все остальные символы ASCII – в символы нижнего регистра
`b.translate (bt, d)`	Возвращает копию объекта `b`, из которой удаляются все байты, входящие в объект `d`, а все остальные байты замещаются байтами из объекта `bt`, причем индекс байта в объекте `bt` определяется значением байта в объекте `b`
`b.upper()`	Возвращает копию объекта `b` типа `bytes/bytearray`, в котором все символы ASCII приведены к верхнему регистру
`b.zfill(w)`	Возвращает копию объекта `b`, который, если его длина меньше величины `w`, дополняется слева символами нуля (байт `0x30`) до длины `w`

Неформатированные двоичные данные с возможным сжатием

Написание собственного программного кода для работы с двоичными данными обеспечивает нам полный контроль над форматом файла. Кроме того, такой подход является более безопасным, чем использование модуля `pickle`, поскольку злонамеренные, недопустимые данные будут обрабатываться нашим программным кодом, а не интерпретатором.

Разрабатывая собственные двоичные форматы файлов, совсем нелишним будет предусмотреть сигнатуру для идентификации типа файла и номер версии для идентификации версии используемого формата. Ниже приводятся определения, используемые в программе *convert-incidents.py*:

```
MAGIC = b"AIB\x00"
FORMAT_VERSION = b"\x00\x01"
```

Мы использовали четыре байта для сигнатуры и два байта для обозначения версии. Порядок следования байтов в машинном слове не имеет значения, потому что будут записываться отдельные байты, а не целые числа в байтовом представлении, то есть последовательность байтов будет одна и та же на любой аппаратной архитектуре.

Для записи и чтения двоичных данных вручную нам необходимо некоторое средство преобразования объектов Python в соответствующее двоичное представление и средство преобразования двоичных данных в объекты Python. Большая часть необходимых нам функциональных возможностей предоставляется модулем `struct`, краткое описание которого приводится во врезке «Модуль struct» (стр. 349), и типами данных `bytes` и `bytearray`, краткое описание которых приводится во врезке «Типы данных bytes и bytearray» (стр. 344).

К сожалению, модуль `struct` может обрабатывать строки только определенной длины, тогда как в нашем случае идентификаторы отчетов, названия аэропортов, типы самолетов и текст комментариев могут быть представлены строками переменной длины. Чтобы удовлетворить требования модуля `struct`, мы создали функцию `pack_string()`, которая принимает строку и возвращает объект `bytes`, состоящий из двух компонентов: первый – это целое число, определяющее длину строки, и второй – последовательность байтов в кодировке UTF-8, представляющих текст строки.

Локальные функции, стр. 409

Так как функция `pack_string()` будет вызываться только внутри функции `export_binary()`, мы поместили определение `pack_string()` внутрь функции `export_binary()`. Это означает, что функция `pack_string()` будет недоступна за пределами функции `export_binary()`, и указывает, что это всего лишь локальная, вспомогательная функция. Ниже

приводится начало функции `export_binary()` и полное определение функции `pack_string()`:

```
def export_binary(self, filename, compress=False):

    def pack_string(string):
        data = string.encode("utf8")
        format = "<H{0}s".format(len(data))
        return struct.pack(format, len(data), data)
```

Метод `str.encode()` возвращает объект `bytes` со строкой, закодированной в соответствии с указанной кодировкой. Кодировка UTF-8 является очень удобной, потому что она может представить любой символ Юникода и обеспечивает компактное представление символов ASCII (по одному байту на символ). В переменной `format` сохраняется формат представления строки с длиной. Например, пусть имеется строка «en.wikipedia.org», тогда ее форматом будет строка `"<H16s"` (обратный порядок следования байтов, 2-байтовое целое число без знака, 16-байтовая строка), а получившийся объект `bytes` будет иметь вид: `b'\x10\x00en.wikipedia.org'`. Для удобства интерпретатор Python отображает объекты типа `bytes` в компактной форме, используя печатаемые символы ASCII, если это возможно, и экранированные шестнадцатеричные значения (и некоторые специальные экранированные последовательности, такие как `\t` и `\n`) в противном случае.

Кодировки символов, стр. 112

Модуль struct

Модуль `struct` предоставляет функции `struct.pack()`, `struct.unpack()` и ряд других функций, а также класс `struct.Struct()`. Функция `struct.pack()` принимает строку формата и одно или более значений и возвращает объект `bytes`, хранящий значения, представленные в соответствии с указанным форматом. Функция `struct.unpack()` принимает формат и объект `bytes` или `bytearray` и возвращает кортеж значений, ранее упакованных с использованием строки формата. Например:

```
data = struct.pack("<2h", 11, -9)   # data == b'\x0b\x00\xf7\xff'
items = struct.unpack("<2h", data)  # items == (11, -9)
```

Строка формата состоит из одного или более символов. Большинство символов представляют значение определенного типа. Если имеется несколько значений одного и того же типа, мы можем записать символ требуемое число раз (`"hh"`) или указать количество значений перед символом, как это сделано в примерах выше (`"2h"`).

В электронной документации к модулю `struct` описывается множество символов формата, включая «b» (8-битовое целое число со знаком), «B» (8-битовое целое число без знака), «h» (16-битовое целое число со знаком – используется в примерах выше), «H» (16-битовое целое число без знака), «i» (32-битовое целое число со знаком), «I» (32-битовое целое число без знака), «q» (64-битовое целое число со знаком), «Q» (64-битовое целое число без знака), «f» (32-битовое число с плавающей точкой), «d» (64-битовое число с плавающей точкой – соответствует типу `float` в языке Python), «?» (логическое значение), «s» (объект типа `bytes` или `bytearray` – строки байтов) и многие другие.

Для некоторых типов данных, таких как многобайтовые целые числа, большое значение имеет порядок следования байтов. Мы можем принудительно использовать какой-то определенный порядок следования байтов независимо от порядка следования байтов, используемого аппаратной архитектурой, для чего строка формата должна начинаться с символа, определяющего порядок следования байтов. В этой книге мы везде будем использовать символ «<», обозначающий обратный порядок следования байтов, который используется в широко распространенных процессорах Intel и AMD. Прямой порядок следования байтов (иногда его называют сетевым порядком следования байтов) обозначается символом «>» (или «!»). Если порядок следования байтов не указан явно, применяется порядок, используемый аппаратной архитектурой машины. Мы рекомендуем всегда явно указывать порядок следования байтов, даже если он совпадает с аппаратным, так как это обеспечит более высокую переносимость.

Функция `struct.calcsize()` принимает строку формата и возвращает количество байтов, которые займет структура указанного формата. Строку формата можно также сохранить, создав объект типа `struct.Struct()`, передав строку формата в виде аргумента, а размер объекта `struct.Struct()` можно получить, обратившись к его атрибуту `size`. Например:

```
TWO_SHORTS = struct.Struct("<2h")
data = TWO_SHORTS.pack(11, -9)   # data == b'\x0b\x00\xf7\xff'
items = TWO_SHORTS.unpack(data)  # items == (11, -9)
```

В обоих примерах число `11` в шестнадцатеричном представлении имеет вид `0x000b`, но оно преобразуется в последовательность байтов `0x0b 0x00`, потому что мы использовали обратный порядок следования байтов.

Функция `pack_string()` может обрабатывать строки, содержащие до 65535 символов в кодировке UTF-8. Мы легко могли бы использовать другой тип целого числа для хранения числа байтов, например, 4-байтовое целое число со знаком (формат «i») позволило бы обрабатывать строки, содержащие до $2^{31}-1$ (более 2 миллиардов) символов.

Модуль `struct` предоставляет похожий встроенный формат, «p», описывающий строки, в которых первый байт используется в качестве счетчика символов, применив который мы смогли бы обрабатывать строки, содержащие до 255 символов. Программный код, выполняющий упаковывание строк с применением формата «p», выглядит намного проще. Но формат «p» ограничивает максимально возможную длину строк 255 символами UTF-8 и практически не дает преимуществ при распаковывании. (Исключительно ради сравнения в файл с исходными текстами *convert-incidents.py* включены версии функций `pack_string()` и `unpack_string()`.)

Теперь можно все наше внимание переключить на остальную часть программного кода в методе `export_binary()`.

```
fh = None
try:
    if compress:
        fh = gzip.open(filename, "wb")
    else:
        fh = open(filename, "wb")
    fh.write(MAGIC)
    fh.write(FORMAT_VERSION)
    for incident in self.values():
        data = bytearray()
        data.extend(pack_string(incident.report_id))
        data.extend(pack_string(incident.airport))
        data.extend(pack_string(incident.aircraft_id))
        data.extend(pack_string(incident.aircraft_type))
        data.extend(pack_string(incident.narrative.strip()))
        data.extend(NumbersStruct.pack(
                        incident.date.toordinal(),
                        incident.pilot_percent_hours_on_type,
                        incident.pilot_total_hours,
                        incident.midair))
        fh.write(data)
    return True
```

Мы опустили блоки `except` и `finally`, потому что они остались такими же, как и в предыдущем подразделе, кроме собственно исключений, обрабатываемых в блоке `except`.

В самом начале мы открываем файл для записи в двоичном режиме либо как обычный файл, либо как сжатый файл, в зависимости от значения флага `compress`. Затем записываются 4-байтовая сигнатура, уникальная (надеемся) для нашей программы, и 2-байтовый номер вер-

сии.[1] Предусмотрев номер версии, мы упрощаем возможность изменения формата в будущем – прочитав номер версии, мы сможем определить, какой программный код использовать для чтения файла.

Затем выполняется обход всех записей об инцидентах, и для каждой создается объект типа `bytearray`. Каждый элемент данных добавляется в массив байтов, начиная со строк переменной длины. Метод `date.toordinal()` возвращает единственное целое число, представляющее сохраняемую дату. Позднее дату можно будет восстановить, передав это целое число методу `datetime.date.fromordinal()`. Объект `NumbersStruct` был определен выше в программе следующей инструкцией:

```
NumbersStruct = struct.Struct("<Idi?")
```

Этот формат определяет обратный порядок следования байтов, 32-битовое целое число без знака (для хранения даты), 64-битовое число с плавающей точкой (налет на данном типе самолетов в процентах от общего времени налета пилота), 32-битовое целое число (общее время налета пилота в часах) и логическое значение (признак того, что инцидент произошел в воздухе). Структура записи об авиационном инциденте схематически изображена на рис. 7.1.

После заполнения объекта `bytearray` полной информацией об одном инциденте производится запись объекта на диск. После записи всех инцидентов возвращается значение `True` (здесь предполагается, что в процессе записи не возникло никаких ошибок). Блок `finally` гарантирует закрытие файла перед тем, как управление будет возвращено вызывающей программе.

Чтение данных выполняется не так просто, как их запись, потому что при чтении приходится выполнять бо́льшее количество проверок на

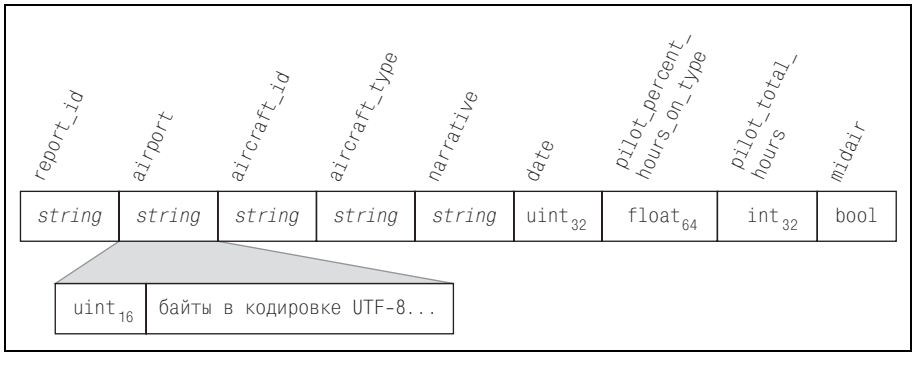

Рис. 7.1. *Структура записи об авиационном инциденте в двоичном формате*

[1] Нигде не существует центрального репозитория сигнатур, какой существует, например, для доменных имен, поэтому мы никогда не можем гарантировать их уникальность.

наличие ошибок. Кроме того, чтение строк переменной длины сопряжено с определенными сложностями. Ниже приводится часть метода `import_binary()` и полное определение функции `unpack_string()`, которая используется для чтения строк переменной длины:

```
def import_binary(self, filename):

    def unpack_string(fh, eof_is_error=True):
        uint16 = struct.Struct("<H")
        length_data = fh.read(uint16.size)
        if not length_data:
            if eof_is_error:
                raise ValueError("missing or corrupt string size")
            return None
        length = uint16.unpack(length_data)[0]
        if length == 0:
            return ""
        data = fh.read(length)
        if not data or len(data) != length:
            raise ValueError("missing or corrupt string")
        format = "<{0}s".format(length)
        return struct.unpack(format, data)[0].decode("utf8")
```

Поскольку каждая запись об инциденте начинается строкой идентификатора, следовательно, когда попытка чтения строки идентификатора оканчивается успехом, это означает, что было начато чтение новой записи. Неудача означает, что достигнут конец файла и можно прекращать чтение. При попытке прочитать строку идентификатора отчета в аргументе `eof_is_error` передается значение `False`; в этом случае отсутствие данных будет означать просто окончание операции чтения. При чтении всех других строк аргумент `eof_is_error` получает значение по умолчанию `True`, потому что отсутствие данных для любых других строк означает ошибку. (Даже пустой строке будет предшествовать 16-битовое целое число без знака, обозначающее длину строки.)

Чтение строки начинается с попытки прочитать ее длину. Если эта попытка терпит неудачу, вызывающей программе возвращается значение `None`, чтобы сообщить о достижении конца файла (если производится попытка прочитать новую запись), или возбуждается исключение `ValueError`, чтобы сообщить о повреждении или об отсутствии данных. Функция `struct.unpack()` и метод `struct.Struct.unpack()` всегда возвращают кортеж, даже если он содержит единственное значение. Мы извлекаем значение длины строки и сохраняем его в переменной `length`. Теперь известно, сколько байтов следует прочитать из файла, чтобы получить строку. Если длина равна нулю, функция просто возвращает пустую строку. В противном случае производится попытка прочитать заданное число байтов. Если в результате попытки чтения было получено меньшее число байтов или вообще ничего не было получено, возбуждается исключение `ValueError`.

Если было прочитано требуемое количество байтов, создается соответствующая строка формата для функции `struct.unpack()` и вызывающей программе возвращается строка с извлеченными данными, представляющая результат декодирования байтов с использованием кодировки UTF-8. (Теоретически последние две строки можно было бы заменить инструкцией `return data.decode("utf-8")`, но мы предпочли пройти через процесс распаковывания, так как вполне возможно, хотя и маловероятно, что формат «s» выполняет некоторые преобразования, которые должны быть применены к данным при чтении.)

Теперь рассмотрим оставшуюся часть метода `import_binary()`, разбив ее на две части для простоты объяснения.

```
fh = None
try:
    fh = open(filename, "rb")
    magic = fh.read(len(GZIP_MAGIC))
    if magic == GZIP_MAGIC:
        fh.close()
        fh = gzip.open(filename, "rb")
    else:
        fh.seek(0)
    magic = fh.read(len(MAGIC))
    if magic != MAGIC:
        raise ValueError("invalid .aib file format")
    version = fh.read(len(FORMAT_VERSION))
    if version > FORMAT_VERSION:
        raise ValueError("unrecognized .aib file version")
    self.clear()
```

Файл может быть сжатым, поэтому здесь используется тот же прием, что использовался при чтении файла с законсервированным объектом, – открытие файла либо с помощью функции `gzip.open()`, либо с помощью встроенной функции `open()`.

После открытия файла производится чтение первых четырех байтов (`len(MAGIC)`). Если они не соответствуют нашей сигнатуре, это говорит о том, что файл не является двоичным файлом с информацией об авиационных инцидентах, и поэтому возбуждается исключение `ValueError`. Затем производится чтение двух байтов с номером версии. С этого момента можно было бы реализовать различные процедуры чтения – в зависимости от номера версии. Сейчас же просто проверяется, не является ли номер версии более поздним, чем тот, что может быть прочитан программой.

Если сигнатура оказалась верной и номер версии соответствует тому, что может быть обработан, можно приступать к чтению данных, поэтому производится удаление всех существующих элементов с информацией об инцидентах.

```
while True:
    report_id = unpack_string(fh, False)
```

```
            if report_id is None:
                break
            data = {}
            data["report_id"] = report_id
            for name in ("airport", "aircraft_id",
                         "aircraft_type", "narrative"):
                data[name] = unpack_string(fh)
            other_data = fh.read(NumbersStruct.size)
            numbers = NumbersStruct.unpack(other_data)
            data["date"] = datetime.date.fromordinal(numbers[0])
            data["pilot_percent_hours_on_type"] = numbers[1]
            data["pilot_total_hours"] = numbers[2]
            data["midair"] = numbers[3]
            incident = Incident(**data)
            self[incident.report_id] = incident
        return True
```

Тело цикла `while` выполняется, пока не будут исчерпаны все данные. Сначала выполняется попытка получить идентификатор отчета. Если в результате было получено значение `None`, это означает, что достигнут конец файла и можно прервать цикл. В противном случае создается словарь `data`, в котором будут храниться сведения об одном инциденте, и предпринимается попытка получить остальные данные. Для извлечения строк используется метод `unpack_string()`, а извлечение остальных данных производится одной операцией чтения структуры `NumbersStruct`. Так как дата хранится в файле в виде целого числа, необходимо выполнить обратное его преобразование, чтобы получить дату в нормальном виде. Но для получения остальных данных достаточно всего лишь выполнить их распаковывание – здесь не требуется выполнять проверку или какие-либо преобразования, потому что выполняется попытка получить данные тех же типов, что были записаны, с использованием формата, хранящегося в структуре `NumbersStruct`.

В случае возникновения каких-либо ошибок, например, при распаковывании всех чисел, будет возбуждено исключение, которое будет обработано блоком `except`. (Мы не приводим здесь блоки `except` и `finally`, потому что они имеют ту же структуру, что и в методе `import_pickle()`, приводившемся в предыдущем подразделе.)

Ближе к концу мы, применяя подходящий синтаксис распаковывания отображения, создаем объект типа `Incident`, после чего объект сохраняется в словаре с информацией обо всех инцидентах.

Распаковывание отображений, стр. 211

Если не принимать во внимание необходимость обработки строк переменной длины, модуль `struct` существенно упрощает сохранение и загрузку данных в двоичном формате. А продемонстрированные здесь методы `pack_string()` и `unpack_string()`, предназначенные для работы со строками переменной длины, прекрасно подходят для большинства ситуаций.

Запись и синтаксический анализ текстовых файлов

Запись текста выполняется очень просто, но обратное его чтение может быть весьма проблематичным, поэтому следует очень тщательно разрабатывать структуру текста, чтобы впоследствии анализировать его было не так сложно. На рис. 7.2 показан пример записи с информацией об авиационном инциденте в текстовом формате, который мы предполагаем использовать. При записи отчетов об инцидентах в файл за каждым из них будет записываться одна пустая строка, но при анализе файла будем считать допустимыми ноль или более пустых строк между отчетами.

```
[20070927022009C]
date=2007-09-27
aircraft_id=1675B
aircraft_type=DHC-2-MK1
airport=MERLE K (MUDHOLE) SMITH
pilot_percent_hours_on_type=46.1538461538
pilot_total_hours=13000
midair=0
.NARRATIVE_START.
    ACCORDING TO THE PILOT, THE DRAG LINK FAILED DUE TO AN OVERSIZED
    TAIL WHEEL TIRE LANDING ON HARD SURFACE.
.NARRATIVE_END.
```

Рис. 7.2. *Пример записи с информацией об авиационном инциденте в текстовом формате*

Запись текста

Каждая запись с информацией об инциденте начинается с идентификатора отчета, заключенного в квадратные скобки (`[]`). Далее следуют все однострочные элементы данных, в форме *ключ=значение*. Многострочный текст комментария начинается с маркера начала (`.NARRATIVE START.`) и заканчивается маркером конца (`.NARRATIVE END.`), а чтобы гарантировать, что никакая строка комментария не будет перепутана с начальным или конечным маркером, текст между ними оформляется с отступами.

Ниже приводится программный код функции `export_text()`, за исключением блоков `except` и `finally`, поскольку они остались теми же, что и прежде, кроме обрабатываемых исключений:

```
    def export_text(self, filename):
        wrapper = textwrap.TextWrapper(initial_indent="    ",
                                       subsequent_indent="    ")
        fh = None
        try:
            fh = open(filename, "w", encoding="utf8")
            for incident in self.values():
                narrative = "\n".join(wrapper.wrap(
                                incident.narrative.strip()))
                fh.write("[{0.report_id}]\n"
                         "date={0.date!s}\n"
                         "aircraft_id={0.aircraft_id}\n"
                         "aircraft_type={0.aircraft_type}\n"
                         "airport={airport}\n"
                         "pilot_percent_hours_on_type="
                         "{0.pilot_percent_hours_on_type}\n"
                         "pilot_total_hours={0.pilot_total_hours}\n"
                         "midair={0.midair:d}\n"
                         ".NARRATIVE_START.\n{narrative}\n"
                         ".NARRATIVE_END.\n\n".format(incident,
                    airport=incident.airport.strip(),
                    narrative=narrative))
            return True
```

Символы перевода строки в тексте комментария не имеют большого значения, потому что мы можем ограничить ширину текста по своему усмотрению. Для этого можно было бы использовать функцию `textwrap.wrap()` из модуля `textwrap`, однако нам требуется не просто обернуть текст, но и добавить отступы, поэтому в самом начале метода создается объект `textwrap.TextWrap`, инициализированный отступами желаемой для нас ширины (по четыре пробела для первой и последующих строк). По умолчанию объект ограничивает ширину текста 70 символами в строке, но эту величину можно изменить, передав еще один именованный аргумент.

Мы могли бы записать этот текст как строку в тройных кавычках, но мы предпочли вручную вставлять символы перевода строки. Объект `textwrap.TextWrap` предоставляет метод `wrap()`, который принимает строку, в данном случае – текст комментария, и возвращает список строк с отступами, каждая из которых не длиннее заданной ширины текста. Затем строки из списка объединяются в единую строку, с использованием символа перевода строки в качестве разделителя. Дата инцидента хранится в объекте `datetime.date`. При записи даты методу `str.format()` предписывается использовать строковое представление даты, в результате чего он воспроизводит строку с датой в формате YYYY-MM-DD, в соответствии со стандартом ISO 8601. При записи признака `midair`, который имеет

Модуль datetime, стр. 253

Метод str.format(), стр. 100

Метод __format__(), стр. 298

тип `bool`, методу `str.format()` предписывается представить его как целое число, что в результате дает **1** – для `True` и **0** – для `False`. Вообще, использование метода `str.format()` существенно упрощает запись текста, потому что он способен автоматически обрабатывать все типы данных языка Python (включая нестандартные, при условии, что они реализуют специальные методы `__str__()` и `__format__()`).

Синтаксический анализ текста

Метод чтения и синтаксического анализа записей с информацией об авиационных инцидентах в текстовом формате – более сложный и более длинный по сравнению с методом записи. При чтении данных из файла метод может пребывать в одном из нескольких состояний. Метод может находиться в середине процедуры чтения строк комментария; он может читать строку *ключ=значение* или читать строку с идентификатором отчета в начале новой записи с информацией об инциденте. Мы рассмотрим метод `import_text_manual()`, разбив его на пять фрагментов.

```
def import_text_manual(self, filename):
    fh = None
    try:
        fh = open(filename, encoding="utf8")
        self.clear()
        data = {}
        narrative = None
```

Работа начинается с того, что файл открывается для чтения в текстовом режиме. Затем производится очистка словаря с инцидентами и создается словарь `data` для хранения данных об одном инциденте – так же, как это делалось, когда мы выполняли чтение записей с информацией об инцидентах в двоичном формате. Переменная `narrative` имеет два назначения: она используется как индикатор состояния и одновременно для хранения текста комментария для текущего инцидента. Если переменная `narrative` имеет значение `None`, это означает, что в настоящий момент не выполняется чтение комментария, но если она содержит строку (пусть даже пустую), это означает, что выполняется чтение строк комментария.

```
        for lino, line in enumerate(fh, start=1):
            line = line.rstrip()
            if not line and narrative is None:
                continue
            if narrative is not None:
                if line == ".NARRATIVE_END.":
                    data["narrative"] = textwrap.dedent(
                                                narrative).strip()
                    if len(data) != 9:
                        raise IncidentError("missing data on "
                                            "line {0}".format(lino))
                    incident = Incident(**data)
```

```
            self[incident.report_id] = incident
            data = {}
            narrative = None
        else:
            narrative += line + "\n"
```

Поскольку строки читаются по отдельности, имеется возможность следить за номером текущей строки и использовать его для вывода более информативных сообщений об ошибках, чем это возможно при чтении файлов с данными в двоичном формате. Сначала из прочитанной строки удаляются начальные и завершающие пробельные символы, и если в результате получилась пустая строка (и при этом метод не находится в процессе чтения строк комментария), то просто выполняется переход к следующей строке. Тем самым мы обеспечиваем допустимость произвольного числа пустых строк между записями об инцидентах и сохраняем пустые строки в тексте комментария.

Если переменная `narrative` не равна `None`, следовательно, выполняется чтение текста комментария. Если прочитанная строка является маркером конца комментария, это означает, что закончено чтение не только комментария, но и всех данных о текущем инциденте. В этом случае текст комментария помещается в словарь `data` (с удалением отступов вызовом функции `textwrap.dedent()`) и, если словарь содержит все девять элементов данных, создается новый объект `Incident`, который затем сохраняется в словаре. После этого выполняется подготовка к приему новой записи: словарь `data` очищается, и переменной `narrative` присваивается исходное значение. С другой стороны, если строка не является маркером конца комментария, она добавляется в конец содержимого переменной `narrative`, включая символ перевода строки, который был удален в самом начале цикла.

```
        elif (not data and line[0] == "["
                       and line[-1] == "]"):
            data["report_id"] = line[1:-1]
```

Если переменная `narrative` содержит значение `None`, следовательно, метод либо прочитал идентификатор нового отчета, либо он находится в процессе чтения каких-либо других данных. Это может быть строка с идентификатором, только если словарь `data` пуст (потому что он пуст изначально и очищается после окончания чтения каждой следующей записи) и если строка начинается с символа «[» и заканчивается символом «]». Если эти условия соблюдаются, идентификатор отчета помещается в словарь `data`. После этого условие в данной ветке `elif` не будет возвращать `True`, пока словарь data снова не будет очищен.

```
        elif "=" in line:
            key, value = line.split("=", 1)
            if key == "date":
                data[key] = datetime.datetime.strptime(value,
                                                       "%Y-%m-%d").date()
```

```
            elif key == "pilot_percent_hours_on_type":
                data[key] = float(value)
            elif key == "pilot_total_hours":
                data[key] = int(value)
            elif key == "midair":
                data[key] = bool(int(value))
            else:
                data[key] = value
        elif line == ".NARRATIVE_START.":
            narrative = ""
        else:
            raise KeyError("parsing error on line {0}".format(
                           lino))
```

Если метод находится не в процессе чтения комментария и был прочитан не идентификатор нового отчета, остаются всего три возможных варианта: был прочитан элемент *ключ=значение*, был прочитан маркер начала комментария или что-то пошло не так.

В случае, если была прочитана строка *ключ=значение*, мы разбиваем ее по первому вхождению символа «=», указав, что число разбиений не должно превышать одного, – это означает, что значение может содержать символы «=». Все данные читаются в виде строк Юникода, поэтому дата, числа и логическое значение должны быть преобразованы из строкового представления в значения соответствующих типов.

Для преобразования даты используется функция `datetime.strptime()` («string parse time» – парсинг строки со значением времени), которая принимает строку формата и возвращает объект `datetime.datetime`. Мы использовали строку формата, которая соответствует стандарту представления дат ISO 8601, а затем для извлечения объекта типа `datetime.date` из полученного объекта `datetime.datetime` использовали метод `datetime.datetime.date()`, так как нам требуется только дата, а не дата/время. Для преобразования числовых значений используются встроенные функции `float()` и `int()`. Обратите внимание, что, например, вызов `int("4.0")` возбудит исключение `ValueError`. Поэтому, если необходимо более либеральное отношение при приеме целочисленных значений, можно использовать выражение `int(float("4.0"))` или, если при этом необходимо выполнять округление, а не просто отсекать дробную часть, `round(float("4.0"))`. Получить логическое значение немножко сложнее – например, вызов `bool ("0")` вернет `True` (непустая строка в логическом контексте имеет значение `True`), поэтому сначала строку необходимо преобразовать в целое число.

Ошибочные, отсутствующие или выходящие за допустимый диапазон значения всегда будут вызывать исключение. Если любое из преобразований потерпит неудачу, будет возбуждено исключение `ValueError`. А если какое-либо из значений выйдет за допустимые пределы, будет возбуждено исключение `IncidentError` в тот момент, когда на основе прочитанных данных будет создаваться объект `Incident`.

Если строка не содержит символ «=», то проверяется – не является ли она маркером начала комментария. В этом случае в переменную `narrative` записывается пустая строка. Это означает, что при чтении всех последующих строк условное выражение в первой инструкции `if` будет давать в результате значение `True`, по меньшей мере, пока не будет прочитан маркер конца комментария.

Если ни одно из условий в ветках `if` и `elif` не было выполнено, следовательно, возникла ошибка, поэтому в заключительном предложении `else` возбуждается исключение `KeyError`, чтобы обозначить ее.

```
            return True
        except (EnvironmentError, ValueError, KeyError,
                IncidentError) as err:
            print("{0}: import error: {1}".format(
                  os.path.basename(sys.argv[0]), err))
            return False
        finally:
            if fh is not None:
                fh.close()
```

По окончании чтения всех строк вызывающей программе возвращается значение `True`, если не было возбуждено исключение, – в этом случае блок `except` перехватит исключение, выведет для пользователя сообщение об ошибке и вернет `False`. И в заключение, независимо от происходящего, файл будет закрыт.

Синтаксический анализ текста с помощью регулярных выражений

Читателям, не знакомым с регулярными выражениями, рекомендуется прочитать главу 12, прежде чем приступать к чтению этого раздела, или сразу перейти к чтению следующего раздела (стр. 364) и вернуться сюда позднее.

Использование регулярных выражений для разбора текста часто дает более короткий программный код по сравнению с тем, где все действия по разбору выполняются вручную, как это делалось в предыдущем подразделе, но в нем сложнее реализовать вывод ясных сообщений об ошибках. Ниже приводится программный код метода `import_text_regex()`, который мы рассмотрим в два приема. Сначала мы обсудим регулярные выражения, а затем реализацию синтаксического анализа, но опустим блоки `except` и `finally`, поскольку в них не появилось ничего нового для нас.

```
def import_text_regex(self, filename):
    incident_re = re.compile(
                    r"\[(?P<id>[^]]+)\](?P<keyvalues>.+?)"
                    r"^\.NARRATIVE_START\.$(?P<narrative>.*?)"
                    r"^\.NARRATIVE_END\.$",
                    re.DOTALL|re.MULTILINE)
```

```
key_value_re = re.compile(r"^\s*(?P<key>[^=]+)\s*=\s*"
                          r"(?P<value>.+)\s*$", re.MULTILINE)
```

«Сырые» строки, стр. 85

Регулярные выражения записаны как «сырые» (raw) строки. Это устраняет необходимость дублировать каждый символ обратного слеша (вместо `\` записывать `\\`); например, если не использовать «сырые» строки, второе регулярное выражение пришлось бы записать как: `"^\\s*(?P<key>[^=]+)  \\s*=\\s*(?P<value>.+)\\s*$"`. В этой книге для записи регулярных выражений мы всегда будем использовать «сырые» строки.

Первое регулярное выражение, `incident_re`, используется для захвата всей записи с информацией об инциденте. При таком подходе любой посторонний текст *между* записями останется незамеченным. Данное регулярное выражение в действительности состоит из двух частей. Первая часть `\[(?P<id>[^]]+)\]` соответствует символу «[», затем соответствует, с захватом в группу `id`, произвольному числу символов, отличных от «]», затем соответствует символу «]» (что дает нам идентификатор отчета) и затем соответствует любому числу (но не менее одного) любых символов (включая символы перевода строки, благодаря флагу `re.DOTALL`), захватывая их в группу `keyvalues`. Символы, включенные в группу `keyvalues`, являются необходимым минимумом, чтобы перейти ко второй части регулярного выражения.

Вторая часть первого регулярного выражения: `^\.NARRATIVE_START\.$(?P<narrative>.*?)^\.NARRATIVE_END\.$`. Она соответствует точному тексту `.NARRATIVE_START.`, затем произвольному числу символов, которые захватываются в группу `narrative`, и затем точному тексту `.NARRATIVE_END.` в конце записи с информацией об инциденте. Флаг `re.MULTILINE` означает, что в данном регулярном выражении символ `^` соответствует началу каждой строки в файле (а не началу всей строки), а символ `$` соответствует концу каждой строки в файле (а не концу всей строки), поэтому соответствие маркерам начала и конца комментария будет обнаруживаться, только если они находятся в начале строки.

Второе регулярное выражение `key_value_re` используется для захвата строк *ключ=значение* и соответствует началу каждой строки в заданном тексте, произвольному (в том числе и нулевое) числу последующих пробельных символов, за которыми следуют символы, отличные от символа «=», захватываемые в группу `key`, последующему символу «=» и всем остальным символам в строке текста (исключая начальные и завершающие пробельные символы), захватываемым в группу `value`.

Основная логика синтаксического анализа файла осталась той же, что использовалась для анализа текста вручную и описана в предыдущем подразделе, только на этот раз сама запись и информация об инциденте извлекаются не посредством построчного чтения содержимого файла, а с помощью регулярных выражений.

```
        fh = None
        try:
            fh = open(filename, encoding="utf8")
            self.clear()
            for incident_match in incident_re.finditer(fh.read()):
                data = {}
                data["report_id"] = incident_match.group("id")
                data["narrative"] = textwrap.dedent(
                            incident_match.group("narrative")).strip()
                keyvalues = incident_match.group("keyvalues")
                for match in key_value_re.finditer(keyvalues):
                    data[match.group("key")] = match.group("value")
                data["date"] = datetime.datetime.strptime(
                                    data["date"], "%Y-%m-%d").date()
                data["pilot_percent_hours_on_type"] = (
                        float(data["pilot_percent_hours_on_type"]))
                data["pilot_total_hours"] = int(
                        data["pilot_total_hours"])
                data["midair"] = bool(int(data["midair"]))
                if len(data) != 9:
                    raise IncidentError("missing data")
                incident = Incident(**data)
                self[incident.report_id] = incident
            return True
```

Метод `re.finditer()` возвращает итератор, который поочередно возвращает неперекрывающиеся совпадения. В начале цикла создается словарь `data` для хранения информации об инциденте, как и раньше, но на этот раз идентификатор отчета и текст комментария извлекаются непосредственно из найденного соответствия регулярному выражению `incident_re`. Затем из группы `keyvalues` извлекаются сразу все строки *ключ=значение* и к ним применяется метод `re.finditer()` регулярного выражения `key_value_re`, чтобы выполнить обход отдельных строк *ключ=значение*. Каждая найденная пара (ключ, значение) помещается в словарь `data`, поэтому все значения сохраняются в виде строк. Далее те значения, которые не должны быть строками, замещаются значениями соответствующих типов, для чего выполняются те же преобразования строк, что применялись при разборе текста вручную.

Мы добавили проверку, чтобы убедиться, что словарь `data` содержит ровно девять элементов данных, потому что в случае повреждения записи с информацией об инциденте итератор `key_value.finditer()` может отыскать слишком много или слишком мало строк *ключ=значение*. Оканчивается метод точно так же, как и раньше, – создается новый объект `Incident`, который затем помещается в словарь инцидентов, после чего вызывающей программе возвращается значение `True`. Если что-то пойдет не так, блок `except` выведет соответствующее сообщение об ошибке и вернет `False`, а блок `finally` закроет файл.

Одной из особенностей, которые делают программный код, анализирующий текст вручную или с применением регулярных выражений, таким коротким и таким простым, является механизм обработки исключений языка Python. Программный код не проверяет результаты преобразований строк в даты, числа или логические значения, и в нем отсутствуют проверки попадания значений в допустимые границы (это делает класс `Incident`). Если какая-либо из этих операций завершится неудачей, интерпретатор возбудит исключение, и мы предусматриваем обработку всех исключений в одном месте, в конце методов. Другое преимущество использования механизма исключений перед явной проверкой на наличие ошибок состоит в хорошей масштабируемости программного кода – даже в случае изменения формата записи и увеличения количества элементов данных программный код, выполняющий обработку ошибок, не будет увеличиваться в объеме.

Запись и синтаксический анализ файлов XML

Некоторые программы используют файлы формата XML для хранения всех обрабатываемых данных, другие обеспечивают только возможность импорта/экспорта в формате XML. Способность импортировать и экспортировать данные в формате XML не будет лишней, и поддержку этого формата всегда стоит предусматривать, даже если основным форматом, с которым работает программа, является текстовый или двоичный формат.

Язык Python предоставляет три способа записи файлов в формате XML: вручную, посредством создания дерева элементов и использования его метода `write()`, а также посредством создания DOM и использования его метода `write()`. Для чтения и анализа файлов XML используется четыре способа: чтение и разбор файла XML вручную (не рекомендуется и не рассматривается в этой книге, поскольку может оказаться чрезвычайно сложно корректно обработать некоторые из наиболее туманных и расширенных возможностей) или с использованием парсеров `ElementTree`, DOM или SAX.

Формат XML записи с информацией об авиационном инциденте приводится на рис. 7.3. В этом разделе будет показано, как выполнять запись в этом формате вручную и как выполнять запись с помощью дерева элементов и DOM, а также как читать и анализировать этот формат с помощью парсеров `ElementTree`, DOM и SAX. Если вас не интересует вопрос выбора способа чтения или записи файлов в формате XML, вы можете просто прочитать подраздел «Деревья элементов», следующий ниже, и затем перейти к заключительному разделу главы «Произвольный доступ к двоичным данным в файлах» (стр. 376).

```
<?xml version="1.0" encoding="UTF-8"?>
<incidents>
<incident report_id="20070222008099G" date="2007-02-22"
    aircraft_id="80342" aircraft_type="CE-172-M"
    pilot_percent_hours_on_type="9.09090909091"
    pilot_total_hours="440" midair="0">
<airport>BOWERMAN</airport>
<narrative>
ON A GO-AROUND FROM A NIGHT CROSSWIND LANDING ATTEMPT THE AIRCRAFT HIT
A RUNWAY EDGE LIGHT DAMAGING ONE PROPELLER.
</narrative>
</incident>
<incident>
    ...
</incident>
     :
</incidents>
```

Рис. 7.3. *Пример записи с информацией об авиационном инциденте в формате XML*

Деревья элементов

Запись данных с использованием дерева элементов выполняется в два этапа: сначала должно быть создано дерево элементов, представляющее данные, и затем дерево должно быть записано в файл. Некоторые программы могут использовать дерево элементов в качестве основной структуры представления своих данных – в этом случае дерево элементов уже имеется изначально и остается лишь записать его в файл. Мы рассмотрим метод `export_xml_etree()`, разделив его на две части:

```
def export_xml_etree(self, filename):
    root = xml.etree.ElementTree.Element("incidents")
    for incident in self.values():
        element = xml.etree.ElementTree.Element("incident",
                report_id=incident.report_id,
                date=incident.date.isoformat(),
                aircraft_id=incident.aircraft_id,
                aircraft_type=incident.aircraft_type,
                pilot_percent_hours_on_type=str(
                        incident.pilot_percent_hours_on_type),
                pilot_total_hours=str(incident.pilot_total_hours),
                midair=str(int(incident.midair)))
        airport = xml.etree.ElementTree.SubElement(element,
                                                   "airport")
        airport.text = incident.airport.strip()
        narrative = xml.etree.ElementTree.SubElement(element,
```

```
                                                    "narrative")
            narrative.text = incident.narrative.strip()
            root.append(element)
        tree = xml.etree.ElementTree.ElementTree(root)
```

Метод начинается с создания корневого элемента (<incidents>). Затем в цикле выполняются итерации по всем записям с информацией об инцидентах. Для каждой записи создается свой элемент (<incident>), в котором будут храниться данные об инциденте, а именованные аргументы определяют атрибуты элемента. Все атрибуты должны иметь текстовый формат, поэтому даты, числа и логические значения преобразуются соответствующим образом. Нам не нужно беспокоиться об экранировании символов «&», «<» и «>» (или о кавычках в значениях атрибутов), так как модуль парсера дерева элементов (а также модули парсеров DOM и SAX) делает это автоматически.

Каждый элемент <incident> имеет два подэлемента, один хранит название аэропорта, а второй – текст комментария. При создании подэлемента мы должны указать родительский элемент и имя тега. Для хранения текста используется атрибут text элемента, доступный для чтения и записи.

После создания элемента <incident> со всеми его атрибутами и подэлементами <airport> и <narrative> мы добавляем его в корневой элемент (<incidents>). В результате у нас получается иерархия элементов, содержащих все записи с информацией об инцидентах, которая затем тривиально просто преобразуется в дерево элементов.

```
        try:
            tree.write(filename, "UTF-8")
        except EnvironmentError as err:
            print("{0}: import error: {1}".format(
                  os.path.basename(sys.argv[0]), err))
            return False
        return True
```

Запись целого дерева элементов с данными в формате XML выполняется простым вызовом его метода, выполняющим запись в указанный файл с использованием указанной кодировки символов.

До сих пор практически всякий раз, когда мы указывали кодировку, мы использовали строку "utf8". Она является вполне допустимой для встроенной функции open(), которая может принимать широкий диапазон кодировок с различными версиями их названий, такими как «UTF-8», «UTF8», «utf-8» и «utf8». Но в файлах XML могут использоваться только официальные названия кодировок, по этой причине название "utf8" нельзя использовать, и мы используем название "UTF-8".[1]

[1] Дополнительную информацию о названиях кодировок вы найдете на сайтах *www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl* и *www.iana.org/assignments/character-sets*.

Чтение файла XML с использованием дерева элементов выполняется ничуть не сложнее, чем запись. Запись также выполняется в два этапа: на первом этапе выполняется чтение и анализ содержимого файла XML, а затем производится обход дерева элементов и заполнение словаря с информацией об инцидентах. Как и прежде, второй этап не является обязательным, если для хранения данных в памяти используется само дерево элементов. Ниже приводится программный код метода `import_xml_etree()`, разбитый на две части:

```
def import_xml_etree(self, filename):
    try:
        tree = xml.etree.ElementTree.parse(filename)
    except (EnvironmentError,
            xml.parsers.expat.ExpatError) as err:
        print("{0}: import error: {1}".format(
              os.path.basename(sys.argv[0]), err))
        return False
```

По умолчанию парсер дерева элементов использует парсер `expat`, именно поэтому мы должны быть готовы перехватывать исключения парсера `expat`.

```
    self.clear()
    for element in tree.findall("incident"):
        try:
            data = {}
            for attribute in ("report_id", "date", "aircraft_id",
                    "aircraft_type",
                    "pilot_percent_hours_on_type",
                    "pilot_total_hours", "midair"):
                data[attribute] = element.get(attribute)
            data["date"] = datetime.datetime.strptime(
                                data["date"], "%Y-%m-%d").date()
            data["pilot_percent_hours_on_type"] = (
                    float(data["pilot_percent_hours_on_type"]))
            data["pilot_total_hours"] = int(
                    data["pilot_total_hours"])
            data["midair"] = bool(int(data["midair"]))
            data["airport"] = element.find("airport").text.strip()
            narrative = element.find("narrative").text
            data["narrative"] = (narrative.strip()
                                 if narrative is not None else "")
            incident = Incident(**data)
            self[incident.report_id] = incident
        except (ValueError, LookupError, IncidentError) as err:
            print("{0}: import error: {1}".format(
                  os.path.basename(sys.argv[0]), err))
            return False
    return True
```

Получив дерево элементов, можно приступать к выполнению итераций по всем элементам <incident> с использованием метода xml.etree.ElementTree.findall(). Информация о каждом инциденте возвращается в виде объекта xml.etree.Element. Здесь используется та же методика обработки атрибутов элемента, что и в разделе с описанием метода import_text_regex(), – сначала все значения сохраняются в словаре data, а затем выполняется преобразование таких данных, как даты, числа и логические значения, в соответствующие типы данных. Для извлечения элементов <airport> и <narrative> и чтения их атрибутов text используется метод xml.etree.Element.find(). Если текстовый элемент не содержит текст, его атрибут text будет иметь значение None, поэтому нам необходимо учитывать это обстоятельство при чтении текстового элемента комментария, который может оказаться пустым. Во всех случаях возвращаемые значения атрибутов и текст не содержат экранированных последовательностей XML, потому что они автоматически преобразуются в соответствующие символы.

При использовании парсеров XML для обработки данных об авиационных инцидентах, как и любых других парсеров, будут возбуждаться исключения: в случае отсутствия элементов с названием аэропорта или с комментарием, в случае ошибки при выполнении какого-либо преобразования или при выходе любого числового значения за границы допустимого диапазона – этим гарантируется, что ошибочные данные будут приводить к прекращению анализа файла и к выводу сообщения об ошибке. Программный код в конце метода, создающий и сохраняющий инциденты, а также программный код обработки исключений остался тем же, что мы уже видели ранее.

DOM (Document Object Model – объектная модель документа)

Модель DOM – это стандартный API представления и манипулирования документами XML в памяти. Программный код создания и записи DOM в файл и анализа файла XML с применением модели DOM по своей структуре близко напоминает программный код, работающий с деревом элементов, только немного длиннее.

Мы рассмотрим метод export_xml_dom(), разделив его на две части. Работа этого метода делится на два этапа: сначала создается дерево DOM, отражающее данные об инцидентах, а потом оно записывается в файл. Как и в случае с деревом элементов, существуют программы, которые используют дерево DOM в качестве основной структуры для хранения своих данных, и в этой ситуации существующие данные просто записываются в файл, минуя первый этап.

```
def export_xml_dom(self, filename):
    dom = xml.dom.minidom.getDOMImplementation()
    tree = dom.createDocument(None, "incidents", None)
```

```
root = tree.documentElement
for incident in self.values():
    element = tree.createElement("incident")
    for attribute, value in (
            ("report_id", incident.report_id),
            ("date", incident.date.isoformat()),
            ("aircraft_id", incident.aircraft_id),
            ("aircraft_type", incident.aircraft_type),
            ("pilot_percent_hours_on_type",
             str(incident.pilot_percent_hours_on_type)),
            ("pilot_total_hours",
             str(incident.pilot_total_hours)),
            ("midair", str(int(incident.midair)))):
        element.setAttribute(attribute, value)
    for name, text in (("airport", incident.airport),
                       ("narrative", incident.narrative)):
        text_element = tree.createTextNode(text)
        name_element = tree.createElement(name)
        name_element.appendChild(text_element)
        element.appendChild(name_element)
root.appendChild(element)
```

Метод начинается с того, что получает реализацию DOM. По умолчанию реализация предоставляется парсером `expat`. Модуль `xml.dom.minidom` предоставляет более простую и более легковесную реализацию DOM по сравнению с той, что предоставляется модулем `xml.dom`, хотя и использует объекты, которые определяются в модуле `xml.dom`. После получения реализации DOM можно приступать к созданию документа. Первый аргумент метода `xml.dom.DOMImplementation.createDocument()` — это URI пространства имен, но в нашем случае он не требуется, поэтому мы передаем значение `None`. Второй аргумент — это квалифицированное имя (имя тега корневого элемента) и третий аргумент — это тип документа, в нем мы также передаем значение `None`, так как у нас отсутствует тип документа. Создав дерево, представляющее документ, мы получаем корневой элемент и в цикле заполняем его информацией об инцидентах.

Для каждого инцидента создается элемент `<incident>`, а для создания каждого атрибута этого элемента вызывается метод `setAttribute()`, которому передаются имя атрибута и значение. Так же как и в случае с деревом элементов, нам не нужно беспокоиться об экранировании символов «&», «<» и «>» (так же, как и о кавычках в значениях атрибутов). Для текстовых данных с названием аэропорта и комментариями необходимо создать текстовые элементы, которые будут хранить сам текст, и обычные элементы (с соответствующим именем тега), которые будут играть роль родительских элементов, после чего обычные элементы (и содержащиеся в нем текстовые элементы) добавляются в текущий элемент `<incident>`. Как только элемент с информацией об инциденте будет заполнен, он добавляется в корневой элемент.

```
        fh = None
        try:
            fh = open(filename, "w", encoding="utf8")
            tree.writexml(fh, encoding="UTF-8")
            return True
```

Кодировки символов в файлах XML, стр. 366

Мы опустили блоки `except` и `finally`, так как они ничем не отличаются от тех, что мы уже видели. Этот фрагмент наглядно демонстрирует различия между строками с именами кодировок, используемыми при работе со встроенной функцией `open()`, и строками с именами кодировок, используемыми для файлов XML, о чем уже говорилось выше.

Импортирование документа в виде дерева DOM напоминает импортирование в дерево элементов, но, как и при экспортировании, для реализации импортирования требуется больший объем программного кода. Мы рассмотрим функцию `import_xml_dom()`, разделив ее на три части, и начнем со строки с инструкцией `def` и определения вложенной функции `get_text()`.

```
    def import_xml_dom(self, filename):

        def get_text(node_list):
            text = []
            for node in node_list:
                if node.nodeType == node.TEXT_NODE:
                    text.append(node.data)
            return "".join(text).strip()
```

Функция `get_text()` выполняет обход списка узлов (то есть дочерних узлов заданного узла) и из каждого текстового узла извлекает его текст и добавляет в конец списка текстов. В конце функция возвращает весь извлеченный текст, объединенный в одну строку, попутно удалив пробельные символы в начале и в конце строки.

```
            try:
                dom = xml.dom.minidom.parse(filename)
            except (EnvironmentError,
                    xml.parsers.expat.ExpatError) as err:
                print("{0}: import error: {1}".format(
                      os.path.basename(sys.argv[0]), err))
                return False
```

Преобразование содержимого файла XML в дерево DOM выполняется достаточно просто, потому что модуль всю основную работу берет на себя, но мы должны быть готовы обработать ошибки парсера `expat`, потому что этот парсер XML, как и в случае с деревом элементов, по умолчанию используется классами DOM.

```
            self.clear()
            for element in dom.getElementsByTagName("incident"):
```

```
            try:
                data = {}
                for attribute in ("report_id", "date", "aircraft_id",
                        "aircraft_type",
                        "pilot_percent_hours_on_type",
                        "pilot_total_hours", "midair"):
                    data[attribute] = element.getAttribute(attribute)
                data["date"] = datetime.datetime.strptime(
                                    data["date"], "%Y-%m-%d").date()
                data["pilot_percent_hours_on_type"] = (
                        float(data["pilot_percent_hours_on_type"]))
                data["pilot_total_hours"] = int(
                        data["pilot_total_hours"])
                data["midair"] = bool(int(data["midair"]))
                airport = element.getElementsByTagName("airport")[0]
                data["airport"] = get_text(airport.childNodes)
                narrative = element.getElementsByTagName(
                                                "narrative")[0]
                data["narrative"] = get_text(narrative.childNodes)
                incident = Incident(**data)
                self[incident.report_id] = incident
            except (ValueError, LookupError, IncidentError) as err:
                print("{0}: import error: {1}".format(
                      os.path.basename(sys.argv[0]), err))
                return False
        return True
```

Локальные функции, стр. 409

После создания дерева DOM производится очистка словаря с инцидентами и выполняются итерации по всем тегам `<incident>`. Из каждого тега инцидента извлекаются его атрибуты, и затем даты, числа и логические значения преобразовываются в соответствующие типы данных тем же способом, который применялся при работе с деревом элементов. Единственное существенное отличие между деревом DOM и деревом элементов состоит в том, как обрабатываются текстовые узлы. Сначала с помощью метода `xml.dom.Element.getElementsByTagName()` извлекаются дочерние элементы с заданными именами тегов, в данном случае это `<airport>` и `<narrative>`, которые, как мы знаем, всегда присутствуют в единственном экземпляре, поэтому мы извлекаем первый (и только первый) дочерний элемент каждого из этих двух типов. Затем с помощью вложенной функции выполняются итерации по всем дочерним узлам этих тегов, чтобы извлечь текст, находящийся в них.

Как обычно, если возникают какие-либо ошибки, соответствующие исключения перехватываются, для пользователя выводится сообщение и вызывающей программе возвращается `False`.

Различия между подходами с использованием DOM и дерева элементов невелики, и, поскольку в обоих случаях в конечном итоге используется парсер `expat`, оба они обладают неплохой производительностью.

Запись файла XML вручную

Запись в файл уже существующего дерева элементов или дерева DOM может быть реализована единственным вызовом метода. Но если данные еще не представлены в какой-либо из этих форм, то сначала будет необходимо создать дерево элементов или дерево DOM, хотя иногда может оказаться гораздо удобнее просто записать данные в файл, минуя этот этап.

При создании файлов XML, чтобы получить правильно оформленный документ XML, необходимо гарантировать корректное экранирование служебных символов в тексте и в значениях атрибутов. Ниже приводится программный код метода `export_xml_manual()`, выполняющий запись данных об инцидентах в файл XML:

```
def export_xml_manual(self, filename):
    fh = None
    try:
        fh = open(filename, "w", encoding="utf8")
        fh.write('<?xml version="1.0" encoding="UTF-8"?>\n')
        fh.write("<incidents>\n")
        for incident in self.values():
            fh.write('<incident report_id={report_id} '
                     'date="{0.date!s}" '
                     'aircraft_id={aircraft_id} '
                     'aircraft_type={aircraft_type} '
                     'pilot_percent_hours_on_type='
                     '"{0.pilot_percent_hours_on_type}" '
                     'pilot_total_hours="{0.pilot_total_hours}" '
                     'midair="{0.midair:d}">\n'
                     '<airport>{airport}</airport>\n'
                     '<narrative>\n{narrative}\n</narrative>\n'
                     '</incident>\n'.format(incident,
                report_id=xml.sax.saxutils.quoteattr(
                                    incident.report_id),
                aircraft_id=xml.sax.saxutils.quoteattr(
                                    incident.aircraft_id),
                aircraft_type=xml.sax.saxutils.quoteattr(
                                    incident.aircraft_type),
                airport=xml.sax.saxutils.escape(incident.airport),
                narrative="\n".join(textwrap.wrap(
                        xml.sax.saxutils.escape(
                            incident.narrative.strip()), 70))))
            fh.write("</incidents>\n")
            return True
```

Как и прежде в этой главе, мы опустили блоки `except` и `finally`.

При записи в файл используется кодировка UTF-8, и ее необходимо указать в вызове встроенной функции `open()`. Строго говоря, кодировку можно и не указывать в объявлении `<?xml?>` потому, что кодировка UTF-8 используется по умолчанию, но мы предпочитаем делать это явно. Мы решили заключать значения атрибутов в кавычки, поэтому при добавлении данных об инциденте для обозначения строк в программном коде используются апострофы, благодаря чему отпала необходимость экранировать кавычки.

Функция `sax.saxutils.quoteattr()` напоминает по своему действию функцию `sax.saxutils.escape()`, используемую для обработки текста XML, – тем, что она корректно экранирует символы «&», «<» и «>». Кроме того, она экранирует кавычки (если это необходимо) и возвращает готовую к использованию строку, уже заключенную в кавычки. По этой причине нам не потребовалось окружать кавычками идентификатор отчета и другие строковые значения атрибутов.

Символы перевода строки, которые мы вставляем, и выравнивание текста комментария – это исключительно косметическое прихорашивание. Сделано это только для того, чтобы содержимое файла проще было читать людям, поэтому их легко можно просто опустить.

Запись данных в формате HTML мало чем отличается от записи данных в формате XML. Программа *convert-incidents.py* включает в себя функцию `export_html()` – в качестве простого примера такой возможности, однако мы не будем рассматривать ее, потому что в ней нет ничего нового, что действительно стоило бы показать.

Синтаксический анализ файлов XML с помощью SAX (Simple API for XML – упрощенный API для XML)

В отличие от дерева элементов и DOM, которые формируют документ XML в памяти целиком, парсеры SAX используют принцип последовательной обработки, реализация которого потенциально обладает более высокой скоростью работы и предъявляет более низкие требования к объему памяти. Однако преимущество в скорости можно не учитывать, так как реализации деревьев элементов и DOM используют быстрый парсер `expat`.

Парсеры SAX, когда встречают начальные теги, конечные теги и другие элементы XML, извещают об этом посредством «событий парсинга». Чтобы иметь возможность обрабатывать интересующие нас события, мы должны создать соответствующий класс обработчика и реализовать в нем предопределенные методы, которые будут вызываться по соответствующим событиям. Наиболее часто в программах реализуется обработчик содержимого, хотя, когда возникает необходимость в более полном управлении процессом парсинга, можно предусмотреть и реализацию обработчиков ошибок других обработчиков.

Ниже приводится полный программный код метода `import_xml_sax()`. Он получился очень коротким благодаря тому, что основная работа выполняется классом `IncidentSaxHandler`:

```
def import_xml_sax(self, filename):
    fh = None
    try:
        handler = IncidentSaxHandler(self)
        parser = xml.sax.make_parser()
        parser.setContentHandler(handler)
        parser.parse(filename)
        return True
    except (EnvironmentError, ValueError, IncidentError,
            xml.sax.SAXParseException) as err:
        print("{0}: import error: {1}".format(
              os.path.basename(sys.argv[0]), err))
        return False
```

Мы создали один обработчик, который будет использоваться нами, затем создали экземпляр парсера SAX и передали ему в качестве обработчика содержимого обработчик, созданный непосредственно перед этим. После этого мы передали методу `parse()` парсера имя файла и вернули `True`, если в ходе анализа файла не возникло никаких ошибок.

Методу инициализации обработчика класса `IncidentSaxHandler` был передан объект `self` (то есть объект класса `IncidentCollection`, являющегося подклассом `dict`). Обработчик удаляет всю прежнюю информацию об инцидентах и затем по мере разбора файла наполняет его новыми данными об инцидентах. По завершении процесса парсинга словарь будет содержать все прочитанные записи об инцидентах.

```
class IncidentSaxHandler(xml.sax.handler.ContentHandler):

    def __init__(self, incidents):
        super().__init__()
        self.__data = {}
        self.__text = ""
        self.__incidents = incidents
        self.__incidents.clear()
```

Наш собственный класс обработчика должен наследовать соответствующий базовый класс. Тем самым гарантируется, что при отсутствии реализации некоторых методов (просто потому, что некоторые события парсинга нас не интересуют) будут вызываться методы базового класса, которые не делают ничего опасного.

В самом начале вызывается метод инициализации базового класса. Вообще, это желательно делать в любых подклассах, хотя для прямых наследников класса `object` в этом нет необходимости (но и нет никакой опасности). Словарь `self.__data` используется для хранения информации об инциденте, строка `self.__text` используется для хранения текста с названием аэропорта или комментария – в зависимости от того,

какой элемент данных читается, и словарь `self.__incidents` является ссылкой на словарь `IncidentCollection`, который будет дополняться обработчиком напрямую. (В качестве альтернативы можно было бы создать внутри обработчика независимый словарь и копировать его в конце вызовом методов `dict.clear()` и `dict.update()`.)

```
def startElement(self, name, attributes):
    if name == "incident":
        self.__data = {}
        for key, value in attributes.items():
            if key == "date":
                self.__data[key] = datetime.datetime.strptime(
                                        value, "%Y-%m-%d").date()
            elif key == "pilot_percent_hours_on_type":
                self.__data[key] = float(value)
            elif key == "pilot_total_hours":
                self.__data[key] = int(value)
            elif key == "midair":
                self.__data[key] = bool(int(value))
            else:
                self.__data[key] = value
    self.__text = ""
```

Всякий раз, когда парсер встречает открывающий тег и его атрибуты, он вызывает метод `xml.sax.handler.ContentHandler.startElement()`, которому передает имя тега и его атрибуты. В файле XML, содержащем информацию об авиационных инцидентах, имеются следующие открывающие теги: `<incidents>`, который мы просто игнорируем; `<incident>`, атрибуты которого помещаются в словарь `self.__data`; а также `<airport>` и `<narrative>`, которые мы тоже игнорируем. Всегда, когда встречается открывающий тег, мы очищаем строку `self.__text`, потому что в формате файла XML с информацией об авиационных инцидентах отсутствуют вложенные текстовые теги.

Мы не предусматриваем обработку исключений в классе `IncidentSaxHandler`. В случае появления исключения оно будет передано вызывающему методу, в данном случае – методу `import_xml_sax()`, который перехватит его и выведет соответствующее сообщение об ошибке.

```
def endElement(self, name):
    if name == "incident":
        if len(self.__data) != 9:
            raise IncidentError("missing data")
        incident = Incident(**self.__data)
        self.__incidents[incident.report_id] = incident
    elif name in frozenset({"airport", "narrative"}):
        self.__data[name] = self.__text.strip()
    self.__text = ""
```

Когда парсер встречает закрывающий тег, он вызывает метод `xml.sax.handler.ContentHandler.EndElement()`. Если был достигнут конец записи

об инциденте, все необходимые данные уже должны быть собраны, поэтому остается только создать новый объект `Incident` и добавить его в словарь с инцидентами. Если был обнаружен закрывающий тег текстового элемента, в словарь `self.__data` добавляется новый элемент с текстом, извлеченным к данному моменту. В конце метод очищает строку `self.__text`, подготавливая ее к дальнейшему использованию. (Строго говоря, ее можно и не очищать, так как она очищается при обнаружении открывающего тега, но очистка может потребоваться при работе с некоторыми другими форматами XML, например, где имеются вложенные теги.)

```
def characters(self, text):
    self.__text += text
```

Когда парсер SAX встречает текст, он вызывает метод `xml.sax.handler.ContentHandler.characters()`. Нет никакой гарантии, что этот метод будет вызван один раз для всего текста – текст может передаваться частями. По этой причине метод просто накапливает текст, а запись текста в словарь выполняется, только когда будет встречен соответствующий закрывающий тег. (Более эффективно было бы сделать переменную `self.__text` списком, тело этого метода – вызовом метода `self.__text.append(text)` и внести соответствующие изменения в другие методы.)

Реализация с использованием SAX API существенно отличается от реализации с использованием дерева элементов или DOM, но она намного эффективнее. Мы можем реализовать другие обработчики и переопределить другие методы в обработчике содержимого, чтобы получить более полный контроль над процессом парсинга. Парсер SAX не поддерживает возможность создания представления документа XML, что делает его идеальным инструментом для чтения данных в формате XML в наши собственные коллекции, но это также означает, что при использовании SAX в памяти нет никакого «документа», готового к записи в файл в формате XML, поэтому запись должна выполняться с использованием одного из подходов, рассматривавшихся выше в этом разделе.

Произвольный доступ к двоичным данным в файлах

В предыдущих разделах рассматривалась методика, когда все данные программы целиком читаются в память, обрабатываются и затем целиком записываются в файл. В современных компьютерах так много оперативной памяти, что эта методика имеет полное право на существование даже в случае больших объемов данных. Однако в некоторых ситуациях более предпочтительной может оказаться методика, когда данные полностью хранятся на диске, в память небольшими порциями читаются только необходимые данные, а на диск записываются только изменения. Подход, основанный на произвольном доступе к данным на

диске, легко реализовать при использовании базы данных типа ключ-значение («DBM») или полноценной базы данных SQL – оба варианта будут рассматриваться в главе 11, а в этом разделе будет показано, как вручную реализовать произвольный доступ к данным в файлах.

Для начала будет представлен класс `BinaryRecordFile.BinaryRecordFile`. Экземпляры этого класса являются универсальным представлением двоичных файлов, доступных для чтения и записи, состоящих из последовательности записей фиксированной длины. Затем, чтобы продемонстрировать, как использовать двоичные файлы с произвольным доступом, будет рассмотрен класс `BikeStock.BikeStock`, хранящий коллекцию объектов `BikeStock.Bike` в виде записей в объекте `BinaryRecordFile.BinaryRecordFile`.

Универсальный класс BinaryRecordFile

Своим прикладным интерфейсом класс `BinaryRecordFile.BinaryRecordFile` напоминает список, так как он обеспечивает возможность получения/добавления/удаления записи по заданному номеру позиции. Когда запись удаляется, она просто помечается как «удаленная», благодаря этому исчезает необходимость перемещать все последующие записи, чтобы заполнить промежуток; что также означает, что после удаления все первоначальные индексы остаются допустимыми. Другое преимущество такого подхода состоит в том, что запись легко может быть восстановлена, достаточно лишь убрать метку. Однако при таком подходе, удаляя записи, мы не можем экономить дисковое пространство. Эта проблема будет решаться за счет методов «уплотнения» файла, которые будут ликвидировать удаленные записи (и соответственно будут изменяться номера позиций записей).

Прежде чем приступить к рассмотрению реализации, взглянем на типичный пример использования:

```
Contact = struct.Struct("<15si")
contacts = BinaryRecordFile.BinaryRecordFile(filename, Contact.size)
```

Здесь создается структура (с обратным порядком следования байтов, 15-байтовая строка байтов и 4-байтовое целое число со знаком), которая будет представлять записи. Затем создается экземпляр класса `BinaryRecordFile.BinaryRecordFile`, которому передается имя файла и размер записи, соответствующий размеру используемой структуры. Если файл уже существует, его содержимое при открытии остается на месте; в противном случае создается новый файл; и в любом случае файл открывается для чтения/записи в двоичном режиме.

```
contacts[4] = Contact.pack("Abe Baker".encode("utf8"), 762)
contacts[5] = Contact.pack("Cindy Dove".encode("utf8"), 987)
```

Мы можем воспринимать файл как список, и использовать оператор доступа к элементам (`[]`). Здесь выполняется присваивание двух бай-

товых строк (объектов bytes, каждый из которых содержит строку и целое число) двум записям, с использованием номеров их позиций в файле. Эти операции присваивания перезапишут прежнее содержимое, а если файл содержит менее шести записей, будут созданы новые записи, каждая из которых будет заполнена байтами 0x00.

```
contact_data = Contact.unpack(contacts[5])
contact_data[0].decode("utf8").rstrip(chr(0)) # вернет: 'Cindy Dove'
```

Поскольку строка «Cindy Dove» содержит менее 15 символов UTF-8, при упаковывании в конец ее будут добавлены байты 0x00. Поэтому при извлечении записи contact_data будет содержать кортеж из двух элементов (b'Cindy Dove\x00\x00\x00\x00\x00', 987). Чтобы получить имя, необходимо декодировать последовательность байтов в кодировке UTF-8 для получения строки Юникода и потом удалить завершающие байты 0x00.

Теперь, когда мы мельком увидели класс в действии, можно приступать к рассмотрению программного кода. Определение класса BinaryRecordFile.BinaryRecordFile находится в файле *BinaryRecordFile.py*. Вслед за обычными предварительными сведениями следуют два частных определения значений байтов:

```
_DELETED = b"\x01"
_OKAY = b"\x02"
```

Каждая запись начинается с байта «состояния», который может иметь одно из двух значений: _DELETED или _OKAY (или b"\x00" в случае пустой записи).

Ниже приводятся строка с инструкцией class и программный код метода инициализации:

```
class BinaryRecordFile:

    def __init__(self, filename, record_size, auto_flush=True):
        self.__record_size = record_size + 1
        mode = "w+b" if not os.path.exists(filename) else "r+b"
        self.__fh = open(filename, mode)
        self.auto_flush = auto_flush
```

Существует два разных размера записи. Значение BinaryRecordFile.record_size определяется пользователем и является размером записи с точки зрения пользователя. Частное значение BinaryRecordFile.__record_size – это истинный размер записи, который включает байт состояния.

Мы предотвращаем усечение файла при открытии, если файл существует (используя режим "r+b"), и создаем его, если файл отсутствует (используя режим "w+b"). Элемент «+» в строке режима указывает, что файл открывается на чтение *и* запись. Если атрибут BinaryRecordFile.auto_flush имеет значение True, файл будет выталкиваться на диск перед каждой операцией чтения и после каждой операции записи.

```
    @property
    def record_size(self):
        return self.__record_size - 1

    @property
    def name(self):
        return self.__fh.name

    def flush(self):
        self.__fh.flush()

    def close(self):
        self.__fh.close()
```

Мы оформили размер записи и имя файла как свойства, доступные только для чтения. Размер записи, который сообщается пользователю, является тем размером, который был установлен пользователем при создании объекта и соответствует размеру записи пользователя. Методы `flush()` и `close()` просто вызывают соответствующие методы объекта файла.

```
    def __setitem__(self, index, record):
        assert isinstance(record, (bytes, bytearray)), \
                "binary data required"
        assert len(record) == self.record_size, (
            "record must be exactly {0} bytes".format(
            self.record_size))
        self.__fh.seek(index * self.__record_size)
        self.__fh.write(_OKAY)
        self.__fh.write(record)
        if self.auto_flush:
            self.__fh.flush()
```

Этот метод обеспечивает поддержку синтаксиса *brf*[*i*] = *data*, *brf* – это объект класса `BinaryRecordFile`, *i* – номер позиции записи и *data* – строка байтов. Обратите внимание, что запись должна иметь тот же размер, что был указан при создании объекта `BinaryRecordFile`. Если аргументы содержат корректные значения, выполняется перемещение указателя в файле в позицию первого байта записи – обратите внимание на то, что здесь используется истинный размер записи, то есть размер с учетом байта состояния. По умолчанию метод `seek()` перемещает указатель в файле в абсолютную позицию. С помощью второго аргумента можно выполнять перемещение относительно текущей позиции или относительно конца файла. (Атрибуты и методы объектов файлов перечислены в табл. 7.4.)

Так как производится изменение элемента, вполне очевидно, что он не был удален, поэтому в байт состояния записывается значение `_OKAY`, а затем записываются двоичные данные пользователя. Объект `BinaryRecordFile` ничего не знает о структуре используемой записи, он беспокоится лишь о том, чтобы записи имели корректный размер.

Метод не проверяет выход индекса за допустимые границы. Если индекс находится за пределами файла, запись будет записана в корректное местоположение, а каждый байт между прежним концом файла и началом новой записи автоматически будет установлен в значение `b"\x00"`. Такие пустые записи не имеют значения `_OKAY` или `_DELETED` в байте состояния, благодаря этому мы сможем их отличать, когда в этом появится необходимость.

```
def __getitem__(self, index):
    self.__seek_to_index(index)
    state = self.__fh.read(1)
    if state != _OKAY:
        return None
    return self.__fh.read(self.record_size)
```

Таблица 7.4. Методы и атрибуты объекта файла

Синтаксис	Описание
`f.close()`	Закрывает объект файла `f` и записывает в атрибут `f.closed` значение `True`
`f.closed`	Возвращает `True`, если файл закрыт
`f.encoding`	Кодировка, используемая при преобразованиях `bytes` ↔ `str`
`f.fileno()`	Возвращает дескриптор файла. (Доступно только для объектов файлов, имеющих дескрипторы.)
`f.flush()`	Выталкивает выходные буферы объекта `f` на диск
`f.isatty()`	Возвращает `True`, если объект файла ассоциирован с консолью. (Доступно только для объектов файлов, ссылающихся на фактические файлы.)
`f.mode`	Режим, в котором был открыт объект файла `f`
`f.name`	Имя файла (если таковое имеется)
`f.newlines`	Виды последовательностей перевода строки, встречающиеся в текстовом файле `f`
`f.__next__()`	Возвращает следующую строку из объекта файла `f`. В большинстве случаев этот метод вызывается неявно, например, `for` *`line`* `in f`
`f.peek(`*`n`*`)`	Возвращает *`n`* байтов без перемещения позиции указателя в файле
`f.read(`*`count`*`)`	Читает до *`count`* байтов из объекта файла `f`. Если значение *`count`* не определено, то читаются все байты, начиная от текущей позиции и до конца. При чтении в двоичном режиме возвращает объект `bytes`, при чтении в текстовом режиме – объект `str`. Если из ничего не было прочитано (конец файла), возвращается пустой объект `bytes` или `str`

Синтаксис	Описание
`f.readable()`	Возвращает `True`, если `f` был открыт для чтения
`f.readinto(ba)`	Читает до `len(ba)` байтов в объект `ba` типа `bytearray` и возвращает число прочитанных байтов (0, если был достигнут конец файла). (Доступен только в двоичном режиме.)
`f.readline(count)`	Читает следующую строку (до *`count`* байтов, если значение *`count`* было определено и число прочитанных байтов было достигнуто раньше, чем встретился символ перевода строки `\n`), включая символ перевода строки `\n`
`f.readlines(`*`sizehint`*`)`	Читает все строки до конца файла и возвращает их в виде списка. Если значение аргумента *`sizehint`* определено, то будет прочитано примерно *`sizehint`* байтов, если внутренние механизмы, на которые опирается объект файла, поддерживают такую возможность
`f.seek(offset,` *`whence`*`)`	Перемещает позицию указателя в файле (откуда будет начато выполнение следующей операции чтения или записи) в заданное смещение, если аргумент *`whence`* не определен или имеет значение `os.SEEK_SET`. Перемещает позицию указателя в файле в заданное смещение (которое может быть отрицательным) относительно текущей позиции, если аргумент *`whence`* имеет значение `os.SEEK_CUR`, или относительно конца файла, если аргумент *`whence`* имеет значение `os.SEEK_END`. Запись всегда выполняется в конец файла, если был определен режим добавления в конец `"a"`, независимо от местоположения указателя в файле. В текстовом режиме в качестве смещений должны использоваться только значения, возвращаемые методом `tell()`
`f.seekable()`	Возвращает `True`, если `f` поддерживает возможность произвольного доступа
`f.tell()`	Возвращает текущую позицию указателя в файле относительно его начала
`f.truncate(`*`size`*`)`	Усекает файл до текущей позиции указателя в файле или до размера *`size`*, если аргумент *`size`* задан
`f.writable()`	Возвращает `True`, если `f` был открыт для записи
`f.write(s)`	Записывает в файл объект `s` типа `bytes/bytearray`, если он был открыт в двоичном режиме, и объект `s` типа `str`, если он был открыт в текстовом режиме
`f.writelines(`*`seq`*`)`	Записывает в файл последовательность объектов (строки – для текстовых файлов, строки байтов – для двоичных файлов)

При чтении записи могут иметь место четыре ситуации, которые следует учитывать: запись не существует, то есть указанный индекс находится за пределами файла; запись пустая; запись была удалена и нормальная запись. Если запись не существует, частный метод `__seek_to_index()` возбудит исключение `IndexError`. В противном случае он переместит указатель в файле в позицию первого байта требуемой записи, и мы можем прочитать байт состояния. Если состояние не равно значению `_OKAY`, то запись должна быть либо пустой, либо удаленной, и в этом случае вызывающей программе возвращается значение `None`, в противном случае возвращается запись. (При попытке чтения пустой или удаленной записи, вместо того чтобы возвращать `None`, можно было бы возбуждать наше собственное исключение, например, `BlankRecordError` или `DeletedRecordError`.)

```
def __seek_to_index(self, index):
    if self.auto_flush:
        self.__fh.flush()
    self.__fh.seek(0, os.SEEK_END)
    end = self.__fh.tell()
    offset = index * self.__record_size
    if offset >= end:
        raise IndexError("no record at index position {0}".format(
                         index))
    self.__fh.seek(offset)
```

Этот частный вспомогательный метод используется некоторыми другими методами для перемещения указателя в файле в позицию первого байта записи с заданным индексом. Сначала метод проверяет, находится ли заданный индекс в пределах файла. Для этого выполняется перемещение указателя в конец файла (смещение 0 относительно конца файла), и с помощью метода `tell()` определяется абсолютная позиция указателя. Если смещение записи (индекс×истинный размер записи) оказывается в конце файла или за его пределами, возбуждается соответствующее исключение. В противном случае выполняется перемещение указателя в заданную позицию, откуда будет выполняться следующая операция чтения или записи.

```
def __delitem__(self, index):
    self.__seek_to_index(index)
    state = self.__fh.read(1)
    if state != _OKAY:
        return
    self.__fh.seek(index * self.__record_size)
    self.__fh.write(_DELETED)
    if self.auto_flush:
        self.__fh.flush()
```

Сначала метод выполняет перемещение в нужную позицию в файле. Если указанный индекс находится в пределах файла (то есть если не было возбуждено исключение `IndexError`) и запись не пустая и не была

удалена ранее, то выполняется запись значения _DELETED в байт состояния записи.

```
def undelete(self, index):
    self.__seek_to_index(index)
    state = self.__fh.read(1)
    if state == _DELETED:
        self.__fh.seek(index * self.__record_size)
        self.__fh.write(_OKAY)
        if self.auto_flush:
            self.__fh.flush()
        return True
    return False
```

Сначала метод отыскивает требуемую запись и читает байт состояния. Если запись была удалена, в байт состояния записывается значение _OKAY и вызывающей программе возвращается значение True как свидетельство успешного выполнения операции; в противном случае (для пустой или не удалявшейся ранее записи) возвращается значение False.

```
def __len__(self):
    if self.auto_flush:
        self.__fh.flush()
    self.__fh.seek(0, os.SEEK_END)
    end = self.__fh.tell()
    return end // self.__record_size
```

Этот метод возвращает количество записей в двоичном файле. Это число определяется путем деления позиции последнего байта в файле (то есть количества байтов в файле) на истинный размер записи.

На этом мы закончили рассмотрение основных функциональных возможностей класса BinaryRecordFile.BinaryRecordFile. Остался последний вопрос, который необходимо рассмотреть: уплотнение файла с целью убрать пустые и удаленные записи. Фактически имеется два способа решения этой задачи. Первый способ состоит в том, чтобы перезаписать пустые или удаленные записи записями с большими значениями индексов и усечь файл с конца, если в нем имелись пустые или удаленные записи. Этот способ реализован в методе inplace_compact(). Другой способ состоит в том, чтобы скопировать непустые и неудаленные записи во временный файл и затем переименовать его, дав имя оригинального файла. Использование временного файла удобно, в частности для создания резервных копий. Этот способ реализован в методе compact().

Начнем рассмотрение с метода inplace_compact(), разделив его на две части:

```
def inplace_compact(self):
    index = 0
    length = len(self)
```

```
        while index < length:
            self.__seek_to_index(index)
            state = self.__fh.read(1)
            if state != _OKAY:
                for next in range(index + 1, length):
                    self.__seek_to_index(next)
                    state = self.__fh.read(1)
                    if state == _OKAY:
                        self[index] = self[next]
                        del self[next]
                        break
                else:
                    break
            index += 1
```

В методе выполняются итерации по всем записям и для каждой определяется ее состояние. Если обнаруживается пустая или удаленная запись, выполняется поиск следующей непустой и неудаленной записи. Если такая запись обнаруживается, производится замещение пустой или удаленной записи непустой и неудаленной записью и выполняется удаление оригинальной записи; в противном случае цикл `while` прерывается, так как были просмотрены все непустые и неудаленные записи.

```
        self.__seek_to_index(0)
        state = self.__fh.read(1)
        if state != _OKAY:
            self.__fh.truncate(0)
        else:
            limit = None
            for index in range(len(self) - 1, 0, -1):
                self.__seek_to_index(index)
                state = self.__fh.read(1)
                if state != _OKAY:
                    limit = index
                else:
                    break
            if limit is not None:
                self.__fh.truncate(limit * self.__record_size)
        self.__fh.flush()
```

Если первая запись пустая или удаленная, то все они должны быть пустыми или удаленными, так как предыдущий фрагмент кода переместил все непустые и неудаленные записи в начало файла, оставив пустые и удаленные записи в конце. В этом случае можно просто усечь размер файла до нуля.

Если имеется хотя бы одна непустая и неудаленная запись, метод выполняет итерации в обратном порядке, от конца файла, поскольку известно, что все пустые и удаленные записи были перемещены в конец. Переменная `limit` получает в качестве значения индекс самой первой

пустой или удаленной записи (или значение None, если в файле нет пустых или удаленных записей) и соответственно этому значению производится усечение файла.

Альтернативное решение задачи уплотнения файла состоит в копировании записей в другой файл, что можно использовать для создания резервных копий. Это решение реализовано в методе compact(), который показан ниже.

```
def compact(self, keep_backup=False):
    compactfile = self.__fh.name + ".$$$"
    backupfile = self.__fh.name + ".bak"
    self.__fh.flush()
    self.__fh.seek(0)
    fh = open(compactfile, "wb")
    while True:
        data = self.__fh.read(self.__record_size)
        if not data:
            break
        if data[:1] == _OKAY:
            fh.write(data)
    fh.close()
    self.__fh.close()

    os.rename(self.__fh.name, backupfile)
    os.rename(compactfile, self.__fh.name)
    if not keep_backup:
        os.remove(backupfile)
    self.__fh = open(self.__fh.name, "r+b")
```

Этот метод создает два файла – уплотненный файл и резервную копию оригинального файла. Имя уплотненного файла совпадает с именем оригинального файла, но к нему добавляется расширение *.$$$*, точно так же имя файла резервной копии совпадает с именем оригинального файла, но имеет расширение *.bak*. Метод читает записи из оригинального файла одну за другой и все непустые и неудаленные записи записываются в уплотненный файл. (Обратите внимание, что записываются истинные записи, то есть байт состояния плюс запись пользователя.)

Инструкция `if data[:1] == _OKAY:` таит в себе одну хитрость. Оба объекта – и объект data и объект _OKAY – являются объектами типа bytes. Нам необходимо сравнить первый байт (один байт) объекта data с объектом _OKAY. Когда к объекту типа bytes применяется операция среза, возвращается объект bytes, но когда извлекается единственный байт, например, data[0], возвращается объект типа int – значение байта. Поэтому здесь сравниваются 1-байтовый срез объекта data (его первый байт, байт состояния) с 1-байтовым объектом _OKAY. (Сравнение можно было бы реализовать как `if data[0] == _OKAY[0]:`, в этом случае сравнивались бы два значения типа int.)

Типы данных bytes и bytearray, стр. 344

В конце оригинальному файлу присваивается имя резервной копии, а уплотненному файлу – имя оригинального файла. После этого, если аргумент keep_backup имеет значение False (по умолчанию), файл резервной копии удаляется. В заключение, чтобы подготовиться к последующим операциям чтения и записи, уплотненный файл (который теперь имеет имя оригинального файла) открывается.

Класс BinaryRecordFile.BinaryRecordFile содержит весьма низкоуровневую реализацию, но он может служить основой для классов более высокого уровня, где необходима возможность произвольного доступа к данным в файлах, хранящих записи фиксированного размера; это будет показано в следующем подразделе.

Пример: классы в модуле BikeStock

Модуль BikeStock использует класс BinaryRecordFile.BinaryRecordFile для управления простым хранилищем информации. Элементами хранения является информация о велосипедах, каждый из которых представляет собой экземпляр класса BikeStock.BikeStock. Класс BikeStock.BikeStock содержит в себе словарь, ключами которого являются идентификаторы велосипедов, а значениями – индексы соответствующих записей в BinaryRecordFile.BinaryRecordFile. Ниже приводится короткий пример, дающий некоторое представление о том, как работают эти классы:

```
bicycles = BikeStock.BikeStock(bike_file)
value = 0.0
for bike in bicycles:
    value += bike.value
bicycles.increase_stock("GEKKO", 2)
for bike in bicycles:
    if bike.identity.startswith("B4U"):
        if not bicycles.increase_stock(bike.identity, 1):
            print("stock movement failed for", bike.identity)
```

Этот фрагмент программного кода открывает файл хранилища информации о велосипедах, выполняет итерации по всем содержащимся в нем записям и определяет общую стоимость (сумма произведений цена× количество) велосипедов на складе. Затем он увеличивает на два количество велосипедов «GEKKO», хранящихся на складе, и на один – количество велосипедов, названия которых начинаются с «B4U». Все эти действия выполняются непосредственно с информацией на диске, поэтому любые другие процессы, обращающиеся к файлу хранилища, имеют доступ к самой свежей информации.

Класс BinaryRecordFile.BinaryRecordFile работает с файлом в терминах индексов, тогда как класс BikeStock.BikeStock работает в терминах идентификаторов велосипедов. Это возможно благодаря тому, что экземпляр класса BikeStock.BikeStock хранит словарь, устанавливаю-

щий отношения между идентификаторами велосипедов и индексами записей.

Сначала рассмотрим инструкцию `class` и метод инициализации класса `BikeStock.Bike`, затем обсудим некоторые методы класса `BikeStock.BikeStock` и в заключение посмотрим на программный код, играющий роль связующего звена между объектами `BikeStock.Bike` и двоичными записями, представляющими их в `BinaryRecordFile.BinaryRecordFile` (весь программный код находится в файле *BikeStock.py*).

```
class Bike:

    def __init__(self, identity, name, quantity, price):
        assert len(identity) > 3, ("invalid bike identity '{0}'"
                                   .format(identity))
        self.__identity = identity
        self.name = name
        self.quantity = quantity
        self.price = price
```

Все атрибуты класса `Bike` доступны внешнему программному коду как свойства – идентификатор велосипеда (`self.__identity`) представляет свойство `Bike.identity`, доступное только для чтения, остальные свойства доступны как для чтения, так и для записи и обеспечивают дополнительную проверку корректности записываемых данных с помощью инструкции `assert`. Дополнительно имеется свойство `Bike.value`, доступное только для чтения, возвращающее произведение цены на количество. (Здесь не приводится программный код реализации свойств, так как он похож на программный код, который приводился ранее.)

Класс `BikeStock.BikeStock` реализует собственные методы манипулирования объектами типа `BikeStock.Bike`, которые используют свойства объектов класса `BikeStock.Bike`, доступные для записи.

```
class BikeStock:

    def __init__(self, filename):
        self.__file = BinaryRecordFile.BinaryRecordFile(filename,
                                                _BIKE_STRUCT.size)
        self.__index_from_identity = {}
        for index in range(len(self.__file)):
            record = self.__file[index]
            if record is not None:
                bike = _bike_from_record(record)
                self.__index_from_identity[bike.identity] = index
```

Класс `BikeStock.BikeStock` – это наш собственный класс коллекций, агрегирующий экземпляр класса `BinaryRecordFile.BinaryRecordFile` (`self.__file`) и словарь (`self.__index_from_identity`), ключами которого являются идентификаторы велосипедов, а значениями – индексы записей с информацией о них.

После открытия файла (или создания, если перед этим файл не существовал) выполняются итерации по записям, содержащимся в нем (если таковые имеются). Каждая извлеченная запись преобразуется из объекта типа `bytes` в объект `BikeStock.Bike` с помощью частной функции `__bike_from_record()`, после чего идентификатор велосипеда и индекс записи добавляются в словарь `self.__index_from_identity`.

```
    def append(self, bike):
        index = len(self.__file)
        self.__file[index] = _record_from_bike(bike)
        self.__index_from_identity[bike.identity] = index
```

Чтобы добавить новый велосипед, необходимо определить подходящий номер позиции и поместить в эту позицию запись с двоичным представлением информации о велосипеде. При этом мы не забываем дополнить словарь `self.__index_from_identity`.

```
def __delitem__(self, identity):
    del self.__file[self.__index_from_identity[identity]]
```

Удаление записи с информацией о велосипеде выполняется очень просто – достаточно по идентификатору определить номер позиции и удалить запись в этой позиции. В классе `BikeStock.BikeStock` не предполагается использовать возможность восстановления удаленных записей, предусматриваемую классом `BinaryRecordFile.BinaryRecordFile`.

```
def __getitem__(self, identity):
    record = self.__file[self.__index_from_identity[identity]]
    return None if record is None else _bike_from_record(record)
```

Записи с информацией о велосипедах извлекаются по идентификатору велосипеда. Если в словаре `self.__index_from_identity` отсутствует запрошенный идентификатор, возбуждается исключение `KeyError`, а если запись пустая или была удалена, объект `BinaryRecordFile.BinaryRecordFile` вернет значение `None`. Но если запись существует, она возвращается в виде объекта `BikeStock.Bike`.

```
    def __change_stock(self, identity, amount):
        index = self.__index_from_identity[identity]
        record = self.__file[index]
        if record is None:
            return False
        bike = _bike_from_record(record)
        bike.quantity += amount
        self.__file[index] = _record_from_bike(bike)
        return True

    increase_stock = (lambda self, identity, amount:
                                self.__change_stock(identity, amount))
    decrease_stock = (lambda self, identity, amount:
                                self.__change_stock(identity, -amount))
```

Частный метод `__change_stock()` содержит реализацию для методов `increase_stock()` и `decrease_stock()`. Он определяет индекс записи и извлекает ее в двоичном представлении. Затем запись преобразуется в объект `BikeStock.Bike`, к этому объекту применяются необходимые изменения, после чего двоичная запись в файле затирается двоичным представлением измененного объекта. (Существует также метод `__change_bike()`, содержащий реализацию методов `change_name()` и `change_price()`, но ни один из них не будет рассматриваться здесь, так как они очень похожи на методы, продемонстрированные выше.)

```
    def __iter__(self):
        for index in range(len(self.__file)):
            record = self.__file[index]
            if record is not None:
                yield _bike_from_record(record)
```

Этот метод обеспечивает возможность итераций через объекты `BikeStock.BikeStock`, как через списки, возвращая на каждой итерации объект `BikeStock.Bike` и пропуская пустые и удаленные записи.

Частные функции `_bike_from_record()` и `_record_from_bike()` отделяют двоичное представление объектов класса `BikeStock.Bike` от класса `BikeStock.BikeStock`, хранящего коллекцию велосипедов. Логическая структура записи с информацией о велосипеде в файле показана на рис. 7.4. Физическая структура записи несколько отличается, потому что каждая запись содержит дополнительный байт состояния.

```
_BIKE_STRUCT = struct.Struct("<8s30sid")

def _bike_from_record(record):
    ID, NAME, QUANTITY, PRICE = range(4)
    parts = list(_BIKE_STRUCT.unpack(record))
    parts[ID] = parts[ID].decode("utf8").rstrip("\x00")
    parts[NAME] = parts[NAME].decode("utf8").rstrip("\x00")
    return Bike(*parts)

def _record_from_bike(bike):
    return _BIKE_STRUCT.pack(bike.identity.encode("utf8"),
```

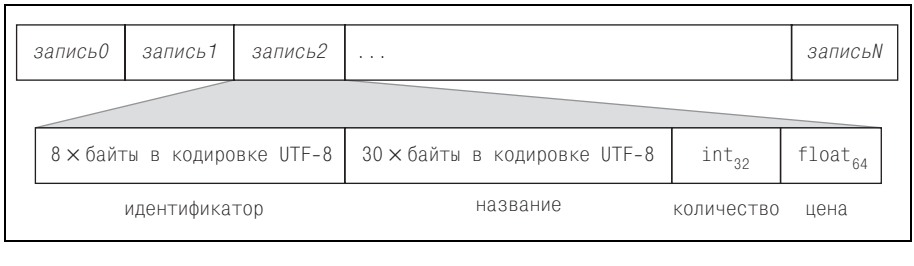

***Рис. 7.4.** Логическая структура файла, хранящего записи с информацией о велосипедах*

```
                bike.name.encode("utf8"),
                bike.quantity, bike.price)
```

При преобразовании двоичной записи в объект `BikeStock.Bike` сначала выполняется преобразование кортежа, возвращаемого методом `unpack()`, в список. Это позволяет выполнять модификацию элементов, в данном случае – преобразовывать байты в кодировке UTF-8 в строки, с усечением завершающих байтов `0x00`. После этого с помощью оператора распаковывания последовательностей (`*`) осуществляется передача отдельных полей записи методу инициализации класса `BikeStock.Bike`. Упаковывание данных выполняется намного проще, при этом не следует забывать о необходимости преобразования строк в последовательности байтов UTF-8.

Потребность в прикладных программах, осуществляющих произвольный доступ к двоичным данным в файлах, уменьшается по мере увеличения объемов оперативной памяти и скорости работы дисков в современных настольных системах. А когда возникает потребность в такой функциональности, часто бывает проще использовать файлы DBM или базы данных SQL. Тем не менее существуют системы, где может оказаться востребованной функциональность, продемонстрированная выше, например, во встроенных системах и других системах с ограниченными ресурсами.

В заключение

В этой главе были продемонстрированы широко используемые приемы сохранения коллекций данных в файлах и загрузки их из файлов. Мы увидели, насколько прост в использовании модуль `pickle` и как можно обрабатывать сжатые и несжатые файлы, не зная заранее, использовалось ли сжатие.

Мы узнали, какую заботу необходимо проявлять при записи и чтении двоичных данных, и увидели, насколько длинным может получиться программный код, когда требуется обеспечить обработку строк переменной длины. Но мы также узнали, что использование двоичных форматов дает в результате файлы наименьшего размера и обеспечивает наивысшую скорость записи и чтения. Кроме того, мы узнали насколько важно использовать сигнатуры для идентификации типа файла и номера версий, чтобы упростить изменение формата файла в будущем.

В этой главе мы увидели, что простой текстовый формат наиболее удобен для восприятия человеком и что при хорошо продуманной структуре он сможет легко обрабатываться дополнительными инструментами, которые будут созданы для манипулирования данными. Однако анализ текста может оказаться непростым делом. Мы видели, как можно читать текстовые данные вручную и с помощью регулярных выражений.

XML – весьма популярный формат обмена данными и, вообще говоря, будет совсем нелишним предусмотреть в программе хотя бы возможность импортирования и экспортирования данных в формате XML, даже если основным используемым форматом является двоичный или текстовый. Мы увидели, как вручную выполнять запись данных в формате XML, включая корректное экранирование значений атрибутов и текстовой информации, и как записывать эти данные средствами дерева элементов и модели DOM. Мы также узнали, как выполнять парсинг содержимого файлов XML с помощью парсеров дерева элементов, DOM и SAX, которые предоставляются стандартной библиотекой языка Python.

В заключительном разделе главы мы увидели, как создать универсальный класс для обеспечения произвольного доступа к двоичным данным в файлах, хранящих записи фиксированного размера, и затем увидели, как использовать этот класс в конкретном контексте.

Этой главой заканчивается изучение фундаментальных основ программирования на языке Python. Уже сейчас можно прекратить чтение книги и, используя полученные знания, писать отличные программы. Но было бы неразумно останавливаться на достигнутом, потому что язык Python может предложить намного больше, начиная от приемов, позволяющих сократить и упростить программный код, и заканчивая ошеломляющими средствами, о существовании которых полезно хотя бы знать, даже если они будут востребованы нечасто. В следующей главе мы продолжим изучение вопросов процедурного и объектно-ориентированного программирования, но дополнительно познакомимся с функциональным программированием. Затем, в последующих главах, мы сосредоточимся на изучении более широких приемов программирования, включая программирование многопоточных приложений, организацию сетевых взаимодействий, работу с базами данных, использование регулярных выражений и создание программ с графическим интерфейсом пользователя.

Упражнения

В первом упражнении предлагается создать более простой модуль для работы с двоичным файлом по сравнению с тем, что был представлен в этой главе. Истинный размер записи в этом файле точно совпадает с размером, который указывается пользователем. Во втором упражнении предлагается изменить модуль `BikeStock` так, чтобы он использовал новый модуль для работы с двоичным файлом. В третьем упражнении предлагается написать программу с самого начала – операции с файлом в ней не отличаются сложностью, но форматирование вывода может оказаться трудным в реализации.

1. Создайте новую версию более простого модуля `BinaryRecordFile`, в котором не используется байт состояния записи. В этой версии

размер записи, устанавливаемый пользователем, должен совпадать с истинным размером записи. Новые записи должны добавляться с помощью нового метода `append()`, который просто перемещает указатель в конец файла и производит вывод записи в файл. Метод `__setitem__()` должен позволять замещать только существующие записи – это легко реализовать с помощью метода `__seek_to_index()`. Из-за отсутствия байта состояния размер метода `__getitem__()` должен сократиться до трех строк. Метод `__delitem__()` придется полностью переписать, так как он должен будет перемещать все записи, следующие за удаленной, чтобы заполнить освободившийся промежуток. Сделать это можно с помощью чуть больше половины десятка строк, но над реализацией придется подумать. Метод `undelete()` нужно будет полностью убрать, так как теперь операция восстановления поддерживаться не будет. Точно так же надо будет убрать методы `compact()` и `inplace_compact()`, так как они больше не нужны.

Чтобы внести описанные изменения, придется добавить не более 20 новых и строк и удалить по крайней мере 60 строк по сравнению с оригинальным модулем, и это без учета доктестов. Пример решения приводится в файле *BinaryRecordFile_ans.py*.

2. Как только вы будете уверены, что ваш более простой класс `BinaryRcordFile` работает, скопируйте файл *BikeStock.py* и измените его так, чтобы он работал с вашим классом `BinaryRcordFile`. Для этого придется изменить всего несколько строк. Пример решения приводится в файле *BikeStock_ans.py*.
3. Отладка действий с двоичными форматами может оказаться достаточно сложным делом, и в этом может помочь инструмент, который выводит шестнадцатеричные дампы содержимого двоичных файлов. Напишите программу, которая выводила бы в консоли следующий текст справки:

```
Usage: xdump.py [options] file1 [file2 [... fileN]]

Options:
  -h, --help            show this help message and exit
  -b BLOCKSIZE, --blocksize=BLOCKSIZE
                        block size (8..80) [default: 16]
-d, --decimal           decimal block numbers [default: hexadecimal]
-e ENCODING, --encoding=ENCODING
                        encoding (ASCII..UTF-32) [default: UTF-8]
(Перевод:
Порядок использования: xdump.py [параметры] file1 [file2 [... fileN]]

Параметры:
  -h, --help            вывести это справочное сообщение и выйти
  -b BLOCKSIZE, --blocksize=BLOCKSIZE
                        Размер блока (8..80) [по умолчанию: 16]
-d, --decimal           блоки десятичных чисел
                        [по умолчанию: шестнадцатеричные]
```

```
-e ENCODING, --encoding=ENCODING
                      кодировка (ASCII..UTF-32) [по умолчанию: UTF-8]
конец перевода)
```

С помощью этой программы при наличии файла, созданного объектом `BinaryRecordFile`, в котором хранятся записи в формате `"<i10s"` (обратный порядок следования байтов, 4-байтовое целое со знаком, 10-байтовая строка байтов), установив размер блока в соответствии с размером одной записи (15 байтов, включая байт состояния), можно было бы получить ясное представление о содержимом файла. Например:

```
xdump.py -b15 test.dat
Block     Bytes                              UTF-8 characters
--------  ---------------------------------  ----------------
00000000  02000000 00416C70 68610000 000000  .....Alpha.....
00000001  01140000 00427261 766F0000 000000  .....Bravo.....
00000002  02280000 00436861 726C6965 000000  .(...Charlie...
00000003  023C0000 0044656C 74610000 000000  .<...Delta.....
00000004  02500000 00456368 6F000000 000000  .P...Echo......
```

Каждый байт представлен двумя шестнадцатеричными цифрами; пробел между группами из четырех байтов (между группами из восьми шестнадцатеричных цифр) добавляется исключительно ради удобочитаемости. В этом примере видно, что вторая запись («Bravo») была удалена, потому что ее байт состояния имеет значение `0x01`, а не `0x02`, используемое для обозначения непустых и неудаленных записей.

Для обработки параметров командной строки используйте модуль `optparse`. (Указав «тип» параметра, можно заставить модуль `optparse` выполнять преобразование значения параметра с размером блока из строкового представления в целочисленное.) Может оказаться совсем непросто правильно выводить строку заголовка для произвольно заданного размера блока и строки символов в последнем блоке, поэтому обязательно проверьте работу программы с разными размерами блоков (например, 8, 9, 10, ..., 40). Кроме того, не забывайте, что в файлах переменной длины последний блок может оказаться коротким. Для обозначения непечатаемых символов используйте точку, как показано в примере.

Программу можно уместить менее чем в 70 строк, распределенных на две функции. Пример решения приводится в файле *xdump.py*.

8

- Улучшенные приемы процедурного программирования
- Улучшенные приемы объектно-ориентированного программирования
- Функциональное программирование

Усовершенствованные приемы программирования

В этой главе мы рассмотрим широкий диапазон различных приемов программирования и представим множество дополнительных, усовершенствованных синтаксических конструкций, поддерживаемых языком Python. Некоторые сведения, приводимые в этой главе, отличаются высокой сложностью, но имейте в виду, что большая часть дополнительных приемов используется нечасто и при первом прочтении вы можете лишь ознакомиться с ними, чтобы получить о них общее представление, и перечитать материал внимательнее, когда в этом возникнет необходимость.

В первом разделе главы более подробно рассматриваются особенности процедурного программирования на языке Python. Раздел начинается с демонстрации решения уже описанных ранее задач новым способом и затем возвращается к теме генераторов, которая кратко рассматривалась в главе 6. Затем в этом разделе рассматриваются приемы динамического программирования – загрузка модулей по имени во время выполнения и выполнение произвольного программного кода. После этого изложение возвращается к теме локальных (вложенных) функций, которая расширена описанием использования ключевого слова `nonlocal` и рекурсивных функций. Ранее мы видели, как можно использовать предопределенные декораторы языка Python, а в этом разделе мы узнаем, как создавать собственные декораторы. Завершается раздел обсуждением аннотаций функций.

Во втором разделе содержатся новые сведения об объектно-ориентированном программировании. Он начинается с представления механизма слотов (`__slots__`), предназначенного для уменьшения объема памя-

ти, занимаемой каждым объектом. Затем он демонстрирует, как организовать доступ к атрибутам без использования свойств. В этом разделе также будут представлены функторы (объекты, которые могут вызываться подобно функциям) и менеджеры контекста – они используются совместно с ключевым словом `with` и во многих случаях (например, при работе с файлами) могут использоваться вместо конструкций `try ... except ... finally`, замещая их более простыми конструкциями `try ... except`. В этом разделе также будет показано, как создавать свои собственные менеджеры контекста, и будут представлены дополнительные улучшенные особенности объектно-ориентированного программирования, включая декораторы классов, абстрактные базовые классы, множественное наследование и метаклассы.

В третьем разделе вводится несколько фундаментальных понятий функционального программирования и представлены некоторые полезные функции из модулей `functools`, `itertools` и `operator`. В этом разделе также будет показано, как использовать возможность частичной подготовки функций для упрощения программного кода.

Предыдущие главы предоставили нам «комплект стандартных инструментов языка Python». Эта глава берет все, что мы уже рассматривали, и превращает в «комплект усовершенствованных инструментов», в котором присутствуют все прежние инструменты (приемы программирования и синтаксические конструкции) плюс множество новых, которые могут сделать программирование проще, легче и эффективнее. Некоторые инструменты являются взаимозаменяемыми, например, некоторые задачи можно решать с помощью декораторов классов или метаклассов, тогда как другие, такие как дескрипторы, при разных способах использования дают различные результаты. Некоторые описываемые здесь инструменты, такие как менеджеры контекста, мы будем использовать постоянно, другие – время от времени, только в определенных ситуациях, когда они способны предложить лучшее решение.

Улучшенные приемы процедурного программирования

Большая часть этого раздела посвящена дополнительным возможностям, касающимся процедурного программирования и функций, но самый первый подраздел в этом отношении стоит особняком, так как в нем представлены полезные приемы программирования, основанные на уже имеющихся у нас знаниях, без введения новых синтаксических конструкций.

Ветвление с использованием словарей

Как уже отмечалось ранее, функции – это объекты, как и все остальное в языке Python, а имена функций – это ссылки на объекты, кото-

рые указывают на функции. Если записать имя функции без скобок, интерпретатор будет считать, что подразумевается ссылка на объект, благодаря чему имеется возможность передавать такие ссылки на объекты точно так же, как ссылки на любые другие объекты. Этот факт можно использовать для замены условных инструкций `if`, содержащих множественные предложения `elif`, единственным вызовом функции.

В главе 11 мы будем рассматривать интерактивную консольную программу с именем *dvds-dbm.py*, которая имеет следующее меню:

```
(A)dd (E)dit (L)ist (R)emove (I)mport e(X)port (Q)uit
```

В программе имеется функция, которая получает символ, выбранный пользователем, и возвращает только допустимый символ, в данном случае «a», «e», «l», «r», «i», «x» или «q». Ниже приводятся два эквивалентных фрагмента программного кода, которые, в зависимости от сделанного выбора, вызывают соответствующую функцию:

```
if action == "a":
    add_dvd(db)
elif action == "e":
    edit_dvd(db)
elif action == "l":
    list_dvds(db)
elif action == "r":
    remove_dvd(db)
elif action == "i":
    import_(db)
elif action == "x":
    export(db)
elif action == "q":
    quit(db)
```

```
functions = dict(a=add_dvd, e=edit_dvd,
                 l=list_dvds, r=remove_dvd,
                 i=import_, x=export, q=quit)
functions[action](db)
```

Выбор, сделанный пользователем, хранится в виде строки из одного символа в переменной `action`, а ссылка на используемую базу данных – в переменной `db`. В имя функции `import_()` включен завершающий символ подчеркивания, чтобы отличить ее от инструкции `import`.

Фрагмент справа создает словарь, ключами которого являются допустимые варианты выбора, а значениями – ссылки на функции. Вторая инструкция в этом фрагменте извлекает ссылку на функцию, соответствующую выбранному действию, и вызывает ее с помощью оператора вызова `()`, передавая аргумент `db`. Фрагмент справа не только короче, но и легко масштабируется (в словаре может быть гораздо больше элементов) без потерь производительности, в отличие от фрагмента слева, скорость работы которого зависит от того, сколько условий в предложениях `elif` придется проверить, прежде чем будет найдена требуемая функция.

Этот прием уже использовался в программе *convert-incidents.py* из предыдущей главы – в методе `import_()`, как показано в выдержке из этого метода ниже:

```
call = {(".aix", "dom"): self.import_xml_dom,
        (".aix", "etree"): self.import_xml_etree,
        (".aix", "sax"): self.import_xml_sax,
        (".ait", "manual"): self.import_text_manual,
        (".ait", "regex"): self.import_text_regex,
        (".aib", None): self.import_binary,
        (".aip", None): self.import_pickle}
result = call[extension, reader](filename)
```

Всего метод содержит 13 строк программного кода. Значение `extension` определяется в самом методе, а значение `reader` передается вызывающей программой. Ключами словаря являются двухэлементные кортежи, а значениями – методы. Если бы в этом случае использовались инструкции `if`, реализация метода выросла бы до 22 строк, а понятие масштабируемости к реализации было бы вообще неприменимо.

Выражения-генераторы и функции-генераторы

Функции-генераторы, стр. 324

В главе 6 мы познакомились с функциями-генераторами и методами-генераторами. Кроме того, существует возможность создавать еще и выражения-генераторы. Синтаксически они очень похожи на генераторы списков, единственное отличие состоит в том, что они заключаются не в квадратные скобки, а в круглые. Ниже приводится синтаксис выражений-генераторов в общем виде:

```
(expression for item in iterable)
(expression for item in iterable if condition)
```

В предыдущей главе мы создавали методы-генераторы, используя инструкцию `yield`. Ниже приводятся два эквивалентных фрагмента программного кода, демонстрирующие, как простой цикл `for ... in`, содержащий выражение `yield`, можно превратить в генератор:

```
def items_in_key_order(d):
    for key in sorted(d):
        yield key, d[key]
```

```
def items_in_key_order(d):
    return ((key, d[key])
            for key in sorted(d))
```

Обе функции возвращают генератор, который воспроизводит список элементов «ключ-значение» для заданного словаря. Если потребуется получить сразу весь список элементов, возвращаемый функциями генератор можно передать функции `list()` или `tuple()`, или, наоборот, выполнять итерации через генератор, извлекая элементы по мере необходимости.

Генераторы представляют собой средство выполнения отложенных вычислений, то есть значения вычисляются, только когда они дейст-

вительно необходимы. Такой подход может оказаться гораздо эффективнее, чем, например, вычисление содержимого огромного списка за один раз. Некоторые генераторы могут воспроизводить столько значений, сколько потребуется – без ограничения сверху. Например:

```
def quarters(next_quarter=0.0):
    while True:
        yield next_quarter
        next_quarter += 0.25
```

Эта функция будет возвращать числа 0.0, 0.25, 0.5 и т. д. до бесконечности. Ниже показано, как можно было бы использовать такой генератор:

```
result = []
for x in quarters():
    result.append(x)
    if x >= 1.0:
        break
```

Применение инструкции `break` здесь очень существенно – без нее цикл `for ... in` был бы бесконечным. После выхода из цикла переменная `result` будет содержать список `[0.0, 0.25, 0.5, 0.75, 1.0]`.

Всякий раз, когда вызывается функция `quarters()`, она возвращает генератор, начинающий счет с 0.0 и на каждом шаге увеличивающий значение на 0.25, но как быть, если требуется, чтобы генератор начал воспроизводить последовательность с текущего значения? Сделать это можно, передав требуемое значение в генератор, как показано в новой версии функции-генератора:

```
def quarters(next_quarter=0.0):
    while True:
        received = (yield next_quarter)
        if received is None:
            next_quarter += 0.25
        else:
            next_quarter = received
```

Выражение `yield` поочередно возвращает каждое значение вызывающей программе. Кроме того, если будет вызван метод `send()` генератора, то переданное значение будет принято функцией-генератором в качестве результата выражения `yield`. Ниже показано, как можно использовать новую функцию-генератор:

```
result = []
generator = quarters()
while len(result) < 5:
    x = next(generator)
    if abs(x - 0.5) < sys.float_info.epsilon:
        x = generator.send(1.0)
    result.append(x)
```

Здесь создается переменная, хранящая ссылку на генератор, и вызывается встроенная функция `next()`, которая извлекает очередной элемент из указанного ей генератора. (Того же эффекта можно было бы достичь вызовом специального метода `__next__()` генератора, в данном случае следующим образом: `x = generator.__next__()`.) Если значение равно 0.5, генератору передается значение 1.0 (которое немедленно возвращается обратно). На этот раз в результате будет получен список `[0.0, 0.25, 1.0, 1.25, 1.5]`.

В следующем подразделе мы рассмотрим программу *magic-numbers.py*, которая обрабатывает файлы, полученные в виде аргументов командной строки. К сожалению, в операционной системе Windows командная оболочка (*cmd.exe*) не обеспечивает расширения шаблонных символов в именах файлов (также называется *подстановкой имен файлов*, *file globbing*), поэтому, если программу запустить в Windows с аргументом `*.*`, в список `sys.argv` попадет не список всех файлов в текущем каталоге, а сам текст «*.*». Эта проблема была решена за счет создания двух различных функций `get_files()`, одной – для Windows и другой – для UNIX. В обеих функциях используются генераторы, как показано ниже:

```
if sys.platform.startswith("win"):
    def get_files(names):
        for name in names:
            if os.path.isfile(name):
                yield name
            else:
                for file in glob.iglob(name):
                    if not os.path.isfile(file):
                        continue
                    yield file
else:
    def get_files(names):
        return (file for file in names if os.path.isfile(file))
```

В обоих случаях функция ожидает получить в виде аргумента список имен файлов, например, `sys.argv[1:]`.

В Windows функция выполняет обход всех имен в списке. Если очередное имя является именем файла, функция возвращает его; если это не имя файла (обычно имя каталога), то используется функция `glob.iglob()` из модуля `glob`, возвращающая итератор имен файлов, соответствующих указанному имени после расширения шаблонных символов. Для обычных имен, таких как *autoexec.bat*, возвращается итератор, воспроизводящий единственный элемент (имя), а для имен, содержащих шаблонные символы, таких как **.txt*, возвращается итератор, который воспроизводит все имена файлов, соответствующие шаблону (в данном случае – все имена файлов с расширением *.txt*). (Существует также функция `glob.glob()`, возвращающая не итератор, а список.)

В операционной системе UNIX подстановка на место шаблонных символов выполняется самой командной оболочкой, поэтому функция просто возвращает генератор всех полученных имен файлов.[1]

Функции-генераторы могут использоваться для создания *сопрограмм* – функций, котрые имеют несколько точек входа и выхода (выражений `yield`) и которые могут приостанавливаться и возобновляться в определенных точках (опять же в местах, где находятся выражения `yield`). Сопрограммы часто используются в качестве более простой и с меньшими накладными расходами альтернативы многопоточному программированию. В каталоге пакетов Python Package Index (*pypi.python.org/pypi*) имеется несколько модулей сопрограмм.

Динамическое выполнение программного кода и динамическое импортирование

В некоторых случаях бывает проще написать программный код, который генерирует другой программный код, чем писать напрямую весь необходимый программный код. А в некоторых случаях бывает удобнее дать пользователю возможность вводить свой программный код (например, функции в электронных таблицах) и позволить интерпретатору Python выполнить введенный программный код, чем тратить время на создание синтаксического анализатора; хотя подобная возможность выполнять произвольный программный код влечет за собой угрозу безопасности. Другой случай, когда может пригодиться возможность динамического выполнения программного кода, – поддержка архитектуры расширений, добавляющих в программу новые функциональные возможности. Недостаток расширяемой архитектуры состоит в том, что не вся необходимая функциональность встроена в программу непосредственно (что может осложнить развертывание программного продукта и добавить риск потери отдельных расширений). Но в этом есть и свои преимущества, так как расширения могут обновляться по отдельности и могут поставляться отдельно от программы, – например, пополняя ее новыми возможностями, которые не были предусмотрены первоначально.

Динамическое выполнение программного кода

Самый простой способ выполнить выражение заключается в использовании встроенной функции `eval()`, с которой впервые мы встретились в главе 6. Например:

```
x = eval("(2 ** 31) - 1") # x == 2147483647
```

[1] Функция `glob.glob()` не обладает такими широкими возможностями, как, скажем, командная оболочка bash в UNIX, – хотя функция и поддерживает шаблонные символы `*`, `?` и синтаксическую конструкцию `[]`, но она не поддерживает синтаксис `{}`.

Такой способ отлично подходит для случая, когда выражение вводится пользователем, но как быть, если необходимо создать функцию динамически? Для этой цели можно использовать встроенную функцию `exec()`. Например, пользователь может ввести формулу, такую как $4\pi r^2$, и ее название – «area of sphere» (площадь поверхности шара), которую необходимо преобразовать в функцию. Предположим, что π мы заменили на `math.pi`; тогда функция, которую требуется создать, могла бы выглядеть, как показано ниже:

```
import math
code = '''
def area_of_sphere(r):
    return 4 * math.pi * r ** 2
'''
context = {}
context["math"] = math
exec(code, context)
```

Мы должны использовать надлежащие отступы, потому что указанный программный код должен соответствовать требованиям языка Python. (Хотя в данном случае мы могли бы записать весь программный код в одной строке, потому что блок функции состоит всего из одной строки.)

Если функции `exec()` в виде единственного аргумента передать некоторый программный код, у нас не будет возможности получить доступ к каким-либо функциям или переменным, созданным в результате выполнения этого программного кода. Кроме того, программный код, выполняемый функцией `exec()`, не имеет доступа к импортированным модулям, переменным функциям и к другим объектам, которые находятся в области видимости в момент вызова. Обе эти проблемы решаются посредством передачи словаря во втором аргументе. Словарь обеспечивает место, где будут сохраняться ссылки на объекты, которые будут доступны после того, как функция `exec()` вернет управление. Например, использование словаря `context` означает, что после вызова функции `exec()` в словаре появится ссылка на объект функции `area_of_sphere()`, созданной в результате вызова `exec()`. В данном примере нам необходимо, чтобы программный код, выполняемый функцией `exec()`, обладал доступом к модулю `math`, поэтому мы добавили в словарь `context` элемент, ключом которого является имя модуля, а значением – ссылка на объект модуля. Тем самым мы обеспечили доступность объекта `math.pi` для программного кода, выполняемого функцией `exec()`.

В некоторых случаях бывает удобно передать функции `exec()` весь глобальный контекст. Сделать это можно, использовав словарь, возвращаемый функцией `globals()`. Недостаток такого подхода состоит в том, что любые объекты, создаваемые вызовом функции `exec()`, будут добавлены в глобальный словарь. Решить эту проблему можно, скопировав глобальный контекст в словарь, например, `context = globals().copy()`.

Такой прием обеспечит программному коду, выполняемому функцией exec(), доступ ко всем импортированным модулям, переменным и другим объектам, имеющимся в области видимости, но любые изменения контекста, производимые в функции exec(), будут сохраняться в словаре context и не затронут глобальное окружение. (Может показаться, что надежнее было бы выполнять копирование с помощью функции copy.deepcopy(), но если проблема обеспечения безопасности стоит остро, то лучше вообще отказаться от использования функции exec().) Точно так же существует возможность передавать локальный контекст, например, передавая результат вызова функции locals() в третьем аргументе – она обеспечивает программному коду, выполняемому функцией exec(), доступ к объектам, созданным в локальной области видимости.

После вызова функции exec() словарь context будет содержать ключ "area_of_sphere", значением которого будет функция area_of_sphere(). Ниже показано, как можно получить доступ к этой функции и вызвать ее:

```
area_of_sphere = context["area_of_sphere"]
area = area_of_sphere(5) # area == 314.15926535897933
```

Объект area_of_sphere – это ссылка на объект функции, созданной динамически, которая может использоваться как любая другая функция. Несмотря на то, что в этом примере была создана единственная функция, тем не менее, в отличие от функции eval(), которая может интерпретировать единственное выражение, функция exec() может выполнять любое число инструкций языка Python, включая целые модули, как будет показано в следующем подразделе.

Динамическое импортирование

В языке Python имеются три простых механизма, которые могут использоваться для создания модулей расширения, причем все они связаны с импортированием модулей во время выполнения. Как только будет выполнено динамическое импортирование дополнительных модулей, можно с помощью функций интроспекции, входящих в состав языка Python, проверить доступность требуемых функциональных возможностей и задействовать их.

В этом подразделе мы рассмотрим программу *magic-numbers.py*. Эта программа считывает первые 1000 байтов из каждого файла, указанного в командной строке, и для каждого из них выводит его тип (или текст «Unknown» (тип неизвестен)) и имя. Ниже приводится пример командной строки и фрагмент вывода программы:

```
C:\Python30\python.exe magic-numbers.py c:\windows\*.*
...
XML.................c:\windows\WindowsShell.Manifest
Unknown.............c:\windows\WindowsUpdate.log
Windows Executable..c:\windows\winhelp.exe
```

```
Windows Executable..c:\windows\winhlp32.exe
Windows BMP Image...c:\windows\winnt.bmp
...
```

Программа пытается загрузить все модули, находящиеся в том же каталоге, что и программа, имя файла которых содержит слово «magic». Такие модули, как ожидается, содержат единственную общедоступную функцию с именем `get_file_type()`. В состав примеров к книге входят два очень простых модуля, *StandardMagicNumbers.py* и *WindowsMagicNumbers.py*, каждый из которых экспортирует функцию `get_file_type()`.

Мы будем рассматривать функцию `main()` программы, разделив ее на две части:

```
def main():
    modules = load_modules()
    get_file_type_functions = []
    for module in modules:
        get_file_type = get_function(module, "get_file_type")
        if get_file_type is not None:
            get_file_type_functions.append(get_file_type)
```

Вскоре мы увидим три различные реализации функции `load_modules()`, возвращающей (возможно, пустой) список объектов модулей, а затем рассмотрим функцию `get_function()`. Для каждого найденного модуля мы попробуем получить доступ к функции `get_file_type()` и добавим все такие функции в список.

```
    for file in get_files(sys.argv[1:]):
        fh = None
        try:
            fh = open(file, "rb")
            magic = fh.read(1000)
            for get_file_type in get_file_type_functions:
                filetype = get_file_type(magic,
                                         os.path.splitext(file)[1])
                if filetype is not None:
                    print("{0:.<20}{1}".format(filetype, file))
                    break
            else:
                print("{0:.<20}{1}".format("Unknown", file))
        except EnvironmentError as err:
            print(err)
        finally:
            if fh is not None:
                fh.close()
```

Этот цикл выполняет итерации по всем файлам, перечисленным в командной строке, и читает первые 1000 байтов из каждого. После этого он пытается вызвать по очереди каждую найденную функцию `get_file_type()`, чтобы определить тип текущего файла. Если имя файла

удается определить, на экран выводится информация о нем, внутренний цикл прерывается и выполняется переход к следующему файлу. Если тип файла определить не удалось или если не удалось найти ни одной функции `get_file_type()`, выводится текст «Unknown» (тип неизвестен).

Теперь рассмотрим три разных (но эквивалентных) способа динамического импортирования модулей, начав с самого длинного и самого сложного, поскольку в нем будет явно продемонстрирован каждый этап работы:

```
def load_modules():
    modules = []
    for name in os.listdir(os.path.dirname(__file__) or '.'):
        if name.endswith('.py') and 'magic' in name.lower():
            filename = name
            name = os.path.splitext(name)[0]
            if name.isidentifier() and name not in sys.modules:
                fh = None
                try:
                    fh = open(filename, 'r', encoding='utf8')
                    code = fh.read()
                    module = type(sys)(name)
                    sys.modules[name] = module
                    exec(code, module.__dict__)
                    modules.append(module)
                except (EnvironmentError, SyntaxError) as err:
                    sys.modules.pop(name, None)
                    print(err)
                finally:
                    if fh is not None:
                        fh.close()
    return modules
```

Функция начинает с того, что запускает итерации по всем файлам, находящимся в каталоге программы. Если это текущий каталог, функция `os.path.dirname(__file__)` вернет пустую строку, что вынудит функцию `os.listdir()` возбудить исключение; чтобы этого не произошло, в этом случае функции передается строка `'.'`. Из каждого имени файла-кандидата (который имеет расширение *.py* и в имени содержит текст «magic») функция получает имя модуля, отсекая расширение от имени файла. Если получившееся имя является допустимым идентификатором, следовательно, его можно рассматривать как имя модуля. Если это имя еще отсутствует в глобальном списке модулей, который предоставляет словарь `sys.modules`, производится попытка импортировать его.

После этого выполняется чтение текста из файла в строку `code`. Следующая строка, `module = type(sys)(name)` таит в себе одну хитрость. Когда вызывается функция `type()`, она возвращает объект типа указанного ей объекта. То есть, вызвав `type(1)`, мы получим `int`. Если попытать-

ся вывести объект типа, будет получено нечто удобочитаемое для человека, например, «int», но если вызвать объект типа как функцию, будет получен объект данного типа. Например, в переменную x можно записать целое число 5 с помощью инструкций x = 5, или x = int(5), или x = type(0)(5), или int_type = type(0); x = int_type(5). В нашем случае вызывается функция type(sys), где sys является модулем, поэтому функция возвращает объект типа для модуля (по сути то же самое, что и объект класса), который может использоваться для создания нового модуля с заданным именем. Точно так же, как и в примере с типом int, где не имело значения, какое число используется для получения объекта типа int, совершенно не важно, какой модуль будет использоваться (при условии, что он существует, то есть был импортирован) для получения объекта типа модуля.

После получения нового (пустого) модуля он добавляется в глобальный список модулей, чтобы предотвратить непреднамеренное повторное его импортирование. Это делается перед вызовом функции exec(), чтобы как можно ближе имитировать поведение инструкции import. Затем вызывается функция exec(), которая выполняет программный код, прочитанный из файла; при этом в качестве контекста используется словарь модуля. В конце полученный модуль добавляется в словарь модулей, которые мы будем использовать при определении типов файлов. Если возникли какие-либо проблемы, модуль удаляется из глобального словаря модулей (если он уже был туда добавлен), то есть модуль не будет добавлен в список модулей, если возникнет какая-либо ошибка. Обратите внимание, что функция exec() может обрабатывать любые объемы программного кода (тогда как функция eval() в состоянии обработать лишь единственное выражение – смотрите табл. 8.1) и возбуждает исключение SyntaxError в случае обнаружения синтаксической ошибки.

Ниже демонстрируется второй способ динамической загрузки модуля во время выполнения программы – программный код, показанный ниже, замещает первый вариант блоком try ... except:

```
try:
    exec("import " + name)
    modules.append(sys.modules[name])
except SyntaxError as err:
    print(err)
```

Одна из теоретических проблем, присущих этому варианту, заключается в его небезопасности. Переменная name может начинаться с подстроки sys;, за которой может следовать некоторый зловредный программный код.

Ниже демонстрируется третий вариант, который также замещает первый вариант блоком try ... except:

```
try:
    module = __import__(name)
```

```
        modules.append(module)
    except (ImportError, SyntaxError) as err:
        print(err)
```

Таблица 8.1. Функции динамического программирования и интроспекции

Синтаксис	Описание
`__import__(...)`	Импортирует модуль по его имени (подробности приводятся в тексте)
`compile(source, file, mode)`	Возвращает объект с программным кодом, полученным в результате компиляции исходного текста `source`; в аргументе `file` передается имя файла или `"<string>"`; аргумент `mode` может принимать одно из трех значений: `"single"`, `"eval"` или `"exec"`
`delattr(obj, name)`	Удаляет из объекта `obj` атрибут с именем `name`
`dir(`*`obj`*`)`	Возвращает список имен в локальной области видимости или, если определено значение аргумента *`obj`*, список имен атрибутов объекта *`obj`* (то есть имена его атрибутов и методов)
`eval(source, `*`globals`*`, `*`locals`*`)`	Возвращает результат вычисления единственного выражения `source`; если определены значения аргументов *`globals`* и *`locals`* (в виде словарей), они будут использованы как глобальный и локальный контекст соответственно
`exec(obj, `*`globals`*`, `*`locals`*`)`	Интерпретирует объект `obj`, который может быть строкой или объектом программного кода, полученным в результате вызова функции `compile()`, и возвращает `None`; если определены значения аргументов *`globals`* и *`locals`*, они будут использованы как глобальный и локальный контекст соответственно
`getattr(obj, name, `*`val`*`)`	Возвращает значение атрибута с именем `name`, принадлежащего объекту `obj`, или значение аргумента *`val`*, если оно определено и в объекте `obj` отсутствует указанный атрибут
`globals()`	Возвращает словарь текущего глобального контекста
`hasattr(obj, name)`	Возвращает `True`, если объект `obj` имеет атрибут с именем `name`
`locals()`	Возвращает словарь текущего локального контекста
`setattr(obj, name, val)`	Устанавливает значение `val` в атрибуте `name` объекта `obj`, создавая атрибут, если это необходимо
`type(obj)`	Возвращает объект типа для объекта `obj`
`vars(`*`obj`*`)`	Возвращает контекст объекта *`obj`* в виде словаря или локальный контекст, если аргумент *`obj`* не определен

Это самый простой способ динамического импортирования модулей, который несколько безопаснее, чем прямое использование функции `exec()`, хотя, как и в любом другом случае динамического импортирования, мы ничего не можем говорить о безопасности, потому что заранее неизвестно, какой программный код будет выполнен при импортировании модуля.

Хотя ни один из приемов, продемонстрированных здесь, не работает с пакетами или модулями в других каталогах, совсем несложно дополнить программный код для реализации этой возможности. Если же потребуется нечто более изощренное, следует обратиться к электронной документации, особенно к описанию функции `__import__()`.

Импортировав модуль, можно получить доступ к функциональности, предоставляемой им. Реализовать это можно с помощью встроенных функций интроспекции `getattr()` и `hasattr()`. Ниже показано, как они используются в реализации функции `get_function()`:

```
def get_function(module, function_name):
    function = get_function.cache.get((module, function_name), None)
    if function is None:
        try:
            function = getattr(module, function_name)
            if not hasattr(function, "__call__"):
                raise AttributeError()
            get_function.cache[module, function_name] = function
        except AttributeError:
            function = None
    return function
get_function.cache = {}
```

Пока не будем акцентировать внимание на программном коде, выполняющем действия с кэшем. Эта функция вызывает функцию `getattr()`, передавая ей имя модуля и имя ожидаемой функции. Если указанный атрибут отсутствует, будет возбуждено исключение `AttributeError`, но если такой атрибут имеется, используется функция `hasattr()`, с помощью которой определяется наличие атрибута `__call__` у данного атрибута – этот атрибут имеется у всех вызываемых объектов (то есть у функций и методов). (Далее мы познакомимся с более элегантным способом проверить, является ли объект вызываемым.) Если атрибут существует и является вызываемым, его можно вернуть вызывающей программе, в противном случае возвращается `None`, чтобы показать, что искомая функция недоступна.

Класс collections.Callable, стр. 453

Когда выполняется обработка нескольких сотен файлов (например, при использовании шаблона *.* в каталоге *C:\windows*), оказывается слишком затратно проверять наличие модуля в каждом файле. Поэтому

сразу вслед за заголовком определения функции `get_function()` к ней добавляется атрибут – словарь с именем `cache`. (Вообще говоря, язык Python позволяет добавлять любые атрибуты к любым объектам.) Когда функция `get_function()` вызывается в первый раз, словарь `cache` не содержит ни одного элемента, поэтому метод `dict.get()` вернет `None`. Но всякий раз, когда будет обнаруживаться подходящая функция, в словарь будет помещаться новый элемент, ключом которого является кортеж из двух элементов, с именами модуля и функции, а значением – сама функция. Поэтому при втором и последующих вызовах запрошенная функция будет возвращаться прямо из кэша, при этом поиск атрибута вообще не будет производиться.[1]

Методика, используемая в функции `get_function()` для кэширования возвращаемых значений с заданным набором аргументов, называется *запоминанием* (*memoizing*). Она может использоваться при реализации любой функции, не имеющей побочных эффектов (не изменяющей никаких глобальных переменных) и всегда возвращающей один и тот же результат при тех же (неизменных) значениях аргументов. Так как программный код, необходимый для создания и управления кэшем любой «запоминающей» функции, остается неизменным, он является прекрасным кандидатом на роль функции-декоратора; некоторые примеры декораторов `@memoize` приводятся в справочнике Python Cookbook на сайте *code.activestate.com/recipes/langs/python/*. Однако сами объекты модулей изменяемы, поэтому не все готовые к употреблению декораторы `@memoize` смогут работать с нашей функцией «как есть». Простое решение этой проблемы состоит в том, чтобы в составе кортежа, применяемого в качестве ключа, использовать не сам модуль, а значение атрибута `__name__` модуля.

Импортировать модули динамически очень просто, и также просто выполнять произвольный программный код на языке Python с помощью функции `exec()`. Это может быть очень удобно, например, когда программный код хранится в базе данных. Однако при этом отсутствует какой-либо контроль над импортируемым или выполняемым программным кодом. Вспомните, что, помимо дополнительных переменных, функций и классов, модули могут также содержать программный код, выполняемый при импортировании. Если такой программный код поступает из непроверенных источников, он может доставить массу неприятностей. Выбор решения зависит от конкретной ситуации – это не может быть поводом для беспокойства в определенных случаях или в личных проектах.

[1] В программе *magic-numbers.py* наряду с показанной здесь функцией `get_function()` присутствует более сложная ее реализация, которая эффективнее обрабатывает модули, в которых отсутствуют искомые функциональные возможности.

Локальные и рекурсивные функции

Часто бывает удобно иметь внутри функции одну-две вспомогательные функции. Язык Python позволяет делать это без лишних сложностей – достаточно просто объявить функцию внутри существующей функции. Такие функции часто называют *вложенными* или *локальными*. Мы уже видели примеры таких функций в главе 7.

Одна из типичных ситуаций использования локальных функций – когда необходимо организовать рекурсию. В этих случаях вызывается объемлющая функция, она выполняет все необходимые предварительные операции и производит первый вызов рекурсивной функции. Рекурсивными называются функции или методы, которые вызывают себя сами. Структурно все рекурсивные функции предусматривают два случая: *базовый случай* и *рекурсивный случай*. Базовый случай используется для прекращения рекурсии.

Рекурсивные функции могут оказаться весьма затратными в смысле потребления вычислительных ресурсов, потому что для каждого рекурсивного вызова создается новый кадр стека; тем не менее некоторые алгоритмы наиболее естественно реализуются с использованием рекурсии. В большинстве реализаций интерпретатора Python имеется ограничение на глубину возможных рекурсивных вызовов. Значение этого ограничения можно получить, обратившись к функции `sys.getrecursionlimit()`, и изменить его с помощью функции `sys.setrecursionlimit()`, хотя необходимость увеличения этого ограничения часто свидетельствует об использовании неподходящего алгоритма или об ошибке в реализации.

Классическим примером рекурсивной функции является функция вычисления факториала.[1] Например, вызов `factorial(5)` вычислит значение 5! и вернет число 120, то есть 1×2×3×4×5:

```
def factorial(x):
    if x <= 1:
        return 1
    return x * factorial(x - 1)
```

Это не самое эффективное решение, но оно наглядно демонстрирует две фундаментальные особенности рекурсивных функций. Если в аргументе `x` передается число 1 или меньше, функция возвращает 1 и рекурсии не возникает – это базовый случай. Но если значение аргумента `x` больше 1, возвращается значение `x * factorial(x - 1)`, и это уже рекурсивный случай, так как здесь функция вызывает саму себя. Эта функция гарантирует, что рано или поздно завершит свою работу, потому что в случае, когда значение `x` меньше или равно 1, выполняется базовый случай, который тут же завершает работу функции, а когда

[1] В модуле `math` имеется более эффективная функция вычисления факториала `math.factorial()`.

значение x больше 1, каждый рекурсивный вызов будет уменьшать это значение на 1, в результате чего оно рано или поздно достигнет значения 1.

Чтобы увидеть локальные и рекурсивные функции в осмысленном контексте, мы рассмотрим функцию indented_list_sort(), которая определена в файле модуля *IndentedList.py*. Эта функция принимает список строк, в котором отступы используются для обозначения иерархии, и строку, в которой хранится отступ на один уровень, а возвращает список с теми же строками, но отсортированными в алфавитном порядке без учета регистра символов, где элементы с отступами рекурсивно отсортированы в пределах своего родительского элемента. Результат работы функции показан на рис. 8.1, в списках before (до сортировки) и after (после сортировки).

Пусть дан список before, тогда список after — результат вызова: after = IndentedList.indented_list_sort(before). По умолчанию в качестве отступа на один уровень используются четыре пробела, такой же отступ используется в строках списка before, поэтому мы не будем явно указывать строку отступа при вызове функции.

```
before = ["Nonmetals",
          "    Hydrogen",
          "    Carbon",
          "    Nitrogen",
          "    Oxygen",
          "Inner Transitionals",
          "    Lanthanides",
          "        Cerium",
          "        Europium",
          "    Actinides",
          "        Uranium",
          "        Curium",
          "        Plutonium",
          "Alkali Metals",
          "    Lithium",
          "    Sodium",
          "    Potassium"]
```

```
after = ["Alkali Metals",
         "    Lithium",
         "    Potassium",
         "    Sodium",
         "Inner Transitionals",
         "    Actinides",
         "        Curium",
         "        Plutonium",
         "        Uranium",
         "    Lanthanides",
         "        Cerium",
         "        Europium",
         "Nonmetals",
         "    Carbon",
         "    Hydrogen",
         "    Nitrogen",
         "    Oxygen"]
```

***Рис. 8.1.** До и после сортировки списка, содержащего строки с отступами*

Сначала мы рассмотрим функцию indented_list_sort() в целом, а затем перейдем к двум ее локальным функциям.

```
def indented_list_sort(indented_list, indent="    "):
    KEY, ITEM, CHILDREN = range(3)
```

```
    def add_entry(level, key, item, children):
        ...

    def update_indented_list(entry):
        ...

    entries = []
    for item in indented_list:
        level = 0
        i = 0
        while item.startswith(indent, i):
            i += len(indent)
            level += 1
        key = item.strip().lower()
        add_entry(level, key, item, entries)

    indented_list = []
    for entry in sorted(entries):
        update_indented_list(entry)
    return indented_list
```

Функция начинается с создания трех констант, которые будут служить именами индексов, используемых локальными функциями. Затем определяются две локальные функции, которые будут рассмотрены чуть позже. Работа алгоритма сортировки делится на две стадии. На первой стадии создается список элементов, каждый из которых представлен трехэлементным кортежем, содержащим «ключ», используемый при сортировке, оригинальную строку и список дочерних элементов со строками. Ключ – это та же самая строка, в которой все символы приведены к нижнему регистру и удалены начальные и завершающие пробелы. В переменной `level` хранится текущий уровень отступа, для элементов верхнего уровня это значение 0, для элементов, дочерних по отношению к верхнему уровню это значение 1 и т. д. На второй стадии создается новый список, куда добавляются строки из отсортированного списка элементов верхнего уровня, строки дочерних элементов и т. д.

```
def add_entry(level, key, item, children):
    if level == 0:
        children.append((key, item, []))
    else:
        add_entry(level - 1, key, item, children[-1][CHILDREN])
```

Эта функция вызывается для каждой строки в списке. Аргумент `children` – это список, куда должны добавляться новые элементы. При вызове из внешней функции (in`dented_list_sort()`) ей передается список `entries`. Эта функция превращает список строк в список элементов, каждый из которых содержит строку верхнего уровня (без отступов) и (возможно, пустой) список дочерних элементов.

На уровне 0 (на самом верхнем уровне) в список `entries` добавляется новый кортеж из трех элементов. Он содержит ключ (для сортировки),

оригинальный элемент (который будет перемещен в список с результатами) и пустой список дочерних записей. Это базовый случай, так как здесь рекурсия не возникает. На других уровнях элемент `item` является дочерним по отношению к последнему элементу в списке `children`. В этом случае функция рекурсивно вызывает саму себя, уменьшая уровень на 1 и передавая дочерний список последнего элемента в списке `children`. На уровне 2 и выше выполняется несколько рекурсивных вызовов, пока наконец не будет достигнут уровень 0 и не будет получен нужный список для добавления элемента.

Например, когда дело доходит до строки «Inner Transitionals», внешняя функция вызывает функцию `add_entry()` со значением 0 в аргументе `level`, с ключом «inner transitionals», с элементом «Inner Transitionals» и списком `entries` в качестве списка дочерних записей. Поскольку текущим является уровень 0, новый элемент просто добавляется в список дочерних элементов (`entries`) с указанным ключом, элементом и пустым списком дочерних элементов. Следующая строка – « Lanthanides»; имеется отступ, следовательно, эта строка – дочерняя для строки «Inner Transitionals». Функция `add_entry()` вызывается со значением 1 в аргументе `level`, с ключом «lanthanides», с элементом « Lanthanides» и списком `entries` в качестве списка дочерних записей. Так как текущим является уровень 1, функция `add_entry()` рекурсивно вызовет саму себя со значением 0 (1 – 1) в аргументе `level`, с тем же самым ключом и элементом и со списком дочерних элементов, принадлежащим последнему элементу, то есть со списком дочерних элементов для элемента «Inner Transitionals».

Ниже показано, как выглядит список `entries` после добавления всех строк, но перед сортировкой:

```
[('nonmetals',
  'Nonmetals',
  [('hydrogen', '    Hydrogen', []),
   ('carbon', '    Carbon', []),
   ('nitrogen', '    Nitrogen', []),
   ('oxygen', '    Oxygen', [])]),
 ('inner transitionals',
  'Inner Transitionals',
  [('lanthanides',
    '    Lanthanides',
    [('cerium', '        Cerium', []),
     ('europium', '        Europium', [])]),
   ('actinides',
    '    Actinides',
    [('uranium', '        Uranium', []),
     ('curium', '        Curium', []),
     ('plutonium', '        Plutonium', [])])]),
 ('alkali metals',
  'Alkali Metals',
  [('lithium', '    Lithium', []),
```

```
    ('sodium', '    Sodium', []),
    ('potassium', '    Potassium', [])])]
```

Вывод списка был произведен с помощью функции `pprint.pprint()` из модуля `pprint` («pretty print» – модуль функций форматированного вывода). Обратите внимание, что список `entries` содержит всего три элемента (каждый из которых представлен трехэлементным кортежем) и в каждом из них последний элемент кортежа является списком дочерних трехэлементных кортежей (или пустым списком).

Функция `add_entry()` одновременно является и локальной и рекурсивной функцией. Подобно любой рекурсивной функции в ней предусматривается *базовый случай* действий (в этой функции он выполняется, когда текущим является уровень 0), завершающий рекурсию, и *рекурсивный случай*.

Эту функцию можно определить немного иначе:

```
def add_entry(key, item, children):
    nonlocal level
    if level == 0:
        children.append((key, item, []))
    else:
        level -= 1
        add_entry(key, item, children[-1][CHILDREN])
```

Здесь вместо того чтобы передавать значение `level` в качестве параметра, используется инструкция `nonlocal`, которая обеспечивает доступ к переменной в объемлющей области видимости. Если бы функция не изменяла переменную `level`, то инструкция `nonlocal` была бы не нужна, так как в этом случае интерпретатор, не обнаружив ее в локальной области видимости (во внутренней функции), продолжил бы поиски в объемлющей области видимости, где и нашел бы эту переменную. Но в этой версии функции `add_entry()` предусматривается изменение значения переменной `level`. Ранее мы использовали инструкцию `global`, чтобы сообщить интерпретатору, что подразумевается изменение глобальной переменной (чтобы предотвратить создание новой локальной переменной вместо изменения существующей глобальной переменной); это относится ко всем переменным, которые требуется изменить и которые находятся в одной из внешних областей видимости. Если в большинстве случаев использования инструкции `global` лучше просто избегать, то к использованию инструкции `nonlocal` следует относиться по крайней мере с осторожностью.

```
def update_indented_list(entry):
    indented_list.append(entry[ITEM])
    for subentry in sorted(entry[CHILDREN]):
        update_indented_list(subentry)
```

На первой стадии алгоритма создается список элементов, каждый из которых представлен кортежем из трех элементов (ключ, элемент,

список дочерних элементов), следующих в том же порядке, в каком они находятся в оригинальном списке. Во второй стадии, с пустым списком результата в начале, выполняются итерации через отсортированный список `entries`, и для каждого элемента вызывается функция `update_indented_list()`, которая выстраивает новый список с результатами. Функция `update_indented_list()` является рекурсивной. Она добавляет каждый элемент верхнего уровня в список `indented_list`, после чего вызывает саму себя, чтобы добавить все элементы из списка дочерних элементов. Добавив очередной дочерний элемент в список `indented_list`, функция вновь вызывает саму себя, чтобы добавить дочерние элементы этого дочернего элемента, и т. д. Базовый случай (когда прекращается рекурсия) наступает, когда элемент, или дочерний элемент, или дочерний элемент дочернего элемента (и так далее), не имеет дочерних элементов.

Интерпретатор пытается отыскать список `indented_list` в локальной области видимости (во внутренней функции) и не находит его; тогда он пытается отыскать список в объемлющей области видимости и находит его там. Но, обратите внимание, что во внутренней функции производится добавление элементов в список `indented_list`, хотя инструкция `nonlocal` при этом не использовалась. Это возможно потому, что инструкция `nonlocal` (как и инструкция `global`) применяется к ссылкам на объекты, а не к самим объектам, на которые они ссылаются. Во второй версии функции `add_entry()` мы использовали инструкцию `nonlocal` для переменной `level` потому, что применяемый оператор `+=` связывает ссылку на объект с новым объектом, то есть в действительности выполняется операция `level = level + 1`, поэтому в переменную `level` записывается ссылка на новый объект типа `int`. Но когда вызывается метод `list.append()` для списка `indented_list`, изменяется сам список, то есть повторного присваивания ссылки здесь не происходит, и потому нет необходимости в использовании инструкции `nonlocal`. (По тем же самым причинам, если у нас имеется словарь, список или другая глобальная коллекция, мы можем добавлять и удалять элементы коллекции без использования инструкции `global`.)

Декораторы функций и методов

Декораторы классов, стр. 438

Декоратор – это функция, которая принимает функцию или метод в качестве единственного аргумента и возвращает новую функцию или метод, включающую декорированную функцию или метод, с дополнительными функциональными возможностями. Нам уже приходилось использовать некоторые предопределенные декораторы, например, `@property` и `@classmethod`. В этом подразделе мы узнаем, как создавать собственные декораторы функций, а позднее в этой главе узнаем, как создавать декораторы классов.

Для первого примера декоратора предположим, что у нас имеется множество функций, выполняющих вычисления, и некоторые из них всегда должны возвращать положительный результат. Мы могли бы добавить в каждую из таких функций инструкцию `assert`, но использование декораторов проще и понятнее. Ниже приводится функция, декорированная декоратором `@positive_result`, который будет создан чуть ниже:

```
@positive_result
def discriminant(a, b, c):
    return (b ** 2) - (4 * a * c)
```

Благодаря декоратору, если функция вернет отрицательный результат, будет возбуждено исключение `AssertionError` и программа завершит работу. И конечно, этот декоратор можно применить к любому числу функций. Ниже приводится реализация декоратора:

```
def positive_result(function):
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        assert result >= 0, function.__name__ + "() result isn't >= 0"
        return result
    wrapper.__name__ = function.__name__
    wrapper.__doc__ = function.__doc__
    return wrapper
```

Декоратор определяет новую локальную функцию, которая вызывает оригинальную функцию. В данном случае объявляется локальная функция `wrapper()`. Она вызывает оригинальную функцию и запоминает результат, который используется в инструкции `assert`, проверяющей результат на положительность (или на необходимость завершить программу). Функция `wrapper()` просто возвращает результат, полученный от декорируемой функции. После создания функции `wrapper()` ее имя и строка документирования приводятся в соответствие с оригинальной функцией. Этим обеспечивается содействие механизму интроспекции, то есть в сообщениях об ошибках будет фигурировать имя оригинальной функции, а не функции `wrapper()`. Наконец декоратор возвращает функцию `wrapper()` — с этого момента она будет использоваться взамен оригинальной.

```
def positive_result(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        assert result >= 0, function.__name__ + "() result isn't >= 0"
        return result
    return wrapper
```

Выше приводится немного более понятная версия декоратора `@positive_result`. На этот раз функция `wrapper()` сама обертывается декоратором `@functools.wraps` из модуля `functools`, который гарантирует, что

функция `wrapper()` будет носить имя оригинальной функции и содержать ее строку документирования.

В некоторых случаях бывает удобно иметь параметризуемый декоратор, но, на первый взгляд, это кажется невозможным, так как декораторы принимают единственный аргумент – функцию или метод. У этой проблемы имеется замечательное решение. Мы можем вызвать функцию с требуемыми параметрами, и она вернет декоратор, который будет использован для декорирования следующей за ним функции. Например:

```
@bounded(0, 100)
def percent(amount, total):
    return (amount / total) * 100
```

Здесь функция `bounded()` вызывается с двумя аргументами и возвращает декоратор, который используется для декорирования функции `percent()`. Цель данного декоратора состоит в том, чтобы гарантировать, что возвращаемое значение всегда будет находиться в диапазоне от 0 до 100 включительно. Ниже приводится реализация функции `bounded()`:

```
def bounded(minimum, maximum):
    def decorator(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            result = function(*args, **kwargs)
            if result < minimum:
                return minimum
            elif result > maximum:
                return maximum
            return result
        return wrapper
    return decorator
```

Эта функция создает функцию-декоратор, которая в свою очередь создает функцию-обертку. Функция-обертка выполняет вычисления и возвращает результат, который гарантированно будет находиться в заданных пределах. Функция `decorator()` возвращает функцию `wrapper()`, а функция `bounded()` возвращает декоратор.

Следует отметить, что всякий раз, когда вызывается функция `bounded()`, внутри нее создается новый экземпляр функции-обертки, которая получает минимальное и максимальное значения, переданные при вызове `bounded()`.

Последний декоратор, который будет создан в этом разделе, имеет немного более сложную реализацию. Это функция регистрации, которая записывает имя, аргументы и результат любой декорируемой функции. Например:

```
@logged
def discounted_price(price, percentage, make_integer=False):
```

```
    result = price * ((100 - percentage) / 100)
    if not (0 < result <= price):
        raise ValueError("invalid price")
    return result if not make_integer else int(round(result))
```

Если интерпретатор выполняет программу в отладочном режиме (обычный режим), то при каждом вызове функции `discounted_price()` в файл *logged.log*, находящийся во временном каталоге, будет записываться сообщение, как показано в следующей выдержке из этого файла:

```
called: discounted_price(100, 10) -> 90.0
called: discounted_price(210, 5) -> 199.5
called: discounted_price(210, 5, make_integer=True) -> 200
called: discounted_price(210, 14, True) -> 181
called: discounted_price(210, -8) <type 'ValueError'>: invalid price
```

Если интерпретатор выполняет программу в оптимизированном режиме (используется ключ командной строки `-O` или переменная окружения `PYTHONOPTIMIZE` содержит значение `-O`[1]), то регистрация отключается. Ниже приводится программный код, выполняющий настройку механизма регистрации и определение самого декоратора:

```
if __debug__:
    logger = logging.getLogger("Logger")
    logger.setLevel(logging.DEBUG)
    handler = logging.FileHandler(os.path.join(
                                tempfile.gettempdir(), "logged.log"))
    logger.addHandler(handler)

    def logged(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            log = "called: " + function.__name__ + "("
            log += ", ".join(["{0!r}".format(a) for a in args] +
                             ["{0!s}={1!r}".format(k, v)
                              for k, v in kwargs.items()])
            result = exception = None
            try:
                result = function(*args, **kwargs)
                return result
            except Exception as err:
                exception = err
            finally:
                log += (") -> " + str(result)) if exception is None
                        else ") {0}: {1}".format(type(exception),
                                                 exception))
            logger.debug(log)
            if exception is not None:
                raise exception
```

[1] В обоих случаях это буква `O`, а не цифра `0`. – *Прим. перев.*

```
        return wrapper
else:
    def logged(function):
        return function
```

При работе в отладочном режиме переменная `__debug__` имеет значение `True`. В этом случае выполняется настройка механизма регистрации с использованием модуля `logging`, и затем создается декоратор `@logged`. Модуль `logging` обладает очень широкими возможностями – он может записывать сообщения в файлы, выполнять ротацию файлов, отправлять сообщения по электронной почте, через сетевые соединения, серверам HTTP и многое другое. Здесь задействованы только самые основные средства – создается объект-регистратор, устанавливается уровень регистрации (поддерживается несколько уровней) и в качестве устройства вывода выбирается файл.

Генераторы словарей, стр. 160

Функция-обертка начинает с того, что создает текст сообщения, включив в него имя функции и значения аргументов. После этого предпринимается попытка вызвать функцию и сохранить результат. Если возникло какое-либо исключение, оно сохраняется. В любом случае выполняется блок `finally`, и здесь в текст сообщения для регистрации добавляется результат (или исключение), после чего производится вывод сообщения в устройство регистрации. Если никаких исключений не возникло, результат возвращается вызывающей программе; в противном случае повторно возбуждается исключение, имитируя поведение оригинальной функции.

При работе в оптимизированном режиме переменная `__debug__` принимает значение `False`. В этом случае используется определение функции `logged()`, которая просто возвращает указанную ей функцию, поэтому, кроме некоторой крошечной задержки, обусловленной созданием функции, во время выполнения никакого снижения производительности не наблюдается.

Обратите внимание, что в стандартной библиотеке присутствуют модули `trace` и `profile`, которые могут запускать и анализировать ход выполнения программ и модулей, а также воспроизводить различные отчеты трассировки и профилирования. Оба они используют механизмы интроспекции, поэтому, в отличие от декоратора `@logged`, ни модуль `trace`, ни модуль `profile` не требуют вносить изменения в исходные тексты.

Аннотации функций

Функции и методы могут определяться с помощью аннотаций – выражений, которые могут использоваться в сигнатурах функций. Ниже приводится общий синтаксис:

```
def functionName(par1 : exp1, par2 : exp2, ..., parN : expN) -> rexp:
    suite
```

Каждое выражение, следующее за двоеточием (*: expX*), является необязательным, как и выражение возвращаемого значения, следующее за стрелкой (*-> rexp*). Последний (или единственный) позиционный параметр (если таковой имеется) может иметь форму **args*, с аннотацией или без. Точно так же последний (или единственный) именованный параметр (если таковой имеется), может иметь форму ***kwargs*, и тоже с аннотацией или без.

Если аннотации присутствуют в заголовке функции, они добавляются в словарь `__annotations__` этой функции. Если аннотации отсутствуют, словарь остается пустым. Ключами словаря служат имена параметров, а значениями – соответствующие им выражения. Синтаксис допускает возможность аннотировать все параметры, некоторые из них или ни одного и, кроме того, аннотировать или не аннотировать возвращаемое значение. Аннотации не имеют специального значения для интерпретатора. Единственное, что интерпретатор делает, когда встречает аннотации, – помещает их в словарь `__annotations__`, оставляя за нами любые действия с ними. Ниже приводится пример аннотированной функции из модуля `Util`:

```
def is_unicode_punctuation(s : str) -> bool:
    for c in s:
        if unicodedata.category(c)[0] != "P":
            return False
    return True
```

Каждый символ Юникода принадлежит какой-то конкретной категории, а каждая категория идентифицируется идентификатором из двух символов. Все категории, имена которых начинаются с символа *P*, содержат знаки пунктуации.

В данном примере в качестве выражений аннотации мы использовали имена типов данных языка Python. Но они не имеют никакого значения для интерпретатора, что наглядно показывают следующие вызовы функции:

```
Util.is_unicode_punctuation("zebr\a")   # вернет: False
Util.is_unicode_punctuation(s="!@#?")   # вернет: True
Util.is_unicode_punctuation(("!", "@")) # вернет: True
```

В первом вызове используется позиционный аргумент, а во втором – именованный, просто для демонстрации, что оба варианта работают так, как и ожидается. В последнем вызове вместо строки передается кортеж, и это вполне допустимо, потому что интерпретатор никак не учитывает аннотации, кроме как записывает их в словарь `__annotations__`.

Если мы хотим извлечь толк из аннотаций, чтобы, например, выполнить проверку типов, можно предусмотреть декорирование требуемой

функции соответствующим декоратором. Ниже приводится очень простой декоратор, выполняющий проверку типов:

```
def strictly_typed(function):
    annotations = function.__annotations__
    arg_spec = inspect.getfullargspec(function)

    assert "return" in annotations, "missing type for return value"
    for arg in arg_spec.args + arg_spec.kwonlyargs:
        assert arg in annotations, ("missing type for parameter `" +
                                    arg + "`")
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        for name, arg in (list(zip(arg_spec.args, args)) +
                          list(kwargs.items())):
            assert isinstance(arg, annotations[name]), (
                    "expected argument '{0}' of {1} got {2}".format(
                    name, annotations[name], type(arg)))
        result = function(*args, **kwargs)
        assert isinstance(result, annotations["return"]), (
                    "expected return of {0} got {1}".format(
                    annotations["return"], type(result)))
        return result
    return wrapper
```

Данный декоратор требует, чтобы все аргументы и возвращаемое значение были аннотированы соответствующими типами данных. Он проверяет наличие в указанной функции аннотаций с типами для всех аргументов и возвращаемого значения и во время выполнения проверяет соответствие фактических аргументов ожидаемым типам данных.

Модуль `inspect` содержит мощные средства интроспекции для объектов. Здесь мы использовали лишь малую часть спецификаций аргументов, возвращаемых модулем, извлекая имена всех позиционных и именованных аргументов – в правильном порядке следования в случае позиционных аргументов. Затем эти имена и словарь с аннотациями используются для проверки наличия аннотации у каждого параметра и возвращаемого значения.

Функция-обертка, созданная внутри декоратора, сначала выполняет итерации по всем парам имя-аргумент для всех позиционных и именованных аргументов. Так как функция `zip()` возвращает итератор, а метод `dict.items()` возвращает представление словаря, мы не можем объединить их непосредственно, поэтому сначала каждый из них преобразуется в список. Если тип фактического аргумента отличается от типа, указанного в аннотации, инструкция `assert` терпит неудачу; в противном случае вызывается фактическая функция, после чего выполняется проверка типа возвращаемого значения, и если это значение имеет требуемый тип, оно возвращается вызывающей программе. В конце функция `strictly_typed()` как обычно возвращает функцию-обертку.

Обратите внимание, что проверка выполняется только в отладочном режиме (который является режимом выполнения по умолчанию и задается ключом командной строки `-O` или переменной окружения `PYTHONOPTIMIZE`).

Если функцию `is_unicode_punctuation()` декорировать декоратором `@strictly_typed` и попытаться выполнить те же вызовы, что и прежде, но уже для декорированной версии, то аннотации вступят в силу, как показано ниже:

```
is_unicode_punctuation("zebr\a")   # вернет: False
is_unicode_punctuation(s="!@#?")   # вернет: True
is_unicode_punctuation(("!", "@")) # возбудит исключение AssertionError
```

Теперь проверка типов аргументов выполняется, поэтому в последнем случае возбуждается исключение `AssertionError`, так как кортеж не является строкой или подклассом класса `str`.

Теперь рассмотрим совершенно иное применение аннотаций. Ниже приводится маленькая функция, которая повторяет функциональность встроенной функции `range()`, за исключением того, что она всегда возвращает число с плавающей точкой.

```
def range_of_floats(*args) -> "author=Reginald Perrin":
    return (float(x) for x in range(*args))
```

Сама функция никак не использует аннотацию, но совсем несложно создать инструмент, который будет импортировать все модули проекта и выводить список функций с именами авторов, извлекая имена функций из атрибута `__name__`, а имена авторов – из элемента словаря `__annotations__` с ключом `"return"`.

Аннотации – это совершенно новая особенность языка Python, и языком не предусматривается какого-то предопределенного назначения для них, поэтому область применения аннотаций ограничивается только воображением программиста. С идеями, касающимися возможного использования аннотаций, можно ознакомиться в предложении по расширению PEP 3107 «Function Annotations» по адресу: *www.python.org/dev/peps/pep-3107*; там же можно найти несколько полезных ссылок.

Улучшенные приемы объектно-ориентированного программирования

В этом разделе мы более подробно рассмотрим поддержку объектно-ориентированного программирования в языке Python. Познакомимся со множеством приемов, которые помогают уменьшить объем программного кода, а также с приемами, расширяющими существующие возможности программирования. Но сначала мы рассмотрим одну новую, очень маленькую и очень простую особенность. Ниже приводится

начало определения класса Point, обладающего теми же возможностями, что и версия этого класса из главы 6:

```
class Point:

    __slots__ = ("x", "y")

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

Возможность доступа к атрибутам функций, стр. 406

Когда класс создается без использования частной переменной __slots__, для каждого экземпляра класса интерпретатор создает словарь с именем __dict__, и этот словарь используется для хранения атрибутов данных экземпляра, поэтому имеется возможность добавлять и удалять атрибуты объектов. (Например, благодаря этой особенности мы смогли добавить атрибут cache к функции get_function() ранее в этой главе.)

Если нам требуется, чтобы объект просто лишь обеспечивал доступ к своим атрибутам и не позволял добавлять или удалять атрибуты, можно создать класс, экземпляры которого не будут иметь словарь __dict__. Этого легко добиться, просто определив атрибут класса с именем __slots__, значением которого является кортеж с именами атрибутов. Каждый объект такого класса будет иметь атрибуты с указанными именами, и в них будет отсутствовать словарь __dict__. Объекты таких классов не допускают возможность добавления или удаления атрибутов. По сравнению с обычными объектами такие объекты занимают меньший объем памяти, хотя это вряд ли имеет большое значение в программах, которые не создают большого числа объектов.

Управление доступом к атрибутам

Иногда бывает удобно иметь в классе такие атрибуты, значения которых не хранятся в памяти, а вычисляются в момент обращения к ним. Ниже приводится полная реализация такого класса:

```
class Ord:

    def __getattr__(self, char):
        return ord(char)
```

Имея класс Ord, можно создать экземпляр этого класса ord = Ord() и получить альтернативу встроенной функции ord(), работающей с любыми символами, допустимыми для использования в идентификаторах. Например, обращение к атрибуту ord.a вернет число 97, ord.Z вернет 90, а ord.e вернет 229. (Но обращение к атрибуту ord.! и подобным ему будет вызывать синтаксическую ошибку.)

Обратите внимание, что если ввести определение класса Ord в среде IDLE, он не будет работать, если выполнить выражение ord = Ord(). Это

обусловлено тем, что экземпляр класса имеет то же имя, что и встроенная функция ord(), которая используется методом класса Ord. В этом случае вызов ord() будет интерпретироваться как попытка вызова экземпляра ord, что будет вызывать исключение TypeError. Эта проблема не проявляется при импортировании модуля с определением класса Ord, потому что в этом случае экземпляр ord, создаваемый в интерактивной оболочке, и функция ord(), используемая классом Ord, будут находиться в разных модулях, и потому не произойдет замещения одного другим. Если же действительно необходимо создать этот класс в интерактивной оболочке и использовать в нем встроенную функцию, это можно реализовать, вынудив класс вызывать именно встроенную функцию – в данном случае, импортировав модуль builtins, обеспечивающий однозначный доступ ко всем встроенным функциям, и вызывая встроенную функцию как builtins.ord(), а не просто ord().

Ниже приводится другой пример небольшого, но законченного класса. Он позволяет создавать «константы». Даже при использовании этого класса совсем несложно изменить значение такой «константы», но он хотя бы предотвращает самые простые ошибки:

```
class Const:

    def __setattr__(self, name, value):
        if name in self.__dict__:
            raise ValueError("cannot change a const attribute")
        self.__dict__[name] = value

    def __delattr__(self, name):
        if name in self.__dict__:
            raise ValueError("cannot delete a const attribute")
        raise AttributeError("'{0}' object has no attribute '{1}'"
                             .format(self.__class__.__name__, name))
```

С помощью этого класса можно создавать объекты констант, скажем, так: const=Const(), и устанавливать любые их атрибуты, какие только потребуется, например, const.limit = 591. Но, как только значение атрибута будет установлено, оно будет доступно только для чтения – любые попытки изменить или удалить атрибут будут возбуждать исключение ValueError. Мы не стали переопределять метод __getattr__(), потому что метод object.__getattr__() базового класса реализует все, что нам необходимо, – возвращает значение требуемого атрибута или возбуждает исключение AttributeError, если указанный атрибут отсутствует. В методе __delattr__() имитируется сообщение об ошибке, которое выводится методом __getattr__() при попытке обратиться к несуществующему атрибуту, для чего нам потребовалось получить имя класса и имя несуществующего атрибута. Работа класса основана на использовании атрибута __dict__ объекта, который также используется следующими методами базового класса: __getattr__(), __setattr__() и __delattr__(); в данном случае мы используем только метод __get-

attr__() базового класса. Все специальные методы, используемые для доступа к атрибутам, перечислены в табл. 8.2.

Таблица 8.2. Специальные методы доступа к атрибутам

Синтаксис	Используется	Описание
`__delattr__(self, name)`	`del x.n`	Удаляет атрибут `n` из объекта `x`
`__dir__(self)`	`dir(x)`	Возвращает список имен атрибутов объекта `x`
`__getattr__(self, name)`	`v = x.n`	Возвращает значение атрибута `n` объекта `x`, если он существует
`__getattribute__(self, name)`	`v = x.n`	Возвращает значение атрибута `n` объекта `x`; подробности в тексте
`__setattr__(self, name, value)`	`x.n = v`	Присваивает значение `v` атрибуту `n` объекта `x`

Существует еще один способ реализации констант – с использованием именованных кортежей. Ниже приводится пара примеров:

```
Const = collections.namedtuple("_", "min max")(191, 591)
Const.min, Const.max                          # вернет: (191, 591)
Offset = collections.namedtuple("_", "id name description")(*range(3))
Offset.id, Offset.name, Offset.description  # вернет: (0, 1, 2)
```

В обоих случаях мы использовали ничего не значащее имя для именованного кортежа, потому что каждый раз нам необходим лишь один экземпляр кортежа, а не подкласс, который мог бы использоваться для создания нескольких экземпляров именованных кортежей. Язык Python не поддерживает такой тип данных, как перечисления, тем не менее мы можем использовать именованные кортежи для достижения того же эффекта.

Класс Image.py, стр. 306

Заканчивая рассмотрение специальных методов доступа к атрибутам, вернемся к примеру, который первый раз был продемонстрирован в главе 6. В этой главе мы создали класс `Image`, который имел фиксированные ширину, высоту и цвет фона, задававшиеся в момент создания экземпляра `Image` (и которые могли изменяться при загрузке изображения из файла). Мы обеспечили доступ к этим атрибутам с помощью свойств, доступных только для чтения. Например:

```
@property
def width(self):
    return self.__width
```

Такой способ отличается простотой, но он может оказаться утомительным, если потребуется реализовать достаточно много свойств, доступных только для чтения. Ниже приводится другое решение, которое за-

ключается в обслуживании всех свойств класса Image, доступных только для чтения:

```
def __getattr__(self, name):
    if name == "colors":
        return set(self.__colors)
    classname = self.__class__.__name__
    if name in frozenset({"background", "width", "height"}):
        return self.__dict__["_{0}__{1}".format(classname, name)]
    raise AttributeError("'{0}' object has no attribute '{1}'"
                         .format(classname, name))
```

Если попытаться обратиться к атрибуту объекта и атрибут не будет обнаружен, интерпретатор вызовет метод __getattr__() (при условии, что класс предоставляет его реализацию и не был переопределен метод __getattribute__()) с именем атрибута в качестве параметра. Реализация метода __getattr__() должна возбуждать исключение AttributeError, если она не обслуживает указанный атрибут.

Например, если в программе производится обращение к атрибуту image.colors и интерпретатор не находит его, будет произведен вызов метода Image.__getattr__(image, "colors"). В данном случае метод __getattr__() обслужит атрибут с именем "colors" и вернет копию множества цветов, использующихся в изображении.

Другие атрибуты являются неизменяемыми объектами, поэтому возврат прямых ссылок на эти атрибуты не таит в себе никакой угрозы. Для каждого атрибута можно было бы предусмотреть отдельную инструкцию elif, как показано ниже:

```
elif name == "background":
    return self.__background
```

Но вместо этого мы использовали более компактное решение. Зная, что все неспециальные атрибуты объекта хранятся в словаре self.__dict__, мы предпочли обращаться к ним напрямую. Для частных атрибутов (имена которых начинаются с двух символов подчеркивания) производится приведение их имен к форме *_className__attributeName*, и мы должны учитывать это обстоятельство при извлечении значений атрибутов из частного словаря объекта.

Чтобы выполнить приведение имени при поиске частного атрибута и воссоздать корректный текст при возбуждении исключения AttributeError, нам необходимо знать имя класса, которому принадлежит метод. (Это может быть не класс Image, потому что объект может оказаться экземпляром подкласса, наследующего класс Image.) Каждый объект имеет специальный атрибут __class__, поэтому внутри методов всегда можно обратиться к атрибуту self.__class__, не рискуя попасть в бесконечную рекурсию.

Обратите внимание на тонкое различие: при использовании метода __getattr__() и выражения self.__class__ обеспечивается доступ к ат-

рибуту экземпляра класса (который может быть подклассом), а при прямом обращении к атрибуту используется класс, в котором этот атрибут был определен.

Нам осталось рассмотреть еще один специальный метод – метод `__getattribute__()`. Если метод `__getattr__()` вызывается в последнюю очередь, когда выполняется поиск (неспециальных) атрибутов, то метод `__getattribute__()` вызывается в первую очередь, при каждом обращении к любому атрибуту. Хотя в определенных случаях метод `__getattribute__()` может оказаться не только полезным, но и необходимым, тем не менее переопределение этого метода может оказаться непростым делом. При переопределении особое внимание следует уделять тому, чтобы исключить возможность рекурсивного вызова – избежать рекурсии в таких случаях часто удается с помощью вызовов `super().__getattribute__()` или `object.__getattribute__()`. Кроме того, поскольку метод `__getattribute__()` вызывается при обращении к любому атрибуту, его переопределение легко может привести к потере производительности по сравнению с прямым доступом к атрибутам или свойствам. Ни один из классов, которые приводятся в этой книге, не переопределяет этот метод.

Функторы

В языке Python *объектами функций* являются ссылки на любые вызываемые объекты, такие как функции, лямбда-функции или методы. Под это определение также подпадают и классы, поскольку ссылки на классы можно вызывать как функции, которые при вызове возвращают объект данного класса, например `x = int(5)`. В информатике *функтором* называется объект, который может вызываться, как если бы он был функцией, поэтому в терминах языка Python функтор является разновидностью объекта функции. Любой класс, имеющий специальный метод `__call__()`, является функтором. Главное достоинство функторов заключается в том, что они могут поддерживать некоторую информацию о состоянии. Например, можно создать функтор, который всегда удаляет основные знаки пунктуации с обоих концов строки, как показано ниже:

```
strip_punctuation = Strip(",;:.!?")
strip_punctuation("Land ahoy!")      # вернет: 'Land ahoy'
```

Здесь создается экземпляр функтора `Strip`, инициализированный значением `",;:.!?"`. Всякий раз, когда будет вызываться экземпляр этого функтора, он будет возвращать полученную строку с отброшенными знаками пунктуации. Ниже приводится полная реализация класса `Strip`:

```
class Strip:

    def __init__(self, characters):
        self.characters = characters

    def __call__(self, string):
        return string.strip(self.characters)
```

Того же эффекта можно было бы добиться с помощью простой функции или лямбда-функции, но когда необходимо хранить чуть больше информации о состоянии или выполнять более сложную обработку, часто правильным решением будет использование функтора.

Способность функтора сохранять информацию о состоянии с помощью класса обеспечивает ему высокую гибкость и чрезвычайно широкие возможности, но иногда такая гибкость и широта оказываются излишними. Существует еще один способ сохранять информацию о состоянии, который заключается в использовании *замыканий*. Замыкание – это функция или метод, которые запоминают некоторое состояние. Например:

```
def make_strip_function(characters):
    def strip_function(string):
        return string.strip(characters)
    return strip_function

strip_punctuation = make_strip_function(",;:.!?")
strip_punctuation("Land ahoy!")       # вернет: 'Land ahoy'
```

Функция `make_strip_function()` принимает в качестве единственного аргумента строку символов, которые следует удалять, и возвращает функцию `strip_function()`, принимающую строковый аргумент и удаляющую из него символы, полученные в момент создания замыкания. Мы можем создать произвольное число экземпляров класса `Strip`, каждый со своим набором удаляемых символов, и точно так же мы можем создать произвольное число замыканий, каждое со своим набором символов.

Классическим примером использования функторов может служить ключевая функция, применяемая при сортировке. Ниже приводится универсальный класс функтора `SortKey` (из файла *SortKey.py*):

```
class SortKey:

    def __init__(self, *attribute_names):
        self.attribute_names = attribute_names

    def __call__(self, instance):
        values = []
        for attribute_name in self.attribute_names:
            values.append(getattr(instance, attribute_name))
        return values
```

Когда создается объект SortKey, он сохраняет кортеж с именами атрибутов, с которыми он был инициализирован. Когда производится вызов объекта, создается список значений атрибутов для заданного экземпляра, следующих в том же порядке, в каком они были указаны при инициализации объекта SortKey. Например, представим, что у нас имеется класс Person:

```
class Person:

    def __init__(self, forename, surname, email):
        self.forename = forename
        self.surname = surname
        self.email = email
```

Допустим, что у нас имеется список people объектов Person. Тогда этот список можно отсортировать по фамилиям людей следующим способом: people.sort(key=SortKey("surname")). Если в списке присутствуют одинаковые фамилии, то можно отсортировать список сначала по фамилиям, а потом по именам: people.sort(key=SortKey("surname", "forename")). А если в списке присутствует набор одинаковых фамилий и имен, можно включить в сортировку еще и адреса электронной почты. Безусловно, точно так же можно было бы отсортировать список сначала по именам, а потом по фамилиям, достаточно лишь изменить порядок следования атрибутов, передаваемых функтору SortKey.

Другой способ добиться того же эффекта, но вообще без создания функтора, заключается в использовании функции operator.attrgetter() из модуля operator. Например, сортировку списка по фамилиям можно было бы выполнить так: people.sort(key=operator.attrgetter("surname")). А сортировку по фамилиям и именам так: people.sort(key=operator.attrgetter("surname", "forename")). Функция operator.attrgetter() возвращает функцию (замыкание), при обращении в контексте объекта возвращающую атрибуты объекта, имена которых были указаны при создании замыкания.

В языке Python функторы используются реже, чем в других языках программирования, поддерживающих такую возможность, потому что в языке Python имеются другие средства достижения того же эффекта, например, замыкания и функции доступа к атрибутам.

Менеджеры контекста

Менеджеры контекста позволяют упростить программный код, гарантируя выполнение определенных операций до и после выполнения некоторого блока программного кода. Такое поведение обусловлено тем, что менеджеры контекста определяют два специальных метода – __enter__() и __exit__(), которые интерпретируются особым образом в области видимости инструкции with. Когда с помощью инструкции with создается менеджер контекста, автоматически вызывается его ме-

тод `__enter__()`, а когда поток выполнения покидает область видимости менеджера контекста, автоматически вызывается его метод `__exit__()`.

У нас имеется возможность создавать свои собственные менеджеры контекста или использовать предопределенные – как будет показано ниже в этом подразделе объекты файлов, возвращаемые встроенной функцией `open()`, являются менеджерами контекста. Ниже приводится синтаксис использования менеджера контекста:

```
with expression as variable:
    suite
```

Выражение *expression* должно быть менеджером контекста или воспроизводить его. Если в инструкции указана необязательная часть `as` *variable*, в переменную *variable* записывается ссылка на объект, возвращаемый методом `__enter__()` менеджера контекста (зачастую это сам менеджер контекста). Поскольку менеджеры контекста гарантируют вызов метода `__exit__()` (даже в случае исключений), во многих ситуациях они могут использоваться для устранения блоков `finally`.

Некоторые типы данных в языке Python являются менеджерами контекста, например, все объекты файлов, создаваемых функцией `open()`; поэтому у нас имеется возможность отказаться от использования блоков `finally` при работе с файлами, как это показано в следующих двух эквивалентных фрагментах (если исходить из предположения, что где-то в другом месте присутствует определение функции `process()`):

```
fh = None
try:
    fh = open(filename)
    for line in fh:
        process(line)
except EnvironmentError as err:
    print(err)
finally:
    if fh is not None:
        fh.close()
```

```
try:
    with open(filename) as fh:
        for line in fh:
            process(line)
except EnvironmentError as err:
    print(err)
```

Объект файла является менеджером контекста, реализация метода `__exit__()` которого всегда закрывает файл, если он был открыт. Метод `__exit__()` будет выполняться независимо от того, возникло исключение или нет, но во втором случае исключение продолжит свое распространение вверх по стеку возвратов. Эта особенность гарантирует, что файл будет закрыт и у нас останется возможность перехватить и обработать любую ошибку, в данном случае – вывести сообщение.

В действительности менеджеры контекстов не обязаны обеспечивать дальнейшего распространения исключений, но это привело бы к сокрытию любых исключений, что почти всегда оказалось бы программной ошибкой. Все встроенные менеджеры контекста и менеджеры

контекста из стандартной библиотеки обеспечивают дальнейшее распространение исключений.

Иногда бывает необходимо одновременно использовать два или более менеджеров контекста. Например:

```
try:
    with open(source) as fin:
        with open(target, "w") as fout:
            for line in fin:
                fout.write(process(line))
except EnvironmentError as err:
    print(err)
```

Здесь выполняется чтение строк из исходного файла и запись обработанных строк в выходной файл.

Использование вложенных инструкций `with` может быстро привести к непомерному увеличению отступов. К счастью, модуль `contextlib` из стандартной библиотеки предоставляет дополнительную поддержку менеджеров контекста, включая функцию `context.nest()`, которая позволяет обрабатывать два или более менеджеров контекста одной инструкцией `with`. Ниже приводится видоизмененная версия программного кода, который только что был продемонстрирован, в которой мы опустили строки, оставшиеся без изменений:

```
try:
    with contextlib.nested(open(source), open(target, "w")) as (
            fin, fout):
        for line in fin:
```

Многопоточная модель выполнения, стр. 467

Менеджерами контекста являются не только объекты файлов. Например, некоторые классы, связанные с реализацией многопоточной модели выполнения, используют менеджеры контекста для установки блокировок. Менеджеры контекста могут использоваться также с числами `decimal.Decimal`, что очень удобно при реализации вычислений с определенными параметрами (например, с различной точностью).

Если возникает необходимость реализовать собственный менеджер контекста, следует создать класс, предоставляющий два метода: `__enter__()` и `__exit__()`. Всякий раз, когда инструкция `with` будет применяться к экземпляру такого класса, интерпретатор автоматически будет вызывать его метод `__enter__()`, а возвращаемое им значение будет присваиваться переменной в части `as` *`variable`* (или просто отбрасываться, если переменная не указана). Когда поток выполнения будет покидать область видимости инструкции `with`, интерпретатор будет вызывать метод `__exit__()` (с информацией об исключении, если оно возникло, в виде аргумента).

Предположим, что нам требуется выполнить некоторые операции над списком в атомарном режиме, то есть либо все операции должны быть выполнены, либо ни одна из них, чтобы получившийся список всегда находился в предсказуемом состоянии. Например, допустим, что у нас имеется список целых чисел и нам требуется добавить одно число, удалить одно число и изменить пару чисел, причем все это должно быть выполнено как единая операция. Реализовать это можно следующим способом:

```
try:
    with AtomicList(items) as atomic:
        atomic.append(58289)
        del atomic[3]
        atomic[8] = 81738
        atomic[index] = 38172
except (AttributeError, IndexError, ValueError) as err:
    print("no changes applied:", err)
```

Если в ходе выполнения операций никаких исключений не возникло, все операции будут применены к оригинальному списку (`items`), но если возникло исключение, список останется без изменений. Ниже приводится реализация менеджера контекста `AtomicList`:

```
class AtomicList:

    def __init__(self, alist, shallow_copy=True):
        self.original = alist
        self.shallow_copy = shallow_copy

    def __enter__(self):
        self.modified = (self.original[:] if self.shallow_copy
                         else copy.deepcopy(self.original))
        return self.modified

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is None:
            self.original[:] = self.modified
```

При создании объекта `AtomicList` мы сохраняем ссылку на оригинальный список. Обратите внимание на флаг, который определяет, какое копирование будет применяться к списку – поверхностное или глубокое. (Поверхностное копирование прекрасно подходит для списков чисел или строк, но если список содержит другие списки или другие коллекции, поверхностного копирования будет недостаточно.)

Поверхностное и глубокое копирование, стр. 173

Затем, когда менеджер контекста `AtomicList` используется в инструкции `with`, вызывается его метод `__enter__()`. В этот момент создается и возвращается копия списка, чтобы все изменения выполнялись в копии.

По достижении конца области видимости инструкции `with` вызывается метод `__exit__()`. Если в процессе работы исключений не возникло, аргумент `exc_type` («exception type» – тип исключения) будет содержать значение `None`, откуда следует, что можно безопасно заместить элементы оригинального списка элементами модифицированного списка. (Здесь нельзя просто использовать инструкцию `self.original = self.modified`, потому что она просто заменит одну ссылку на объект другой ссылкой на объект и не окажет никакого воздействия на оригинальный список.) Но если было возбуждено исключение, метод ничего не делает с оригинальным списком, а модифицированный список просто уничтожается.

Возвращаемое значение метода `__exit__()` используется интерпретатором, чтобы определить, следует ли продолжить распространение исключения, если оно возникло. Значение `True` свидетельствует о том, что метод выполнил обработку исключения и дальнейшее распространение исключения не требуется. В большинстве случаев мы будем возвращать значение `False` или некоторое выражение, которое в логическом контексте дает значение `False`, чтобы обеспечить возможность распространения исключений. В отсутствие явной инструкции `return` наш метод `__exit__()` будет возвращать значение `None`, которое в логическом контексте дает значение `False`, в результате чего любые исключения будут продолжать свое распространение.

В главе 10 мы будем использовать собственный менеджер контекста, чтобы обеспечить закрытие сетевых подключений и сжатых файлов, а в главе 9 – некоторые менеджеры контекста из модуля `threading` – для проверки отсутствия взаимоисключающих блокировок. Кроме того, при работе над упражнениями к этой главе у нас появится шанс создать более универсальный менеджер контекста для выполнения атомарных операций.

Дескрипторы

Дескрипторы – это классы, которые обеспечивают доступ к атрибутам других классов. Любой класс, реализующий один или более специальных методов дескрипторов – `__get__()`, `__set__()` и `__delete__()`, называется дескриптором (и может использоваться как дескриптор).

Реализации встроенных функций `property()` и `classmethod()` используют в своей работе дескрипторы. Чтобы разобраться в дескрипторах, важно понять, что хотя они и создаются в классах как атрибуты классов, тем не менее интерпретатор обращается к ним как к экземплярам классов.

Чтобы пояснить вышесказанное, представим, что у нас имеется класс, экземпляры которого хранят некоторые строки. Нам необходимо обеспечить доступ к строкам обычным способом, например, как к свойству, а также необходимо обеспечить возможность получения версий строк, в которых были бы экранированы служебные символы XML.

Одним из простых решений было бы сразу же создавать копии экранированных строк. Но если у нас имеются тысячи строк и нам необходимо прочитать лишь несколько экранированных версий строк, такое решение привело бы к неоправданному перерасходу памяти и вычислительных мощностей. Поэтому мы создадим дескриптор, который будет возвращать экранированные строки по требованию, не сохраняя их в памяти. Сначала рассмотрим клиентский класс (класс-владелец), то есть класс, который использует дескриптор:

```
class Product:

    __slots__ = ("__name", "__description", "__price")

    name_as_xml = XmlShadow("name")
    description_as_xml = XmlShadow("description")

    def __init__(self, name, description, price):
        self.__name = name
        self.description = description
        self.price = price
```

Единственное, что мы опустили, – это определение свойств. Свойство `name` доступно только для чтения, а свойства `description` и `price` доступны для чтения и для записи. Все эти свойства определяются обычным способом. (Полный программный код вы найдете в файле *XmlShadow.py*.) Мы использовали переменную `__slots__`, чтобы у класса не было атрибута `__dict__` и его экземпляры могли иметь только эти три атрибута. Такое решение никак связано с использованием дескриптора и не является обязательным. Атрибуты класса `name_as_xml` и `description_as_xml` определяются как экземпляры дескриптора `XmlShadow`. И хотя объекты класса `Product` не имеют атрибутов `name_as_xml` и `description_as_xml`, тем не менее благодаря дескриптору мы имеем возможность написать следующий программный код (фрагмент взят из доктестов модуля):

```
>>> product = Product("Chisel <3cm>", "Chisel & cap", 45.25)
>>> product.name, product.name_as_xml, product.description_as_xml
('Chisel <3cm>', 'Chisel &lt;3cm&gt;', 'Chisel &amp; cap')
```

Такое возможно благодаря тому, что при попытке обратиться к атрибуту, например, `name_as_xml`, интерпретатор обнаруживает, что класс `Product` имеет дескриптор с таким именем и использует этот дескриптор для получения значения атрибута. Ниже приводится полное определение класса `XmlShadow`:

```
class XmlShadow:

    def __init__(self, attribute_name):
        self.attribute_name = attribute_name

    def __get__(self, instance, owner=None):
        return xml.sax.saxutils.escape(
                            getattr(instance, self.attribute_name))
```

В момент создания объектов `name_as_xml` и `description_as_xml` им посредством вызова метода инициализации класса `XmlShadow` передаются имена соответствующих атрибутов класса `Product`, чтобы дескриптор знал, с каким атрибутом ему предстоит работать. Затем, когда интерпретатор выполняет поиск атрибута `name_as_xml` или `description_as_xml`, он вызывает метод `__get__()` дескриптора. Аргумент `self` – это экземпляр дескриптора, аргумент `instance` – это экземпляр класса `Product` (то есть значение ссылки `self` экземпляра класса `Product`), а аргумент `owner` – это класс владельца (в данном случае – класс `Product`). Для получения значения соответствующего атрибута экземпляра класса `Product` используется функция `getattr()` (в данном случае – значение соответствующего свойства), которая возвращает его экранированную версию.

В случае, когда в программе только для малой части всех строк необходимо предоставить экранированные версии строк, но эти строки очень длинные, а обращения к ним следуют достаточно часто, можно было бы предусмотреть использование кэша. Например:

```
class CachedXmlShadow:

    def __init__(self, attribute_name):
        self.attribute_name = attribute_name
        self.cache = {}

    def __get__(self, instance, owner=None):
        xml_text = self.cache.get(id(instance))
        if xml_text is not None:
            return xml_text
        return self.cache.setdefault(id(instance),
                xml.sax.saxutils.escape(
                            getattr(instance, self.attribute_name)))
```

Здесь в качестве ключа используется уникальный числовой идентификатор, а не сам экземпляр, потому что ключи словаря должны быть хешируемыми (каковыми и являются числовые идентификаторы), но нам не хотелось бы накладывать такое ограничение на классы, использующие дескриптор `CachedXmlShadow`. Ключи необходимы, потому что дескрипторы создаются для всего класса, а не для его экземпляров. (Метод `dict.setdefault()` возвращает значение для заданного ключа или, если элемента с таким ключом нет, создает новый элемент с заданным ключом и значением и возвращает значение, что весьма удобно для нас.)

Получив представление о том, как могут использоваться дескрипторы для генерирования данных без необходимости сохранять их, перейдем теперь к рассмотрению дескриптора, который может использоваться для сохранения всех атрибутов данных объекта, сняв с объекта необходимость что-либо сохранять. В следующем примере мы будем использовать словарь, но в более жизненной ситуации данные можно было бы сохранять в файле или в базе данных. Ниже приводится начало

определения класса Point, использующего дескриптор (из файла *ExternalStorage.py*).

```
class Point:

    __slots__ = ()

    x = ExternalStorage("x")
    y = ExternalStorage("y")

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

Определив пустой кортеж в качестве значения атрибута __slots__, мы тем самым гарантируем, что класс вообще не будет иметь никаких атрибутов данных. При попытке выполнить присваивание атрибуту self.x интерпретатор обнаружит наличие дескриптора с именем «x» и вызовет его метод __set__(). Остальная часть определения класса здесь не показана, но она полностью повторяет определение класса Point из главы 6. Ниже приводится полное определение класса дескриптора ExternalStorage:

```
class ExternalStorage:

    __slots__ = ("attribute_name",)
    __storage = {}

    def __init__(self, attribute_name):
        self.attribute_name = attribute_name

    def __set__(self, instance, value):
        self.__storage[id(instance), self.attribute_name] = value

    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        return self.__storage[id(instance), self.attribute_name]
```

Каждый объект класса ExternalStorage имеет единственный атрибут данных, attribute_name, который хранит имя атрибута данных класса-владельца. Всякий раз, когда выполняется присваивание значения атрибуту, оно сохраняется в частном словаре класса __storage. Точно так же, когда производится попытка прочитать значение атрибута, оно извлекается из словаря __storage.

Как и в любых других методах дескриптора, аргумент self ссылается на экземпляр дескриптора, а instance – это ссылка self для объекта, содержащего дескриптор, то есть здесь self ссылается на объект класса ExternalStorage, а instance – на объект класса Point.

Несмотря на то, что атрибут __storage является атрибутом класса, тем не менее к нему можно обращаться следующим образом: self.__storage (точно так же, как можно обращаться к некоторому методу класса self.*method()*), потому что интерпретатор, не обнаружив его среди

атрибутов экземпляра, найдет его среди атрибутов класса. Единственный недостаток такого подхода состоит в том, что если экземпляр будет иметь атрибут с именем, совпадающим с именем атрибута класса, при попытке обратиться к этому имени всегда будет использоваться атрибут экземпляра. (Если это действительно необходимо, к атрибуту класса всегда можно обратиться, квалифицировав его именем класса, то есть `ExternalStorage.__storage`. Хотя такое жесткое определение в общем случае может отрицательно сказаться при создании подклассов, к частным атрибутам это не относится, так как механизм интерпретатора приведения имен все равно включает имя класса в имена таких атрибутов.)

Здесь используется немного более сложная, чем прежде, реализация специального метода `__get__()`, потому что мы предусмотрели возможность обращения объекта `ExternalStorage` к самому себе. Например, представим, что у нас имеется экземпляр `p = Point(3, 4)`, в этом случае доступ к координате *x* можно получить, обратившись к атрибуту `p.x`, а доступ к объекту `ExternalStorage`, хранящему все координаты *x*, обратившись к `Point.x`.

В завершение обсуждения дескрипторов создадим дескриптор `Property`, имитирующий поведение встроенной функции `property()` по крайней мере в отношении реализации методов доступа. Полный программный код находится в файле *Property.py*. Ниже приводится полное определение класса `NameAndExtension`, использующего этот дескриптор:

```
class NameAndExtension:

    def __init__(self, name, extension):
        self.__name = name
        self.extension = extension

    @Property               # Задействуется нестандартный дескриптор Property
    def name(self):
        return self.__name

    @Property               # Задействуется нестандартный дескриптор Property
    def extension(self):
        return self.__extension

    @extension.setter       # Задействуется нестандартный дескриптор Property
    def extension(self, extension):
        self.__extension = extension
```

Порядок использования дескриптора точно такой же, как и в случае использования встроенных декораторов `@property` и `@`*`propertyName`*`.setter`. Ниже приводится начало определения дескриптора `Property`:

```
class Property:

    def __init__(self, getter, setter=None):
        self.__getter = getter
```

```
        self.__setter = setter
        self.__name__ = getter.__name__
```

Метод инициализации класса принимает одну или две функции в качестве аргументов. Если он используется как декоратор, он просто получит декорируемую функцию, которая станет функцией чтения, а в качестве функции записи будет установлено значение `None`. В качестве имени свойства здесь используется имя метода чтения. Для каждого свойства, для которого уже определена функция чтения, имеется возможность определить функцию записи, используя имя свойства.

```
    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        return self.__getter(instance)
```

Когда выполняется обращение к свойству, возвращается результат вызова функции чтения, которой в первом аргументе передается экземпляр класса. На первый взгляд запись `self.__getter()` напоминает вызов метода, но в действительности это не так. На самом деле `self.__getter` – это атрибут, который содержит ссылку на заданный метод. Поэтому фактически сначала происходит извлечение значения атрибута (`self.__getter`), а затем это значение вызывается как функция `()`. А так как атрибут вызывается как функция, а не как метод, мы должны явно передать ей соответствующий объект `self`. Внутри методов дескриптора ссылка на сам объект (экземпляр класса, использующего дескриптор) называется `instance` (так как `self` – это объект дескриптора). То же относится и к методу `__set__()`.

```
    def __set__(self, instance, value):
        if self.__setter is None:
            raise AttributeError("'{0}' is read-only".format(
                                 self.__name__))
        return self.__setter(instance, value)
```

В случае отсутствия функции записи возбуждается исключение `AttributeError`; в противном случае функция вызывается и ей передаются ссылка на экземпляр класса и новое значение атрибута.

```
    def setter(self, setter):
        self.__setter = setter
        return self.__setter
```

Этот метод вызывается, когда интерпретатор достигает, например, вызова `@extesion.setter`, с декорируемой функцией в качестве аргумента. Он сохраняет указанный метод записи (который теперь может вызываться методом `__set__()`) и возвращает функцию записи, потому что любой декоратор должен возвращать декорированную им версию функции или метода.

Мы рассмотрели три совершенно разные области использования дескрипторов. Дескрипторы представляют собой очень гибкое и мощное

средство, позволяющее выполнять за кулисами самые разные действия и выглядеть при этом простыми атрибутами клиентского класса (класса владельца).

Декораторы классов

Точно так же, как имеется возможность создавать декораторы для функций и методов, можно создавать декораторы для целых классов. Декораторы классов принимают класс (результат действия инструкции `class`) и должны возвращать класс – обычно модифицированную версию декорируемого класса. В этом подразделе мы познакомимся с двумя декораторами классов и рассмотрим их реализацию.

Класс SortedList, стр. 314

В главе 6 мы создали собственный тип коллекции `SortedList`, который содержит обычный список в виде частного атрибута `self.__list`. Восемь методов этого класса просто перекладывают свою работу на методы частного атрибута. В качестве примера ниже приводится реализация методов `SortedList.clear()` и `SortedList.pop()`:

```
def clear(self):
    self.__list = []

def pop(self, index=-1):
    return self.__list.pop(index)
```

В методе `clear()` не делается ничего особенного, так как тип `list` не имеет соответствующего метода, но в методе `pop()` и еще в шести других методах, которые делегируются классом `SortedList`, можно просто вызывать соответствующие методы класса `list`. Реализовать это можно с помощью декоратора классов `@delegate`, реализацию которого можно найти в модуле `Util`, поставляемом с примерами к книге. Ниже приводится начало определения новой версии класса `SortedList`:

```
@Util.delegate("__list", ("pop", "__delitem__", "__getitem__",
                          "__iter__", "__reversed__", "__len__", "__str__"))
class SortedList:
```

Первый аргумент – это имя атрибута, которому будут делегированы операции, а второй аргумент – это последовательность из одного или более методов, которые должен будет реализовать декоратор `delegate()`, чтобы избавить нас от этой рутины. Этот подход используется при определении класса `SortedList`, в файле `SortedListDelegate.py`, поэтому в нем отсутствует явная реализация перечисленных методов, но несмотря на это он полностью их поддерживает. Ниже приводится определение декоратора классов, который создает реализацию методов:

```
def delegate(attribute_name, method_names):
    def decorator(cls):
        nonlocal attribute_name
        if attribute_name.startswith("__"):
```

```
            attribute_name = "_" + cls.__name__ + attribute_name
        for name in method_names:
            setattr(cls, name, eval("lambda self, *a, **kw: "
                                    "self.{0}.{1}(*a, **kw)".format(
                                    attribute_name, name)))
        return cls
    return decorator
```

Мы не можем использовать простой декоратор, т. к. декоратору требуется передавать аргументы, поэтому мы создали функцию, которая принимает наши аргументы и возвращает декоратор класса. Сам декоратор принимает единственный аргумент – класс (так же как декоратор функций принимает функцию или метод в виде единственного аргумента).

Мы вынуждены использовать инструкцию `nonlocal`, потому что вложенная функция обращается к аргументу `attribute_name`, находящемуся в области видимости внешней функции. А нам, в случае необходимости, нужна возможность корректировать имя атрибута, чтобы учесть приведение имен частных атрибутов. Декоратор обладает весьма простым поведением: он выполняет итерации по всем именам методов, которые были переданы функции `delegate()`, и для каждого из них создает новый метод, который устанавливается в качестве атрибута класса с заданным именем метода.

Для создания каждого из делегируемых методов используется функция `eval()`, которая может использоваться для выполнения единственной инструкции `lambda`, воспроизводящей метод или функцию. Например, данный программный код воспроизводит метод `pop()`, как показано ниже:

```
lambda self, *a, **kw: self._SortedList__list.pop(*a, **kw)
```

Использование формы представления аргументов `*` и `**` позволяет любым аргументам, даже относящимся к делегируемым методам, принимать требуемую форму списка аргументов. Например, метод `list.pop()` принимает единственный аргумент – номер позиции в списке (или ни одного аргумента, в этом случае используется значение по умолчанию – номер позиции последнего элемента). Этот прием пригоден, даже когда методу передается неверное число аргументов или недопустимые значения, потому что в этом случае вызываемый метод класса `list` возбудит соответствующее исключение.

Второй декоратор классов, который мы рассмотрим, также будет применяться к классу, созданному в главе 6. Когда мы создавали класс `FuzzyBool`, мы упоминали, что при наличии реализации всего двух специальных методов, `__lt__()` и `__eq__()` (выполняющих операции `<` и `==`), остальные методы сравнения будут генерироваться автоматически. Но тогда мы не показали полное начало определения класса:

Класс `FuzzyBool`, стр. 291

```
@Util.complete_comparisons
class FuzzyBool:
```

Остальные четыре метода сравнивания определяются декоратором `complete_comparsions()` класса. Используя только реализацию поддержки оператора `<` (или `<` и `==`), декоратор воспроизводит реализацию недостающих методов, используя следующие логические соотношения:

$$x = y \Leftrightarrow \neg(x < y \vee y < x)$$
$$x \neq y \Leftrightarrow \neg(x = y)$$
$$x > y \Leftrightarrow y < x$$
$$x \leq y \Leftrightarrow \neg(y < x)$$
$$x \geq y \Leftrightarrow \neg(x < y)$$

Если декорируемый класс содержит реализацию операторов `<` и `==`, декоратор будет использовать обе реализации; при этом декоратор перейдет к воспроизводению всех методов сравнивания через оператор `<`, если в классе присутствует реализация только этого оператора. (В действительности интерпретатор автоматически воспроизводит поддержку оператора `>`, если поддерживается оператор `<`; `!=`, если поддерживается оператор `==` и `>=`, если поддерживается `<=`. Поэтому будет вполне достаточно реализовать всего три оператора: `<`, `<=` и `==`, а реализацию поддержки остальных операторов оставить за интерпретатором. Однако при использовании декоратора класса этот минимум снижается до реализации единственного оператора `<`. Это, во-первых, очень удобно, а во-вторых, гарантирует, что все операторы сравнивания будут использовать одну и ту же непротиворечивую логику.)

```
def complete_comparisons(cls):
    assert cls.__lt__ is not object.__lt__, (
            "{0} must define < and ideally ==".format(cls.__name__))
    if cls.__eq__ is object.__eq__:
        cls.__eq__ = lambda self, other: (not
                (cls.__lt__(self, other) or cls.__lt__(other, self)))
    cls.__ne__ = lambda self, other: not cls.__eq__(self, other)
    cls.__gt__ = lambda self, other: cls.__lt__(other, self)
    cls.__le__ = lambda self, other: not cls.__lt__(other, self)
    cls.__ge__ = lambda self, other: not cls.__lt__(self, other)
    return cls
```

Одна из проблем, которую необходимо решить декоратору, состоит в том, что класс `object`, от которого в конечном счете происходят все остальные классы, определяет реализацию всех шести операторов сравнивания, возбуждающих исключение `TypeError`. Поэтому необходимо узнать, были ли переопределены методы поддержки операторов `<` и `==` (и, следовательно, узнать, можно ли их использовать). Это легко можно сделать, сравнив соответствующие специальные методы декорируемого класса с методами класса `object`.

Если декорируемый класс не имеет собственной реализации поддержки оператора `<`, инструкция `assert` возбудит исключение, потому что

поддержка этого оператора является минимально необходимой. Если в классе присутствует поддержка оператора ==, мы будем использовать существующую реализацию; в противном случае создадим ее. После этого создаются остальные методы сравнивания, и возвращается класс, содержащий реализацию всех шести методов.

Использование декораторов классов является, пожалуй, самым простым и самым легким способом изменения классов. Другой способ основан на использовании метаклассов – эта тема будет рассматриваться ниже в этой главе.

Метаклассы, стр. 452

Абстрактные базовые классы

Абстрактным базовым классом (Abstract Base Class, ABC) называется класс, который не может использоваться для создания объектов. Назначение таких классов состоит в определении интерфейсов, то есть в том, чтобы перечислить методы и свойства, которые должны быть реализованы в классах, наследующих абстрактный базовый класс. Это удобно, так как можно использовать абстрактный базовый класс как своего рода договоренность – договоренность о том, что любые порожденные классы обеспечат реализацию методов и свойств, объявленных в абстрактном базовом классе.[1]

Абстрактные классы – это классы, имеющие как минимум один абстрактный метод или свойство. Объявления абстрактных методов могут не содержать их реализацию (то есть блок кода метода состоит из единственной инструкции `pass`, или, если необходимо предусмотреть обязательное переопределение метода в подклассах, из инструкции `raize NotImplementedError()`) или могут иметь действующую реализацию, которая может вызываться подклассами, например, предусматривающую обработку общих случаев. Кроме того, абстрактные классы могут содержать обычные (то есть неабстрактные) методы и свойства.

Классы, наследующие ABC, могут использоваться для создания экземпляров, только если они переопределяют все унаследованные абстрактные методы и абстрактные свойства. Дочерние классы с помощью функции `super()` могут использовать версии абстрактных методов, имеющих действующую реализацию (даже если она состоит из единственной инструкции `pass`). Все неабстрактные методы или свойства наследуются дочерними классами обычным образом. Любые абстрактные базовые классы должны использовать метакласс `abc.ABCMeta` (из модуля `abc`) или метакласс от одного из его подклассов. Метаклассы мы будем рассматривать немного ниже.

---

[1] Абстрактные базовые классы языка Python описываются в PEP 3119 (*www.python.org/dev/peps/pep-3119*), где вы также найдете очень полезные объяснения, которые стоит прочитать.

В языке Python имеется две группы абстрактных базовых классов. Одна находится в модуле `collections`, а другая – в модуле `numbers`. Они позволяют получать информацию об объекте; например, если у нас имеется переменная *x*, то мы можем определить, является она последовательностью – с помощью функции `isinstance(x, collections.MutableSequence)` или целым числом – с помощью инструкции `isinstance(x, numbers.Integral)`. Это особенно удобно, учитывая динамическую типизацию в языке Python, когда нам не требуется знать точный тип объекта, а достаточно лишь убедиться, что он поддерживает операции, которые предполагается к нему применить. Числовые абстрактные классы и абстрактные классы коллекций перечислены в табл. 8.3 и 8.4. Еще одним основным абстрактным базовым классом является класс `io.IOBase`, который наследуется всеми классами, выполняющими работу с файлами и потоками.

Таблица 8.3. Абстрактные базовые классы в модуле numbers

ABC	Наследует	API	Примеры
`Number`	`object`		`complex`, `decimal.Decimal`, `float`, `fractions.Fraction`, `int`
`Complex`	`Number`	`==`, `!=`, `+`, `-`, `*`, `/`, `abs()`, `bool()`, `complex()`, `conjugate()`; **а так же свойства** `real` **и** `imag`	`complex`, `decimal.Decimal`, `float`, `fractions.Fraction`, `int`
`Real`	`Complex`	`<`, `<=`, `==`, `!=`, `>=`, `>`, `+`, `-`, `*`, `/`, `//`, `%`, `abs()`, `bool()`, `complex()`, `conjugate()`, `divmod()`, `float()`, `math.ceil()`, `math.floor()`, `round()`, `trunc()`; **а также свойства** `real` **и** `imag`	`decimal.Decimal`, `float`, `fractions.Fraction`, `int`
`Rational`	`Real`	`<`, `<=`, `==`, `!=`, `>=`, `>`, `+`, `-`, `*`, `/`, `//`, `%`, `abs()`, `bool()`, `complex()`, `conjugate()`, `divmod()`, `float()`, `math.ceil()`, `math.floor()`, `round()`, `trunc()`; **а также свойства** `real`, `imag`, `numerator` **и** `denominator`	`fractions.Fraction`, `int`
`Integral`	`Rational`	`<`, `<=`, `==`, `!=`, `>=`, `>`, `+`, `-`, `*`, `/`, `//`, `%`, `<<`, `>>`, `~`, `&`, `^`, `|`, `abs()`, `bool()`, `complex()`, `conjugate()`, `divmod()`, `float()`, `math.ceil()`, `math.floor()`, `pow()`, `round()`, `trunc()`; **а также свойства** `real`, `imag`, `numerator` **и** `denominator`	`int`

Таблица 8.4. Основные абстрактные базовые классы в модуле collections

ABC	Наследует	API	Примеры
Callable	object	()	**Все функции, методы и лямбда-функции**
Container	object	in	bytearray, bytes, dict, frozenset, list, set, str, tuple
Hashable	object	hash()	bytes, frozenset, str, tuple
Iterable	object	iter()	bytearray, bytes, collections.deque, dict, frozenset, list, set, str, tuple
Iterator	Iterable	iter(), next()	
Sized	object	len()	bytearray, bytes, collections.deque, dict, frozenset, list, set, str, tuple
Mapping	Container, Iterable, Sized	==, !=, [], len(), iter(), in, get(), items(), keys(), values()	dict
MutableMapping	Mapping	==, !=, [], del, len(), iter(), in, clear(), get(), items(), keys(), pop(), popitem(), setdefault(), update(), values()	dict
Sequence	Container, Iterable, Sized	[], len(), iter(), reversed(), in, count(), index()	bytearray, bytes, list, str, tuple
MutableSequence	Container, Iterable, Sized	[], +=, del, len(), iter(), reversed(), in, append(), count(), extend(), index(), insert(), pop(), remove(), reverse()	bytearray, list
Set	Container, Iterable, Sized	<, <=, ==, !=, =>, >, &, |, ^, len(), iter(), in, isdisjoint()	frozenset, set
MutableSet	Set	<, <=, ==, !=, =>, >, &, |, ^, &=, |=, ^=, -=, len(), iter(), in, add(), clear(), discard(), isdisjoint(), pop(), remove()	set

Чтобы полностью интегрировать наши собственные числовые классы или классы коллекций, мы должны наследовать их от стандартных абстрактных базовых классов. Например, класс `SortedList` является последовательностью, но если ничего не предпринять, то инструкция `isinstance(L, collections.Sequence)` вернет значение `False`. Чтобы исправить этот недостаток, можно просто унаследовать соответствующий абстрактный базовый класс:

```
class SortedList(collections.Sequence):
```

Метаклассы, стр. 452

После того как `collections.Sequence` будет использован в качестве базового класса, инструкция `isinstance()` будет возвращать `True`. Кроме того, в этом случае нам придется реализовать методы `__init__()` (или `__new__()`), `__getitem__()` и `__len__()` (что мы уже сделали). Абстрактный базовый класс `collections.Sequence` также предоставляет неабстрактные реализации методов `__contains__()`, `__iter__()`, `__reversed__()`, `count()` и `index()`. Мы переопределили в классе `SortedList` все эти методы, но мы могли бы использовать версии этих методов из абстрактного базового класса, не создавая повторные реализации. Мы не можем объявить класс `SortedList` подклассом класса `collections.MutableSequence`, хотя список относится к категории изменяемых объектов, потому что класс `SortedList` не имеет всех методов, которые должны реализовать наследники класса `collections.MutableSequence`, – таких как `__setitem__()` и `append()`. (Реализация такой версии класса `SortedList` приводится в файле *SortedListAbc.py*. Альтернативный способ превращения класса `SortedList` в подкласс класса `collections.Sequence` будет описан в разделе «Метаклассы».)

Теперь, когда мы знаем, как создавать собственные классы с использованием стандартных абстрактных классов, перейдем к другому применению абстрактных базовых классов: для обеспечения соглашений об интерфейсе в наших собственных классах. Мы рассмотрим три различных примера, чтобы представить различные аспекты создания и использования абстрактных базовых классов.

Начнем с очень простого примера, демонстрирующего, как организовать работу со свойствами, доступными для чтения и для записи. Класс используется для представления бытовых приборов. Каждый объект класса, представляющий прибор, должен содержать строку с названием модели, доступную только для чтения, и цену, доступную для чтения и для записи. Нам также необходимо гарантировать переопределение метода `__init__()` базового абстрактного класса в классах-наследниках. Ниже приводится определение абстрактного базового класса (из файла *Appliance.py*); мы опустили строку с инструкцией

import abc, которая необходима, чтобы получить доступ к функциям abstractmethod() и abstractproperty(), каждая из которых может использоваться как декоратор:

```
class Appliance(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def __init__(self, model, price):
        self.__model = model
        self.price = price

    def get_price(self):
        return self.__price

    def set_price(self, price):
        self.__price = price

    price = abc.abstractproperty(get_price, set_price)

    @property
    def model(self):
        return self.__model
```

В качестве метакласса мы указали abc.ABCMeta, так как это является обязательным требованием при создании абстрактных классов. Безусловно, точно так же можно было бы использовать любой из подклассов класса abc.ABCMeta. Метод __init__() объявлен абстрактным, чтобы гарантировать его переопределение в дочерних классах, и предусмотрена его реализация, которая, как ожидается (но не обязательно), будет использоваться классами-наследниками. Мы не можем использовать декоратор для создания абстрактного свойства, доступного для чтения и для записи; кроме того, мы не использовали частные имена для методов чтения и записи, так как это привело бы к неудобствам при переопределении в подклассах. Свойство model не является абстрактным, поэтому его не обязательно переопределять в подклассах. Класс Appliance не может использоваться для создания объектов, так как он содержит абстрактные атрибуты. Ниже приводится пример его подкласса:

```
class Cooker(Appliance):

    def __init__(self, model, price, fuel):
        super().__init__(model, price)
        self.fuel = fuel

    price = property(lambda self: super().price,
                     lambda self, price: super().set_price(price))
```

Класс Cooker должен переопределить метод __init__() и свойство price. Переопределив свойство, мы просто переложили всю работу на базовый класс. Свойство model, доступное только для чтения, наследуется в обычном порядке. Мы могли бы на основе класса Appliance создать намного больше классов, таких как Fridge, Toaster и т. д.

Следующий абстрактный базовый класс, который мы рассмотрим, еще короче. Это абстрактный класс функтора (в файле *TextFilter.py*), выполняющего фильтрацию текста:

```
class TextFilter(metaclass=abc.ABCMeta):

    @abc.abstractproperty
    def is_transformer(self):
        raise NotImplementedError()

    @abc.abstractmethod
    def __call__(self):
        raise NotImplementedError()
```

Абстрактный класс `TextFilter` вообще не содержит никакой функциональности – он существует исключительно ради того, чтобы определить интерфейс, – в данном случае свойство `is_transformer` и метод `__call__()`, которые должны быть переопределены во всех его подклассах. Поскольку абстрактные свойство и метод не имеют реализации, отсутствует возможность обращаться к ним из подклассов, поэтому при попытке задействовать их (например, с помощью функции `super()`) вместо выполнения безвредной инструкции `pass` возбуждается исключение.

Ниже приводится пример простого подкласса:

```
class CharCounter(TextFilter):

    @property
    def is_transformer(self):
        return False

    def __call__(self, text, chars):
        count = 0
        for c in text:
            if c in chars:
                count += 1
        return count
```

Данный фильтр текста не является преобразователем, потому что он не изменяет заданный текст, а просто возвращает количество указанных символов в тексте. Ниже приводится пример использования этого класса:

```
vowel_counter = CharCounter()
vowel_counter("dog fish and cat fish", "aeiou") # вернет: 5
```

Два других класса текстовых фильтров, `RunLengthEncode` и `RunLengthDecode`, являются преобразователями. Ниже приводятся примеры их использования:

```
rle_encoder = RunLengthEncode()
rle_text = rle_encoder(text)
...
```

```
rle_decoder = RunLengthDecode()
original_text = rle_decoder(rle_text)
```

Класс RunLengthEncode преобразует строку байтов в кодировке UTF-8, замещая байты 0x00 последовательностью 0x00, 0x01, 0x00, и любые последовательности, содержащие от трех до 255 одинаковых байтов, – последовательностями 0x00, *количество*, *байт*. Если в строке имеется много фрагментов, состоящих из четырех идущих подряд одинаковых символов, этот класс будет способен воспроизвести более короткую строку байтов, чем простая последовательность байтов в кодировке UTF-8. Класс RunLengthDecode принимает строку байтов, созданную классом RunLengthEncode, и возвращает оригинальную строку. Ниже приводится начало определения класса RunLengthDecode:

```
class RunLengthDecode(TextFilter):

    @property
    def is_transformer(self):
        return True

    def __call__(self, rle_bytes):
        ...
```

Мы опустили тело метода __call__(), но вы можете увидеть его в файле с исходными текстами, среди примеров к этой книге.[1] Класс RunLengthEncode имеет ту же структуру.

Последний абстрактный базовый класс, который мы рассмотрим, описывает прикладной программный интерфейс (Application Programming Interface, API) и предоставляет реализацию по умолчанию механизма отмены изменений. Ниже приводится полное определение абстрактного класса (из файла *Abstract.py*):

```
class Undo(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def __init__(self):
        self.__undos = []

    @abc.abstractproperty
    def can_undo(self):
        return bool(self.__undos)

    @abc.abstractmethod
    def undo(self):
        assert self.__undos, "nothing left to undo"
        self.__undos.pop()(self)

    def add_undo(self, undo):
        self.__undos.append(undo)
```

[1] *TextFilter.py. – Прим. перев.*

Методы `__init__()` и `undo()` должны переопределяться в дочерних классах, потому что оба они объявлены абстрактными. Точно так же должно переопределяться свойство `can_undo`, доступное только для чтения. Подклассы могут не переопределять метод `add_undo()`, хотя это и не возбраняется. Метод `undo()` таит в себе одну хитрость. Список `self.__undos`, как ожидается, должен хранить ссылки на методы. Каждый метод в списке должен выполнять действия по отмене соответствующих изменений – все станет намного понятнее, когда мы рассмотрим подкласс класса `Undo`, который приводится чуть ниже. То есть, чтобы выполнить отмену, из списка `self.__undos` извлекается последний метод отмены и затем вызывается как функция, которой в виде аргумента передается ссылка `self`. (Мы вынуждены передавать ссылку `self`, потому что в данном случае метод вызывается как функция, а не как метод.)

Ниже приводится начало определения класса `Stack`. Он наследует класс `Undo`, поэтому любые действия, выполняемые над ним, можно отменить, вызвав метод `Stack.undo()` без аргументов:

```
class Stack(Undo):

    def __init__(self):
        super().__init__()
        self.__stack = []

    @property
    def can_undo(self):
        return super().can_undo

    def undo(self):
        super().undo()

    def push(self, item):
        self.__stack.append(item)
        self.add_undo(lambda self: self.__stack.pop())

    def pop(self):
        item = self.__stack.pop()
        self.add_undo(lambda self: self.__stack.append(item))
        return item
```

Мы опустили методы `Stack.top()` и `Stack.__str__()`, поскольку ни в одном из них не содержится ничего нового для нас, и ни один из них никак не взаимодействует с базовым классом `Undo`. В случае со свойством `can_undo` и методом `undo()` мы просто перекладываем работу на базовый класс. Если бы они не были объявлены как абстрактные, нам вообще не пришлось бы переопределять их, чтобы добиться того же эффекта. Но в данном случае мы специально предусмотрели обязательное их переопределение в подклассах, чтобы реализация отмены выполнялась с учетом особенностей подкласса. Методы `push()` и `pop()` выполняют основную операцию и добавляют в список методов отмены функцию,

с помощью которой можно будет выполнить отмену только что выполненной операции.

Наибольшую пользу абстрактные классы приносят в крупных программах, в библиотеках и в прикладных платформах, где они помогают обеспечить взаимодействие между классами независимо от того, кем они написаны, и от особенностей их реализации, потому что они будут обеспечивать прикладные интерфейсы, объявляемые абстрактными базовыми классами.

Множественное наследование

Множественное наследование возникает там, где один класс наследует два или более других классов. Хотя язык Python (и, например, C++) полностью поддерживает множественное наследование, некоторые языки программирования, наиболее заметным из которых является язык Java, такой возможностью не обладают. Одна из проблем состоит в том, что множественное наследование может привести к тому, что один и тот же класс будет унаследован несколько раз (например, когда два базовых класса наследуют один общий класс), а это означает, что вызываемая версия метода, не реализованного в подклассе, но реализованного в двух или более базовых классах (или в их базовых классах и т. д.), зависит от того, в каком порядке выполняется поиск в базовых классах, что может сделать классы, наследующие несколько классов, довольно неустойчивыми.

Вообще говоря, множественного наследования можно избежать, применяя простое наследование (один базовый класс) и добавляя метаклассы, когда возникает необходимость в поддержке дополнительных API, поскольку, как будет показано в следующем подразделе, метакласс может использоваться, чтобы взять обязательство о поддержке определенного API, без фактического наследования каких-либо методов или атрибутов данных. Как вариант, при множественном наследовании можно использовать один конкретный класс и один или более абстрактных классов – для обеспечения поддержки дополнительных API. Еще одно решение состоит в том, чтобы использовать простое наследование и агрегировать экземпляры других классов.

Тем не менее в некоторых случаях множественное наследование предоставляет очень удобное решение. Например, предположим, что нам необходимо создать новую версию класса `Stack`, рассматривавшегося в предыдущем подразделе, которая обеспечивала бы возможность загружать и сохранять данные с помощью модуля `pickle`. Нам необходимо добавить возможность загрузки и сохранения в нескольких классах, поэтому мы реализуем эту функциональность в виде отдельного класса:

```
class LoadSave:

    def __init__(self, filename, *attribute_names):
```

```
            self.filename = filename
            self.__attribute_names = []
            for name in attribute_names:
                if name.startswith("__"):
                    name = "_" + self.__class__.__name__ + name
                self.__attribute_names.append(name)

        def save(self):
            with open(self.filename, "wb") as fh:
                data = []
                for name in self.__attribute_names:
                    data.append(getattr(self, name))
                pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)

        def load(self):
            with open(self.filename, "rb") as fh:
                data = pickle.load(fh)
                for name, value in zip(self.__attribute_names, data):
                    setattr(self, name, value)
```

Класс имеет два атрибута: `filename` – который является общедоступным и может изменяться в любой момент, и `__attribute_names` – который доступен только для чтения и может устанавливаться только в момент создания экземпляра. Метод `save()` выполняет итерации по всем именам атрибутов и создает список с именем `data`, в котором запоминаются значения всех сохраняемых атрибутов, после чего данные записываются в файл средствами модуля `pickle`. Инструкция `with` гарантирует, что открытый файл будет закрыт и любое возникшее исключение будет передано вверх по стеку вызовов. Метод `load()` выполняет итерации по именам атрибутов и соответствующим им элементам данных, и в каждый из атрибутов записывается его значение, загруженное из файла.

Ниже приводится начало определения класса `FileStack`, наследующего класс `Undo` из предыдущего подраздела и класс `LoadSave` из этого подраздела:

```
class FileStack(Undo, LoadSave):

    def __init__(self, filename):
        Undo.__init__(self)
        LoadSave.__init__(self, filename, "__stack")
        self.__stack = []

    def load(self):
        super().load()
        self.clear()
```

Остальная часть класса совпадает с определением класса `Stack`, поэтому мы не стали воспроизводить ее здесь. Мы используем метод `__init__()`, в котором задаем инициализируемые базовые классы, вместо использования функции `super()`, которая не способна предполагать, метод какого из базовых классов следует вызывать. Методу ини-

циализации класса LoadSave передаются имя файла и имена сохраняемых атрибутов – в данном случае это единственный атрибут, частный атрибут __stack. (Мы не предполагаем (и не могли бы) сохранять значение атрибута __undos, потому что его значением является список методов, которые невозможно сохранить в файле.)

Класс FileStack содержит все необходимые методы отмены, а класс LoadSave – методы save() и load(). Мы не переопределяем метод save(), поскольку его реализация в базовом классе вполне отвечает нашим требованиям, но в методе load() сразу после загрузки нам требуется дополнительно очистить список отмен. Это необходимо, потому что после сохранения стека в файле мы могли выполнить некоторые операции над ним, а затем загрузить сохраненные ранее данные. Операция загрузки затирает данные, которые раньше находились в стеке, поэтому наличие каких-либо методов отмены в списке теряет всякий смысл. Оригинальный класс Undo не имеет метода clear(), поэтому мы добавили свой:

```
def clear(self):        # В классе Undo
    self.__undos = []
```

В методе Stack.load() мы использовали функцию super() для вызова унаследованного метода LoadSave.load(), потому что в классе Undo отсутствует метод load(), который мог бы быть причиной неоднозначности. Если бы в обоих базовых классах имелся метод load(), то выбор вызываемого метода зависел бы от того, в каком порядке интерпретатор осуществляет поиск методов в базовых классах. Предпочтительно использовать функцию super() только при отсутствии неоднозначности, а в противном случае прямо указывать имя базового класса, чтобы не зависеть от того, в каком порядке интерпретатор просматривает базовые классы при поиске методов. В случае с вызовом self.clear() тоже нет никакой неоднозначности, потому что метод clear() имеется только в классе Undo, при этом нам не требуется использовать функцию super(), потому что в классе FileStack (в отличие от метода load()) отсутствует собственный метод clear().

Что произойдет, если позднее в класс FileStack будет добавлен метод clear()? Это может нарушить работу метода load(). Одно из решений этой проблемы состоит в том, чтобы внутри метода load() вместо self.clear() производить вызов метода как super().clear(). Но это в свою очередь может привести к тому, что будет вызван первый метод clear(), найденный в базовых классах. Чтобы защитить себя от этих проблем, при использовании множественного наследования можно выбрать тактику прямого обращения к базовым классам (в данном примере мы могли бы прямо вызывать метод Undo.clear()). Или можно вообще отказаться от использования множественного наследования и применить прием агрегирования, например, наследовать класс Undo и определить класс LoadSave так, чтобы он мог использоваться для определения атрибутов.

В этом примере множественное наследование позволило нам получить смесь двух очень разных классов и избежать необходимости самим реализовывать отмену изменений или сохранение и загрузку данных, вместо этого опираясь исключительно на возможности базовых классов. Это может быть очень удобно и оправданно, особенно, если наследуемые классы не реализуют перекрывающиеся API.

Метаклассы

Метакласс – это класс, экземплярами которого являются другие классы, то есть метаклассы используются для создания классов так же, как классы используются для создания объектов. И так же, как имеется возможность определить, какому классу принадлежит объект, используя функцию `instance()`, имеется возможность определить, наследует ли объект класса (такой как `dict`, `int` или `SortedList`) другой класс, используя для этого функцию `issubclass()`.

Самый простой способ использования метаклассов заключается в том, чтобы поместить собственный класс в стандартную иерархию абстрактных базовых классов Python. Например, чтобы сделать класс `SortedList` наследником `collections.Sequence`, вместо наследования абстрактного базового класса можно просто зарегистрировать класс `SortedList`, как `collections.Sequence`:

```
class SortedList:
    ...
collections.Sequence.register(SortedList)
```

После того как класс будет определен обычным способом, его можно зарегистрировать как подкласс абстрактного базового класса `collections.Sequence`. Операция регистрации, как показано выше, превращает класс в *виртуальный подкласс*.[1] Виртуальный подкласс сообщает (например, с помощью функций `isinstance()` или `issubclass()`), что он является подклассом класса или классов и был зарегистрирован с их помощью, но не наследует никаких данных или методов любого из этих классов.

Регистрация класса – это своего рода обещание, что класс реализует API классов, с помощью которых он был зарегистрирован, но при этом нет никаких гарантий, что обещания будут выполнены. Одно из предназначений метаклассов состоит в том, чтобы обеспечить возможность дать обещания и гарантии их соблюдения относительно API класса. Другое предназначение состоит в том, чтобы обеспечить возможность модификации класса (подобно декораторам классов). И, конечно, метаклассы могут использоваться для достижения обеих указанных целей одновременно.

---

[1] В терминологии языка Python слово *виртуальный* означает не совсем то, что оно означает в терминологии языка C++.

Предположим, что нам требуется создать группу классов, реализующих методы load() и save(). Сделать это можно, создав класс, а затем используя его как метакласс для проверки наличия этих методов:

```
class LoadableSaveable(type):

    def __init__(cls, classname, bases, dictionary):
        super().__init__(classname, bases, dictionary)
        assert hasattr(cls, "load") and \
               isinstance(getattr(cls, "load"),
                          collections.Callable), ("class '" +
               classname + "' must provide a load() method")
        assert hasattr(cls, "save") and \
               isinstance(getattr(cls, "save"),
                          collections.Callable), ("class '" +
               classname + "' must provide a save() method")
```

Классы, играющие роль метаклассов, должны наследовать общий базовый класс type или один из его подклассов.

Обратите внимание, что этот класс вызывается, когда создаются определения *классов*, использующие его, что происходит достаточно редко, поэтому затраты на метаклассы во время выполнения чрезвычайно низки. Обратите также внимание на то, что проверки должны выполняться после создания класса (вызов функции super()), поскольку только после этого атрибуты класса будут доступны. (Атрибуты находятся в словаре, но при выполнении проверок мы предпочитаем работать с фактически инициализированным классом.)

Можно было бы проверить, являются ли атрибуты load и save вызываемыми, используя функцию hasattr() для проверки наличия атрибута __call__, но вместо этого мы предпочли проверить, являются ли они экземплярами collections.Callable. Абстрактный базовый класс collections.Callable обещает (но не гарантирует), что экземпляры его подклассов (или виртуальных подклассов) смогут вызываться.

Абстрактные базовые классы модуля, стр. 443

После создания класса (вызовом type.__new__() или переопределенным методом __new__()) выполняется инициализация метакласса вызовом метода __init__(). Методу __init__() передаются: в аргументе cls – только что созданный класс; в аргументе classname – имя класса (доступно также в виде атрибута cls.__name__); в аргументе bases – список базовых классов (кроме класса object, вследствие чего список может быть пустым); в аргументе dictionary – словарь с атрибутами, которые стали атрибутами класса после создания класса cls, при условии, что мы не вмешивались в переопределение метода __new__() метакласса.

Ниже приводится пара примеров, выполненных в интерактивной оболочке, которые демонстрируют, что происходит при создании новых классов, использующих метакласс LoadableSaveable:

```
>>> class Bad(metaclass=Meta.LoadableSaveable):
...     def some_method(self): pass
Traceback (most recent call last):
...
AssertionError: class 'Bad' must provide a load() method
(AssertionError: класс 'Bad' должен иметь реализацию метода load())
```

Метакласс требует, чтобы класс, использующий его, реализовал определенные методы; в противном случае, как в данном примере, возбуждается исключение `AssertionError`.

```
>>> class Good(metaclass=Meta.LoadableSaveable):
...     def load(self): pass
...     def save(self): pass
>>> g = Good()
```

Класс `Good` соблюдает требования к API, предъявляемые метаклассом, несмотря на то, что реализация не соответствует нашим представлениям о том, каким поведением она должна обладать.

Метаклассы могут также применяться для изменения классов, использующих их. Если изменяется имя, список базовых классов или словарь создаваемого класса (например, его слоты), то необходимо будет переопределить метод `__new__()` метакласса; но в случае других изменений, например, при добавлении новых методов или атрибутов данных, достаточно будет переопределить метод `__init__()`, хотя все необходимые действия можно было бы реализовать и в методе `__new__()`. Теперь перейдем к рассмотрению метакласса, который модифицирует классы, использующие его исключительно посредством метода `__new__()`.

Вместо использования декораторов `@property` и `@`*`name`*`.setter` мы могли бы создать классы, применяющие простые соглашения об именах, используемых для идентификации свойств. Например, если класс имеет методы `get_`*`name`*`()` и `set_`*`name`*`()`, в соответствии с соглашениями можно было бы ожидать, что класс имеет частное свойство `__name`, доступное как *`instance.name`*. Реализовать это можно с помощью метакласса. Ниже приводится пример класса, в котором используется данное соглашение:

```
class Product(metaclass=AutoSlotProperties):

    def __init__(self, barcode, description):
        self.__barcode = barcode
        self.description = description

    def get_barcode(self):
        return self.__barcode

    def get_description(self):
        return self.__description

    def set_description(self, description):
```

```
        if description is None or len(description) < 3:
            self.__description = "<Invalid Description>"
        else:
            self.__description = description
```

Мы вынуждены выполнить присваивание частному атрибуту `__barcode` в методе инициализации, поскольку для него отсутствует метод записи; другое следствие этого – то, что свойство `barcode` доступно только для чтения. С другой стороны, свойство `description` доступно для чтения и для записи. Ниже приводятся несколько примеров использования этого класса в интерактивной оболочке:

```
>>> product = Product("101110110", "8mm Stapler")
>>> product.barcode, product.description
('101110110', '8mm Stapler')
>>> product.description = "8mm Stapler (long)"
>>> product.barcode, product.description
('101110110', '8mm Stapler (long)')
```

Если попытаться присвоить новое значение свойству `barcode`, будет возбуждено исключение `AttributeError` с текстом сообщения «can't set attribute» (невозможно установить значение атрибута).

Если попытаться получить перечень атрибутов класса `Product` (например, с помощью функции `dir()`), будут обнаружены только общедоступные свойства `barcode` и `description`. Методы `get_`*`name`*`()` и `set_`*`name`*`()` не попадут в этот список – их заменит свойство *`name`*. А переменные, хранящие штрих-код и описание (`__barcode` и `__description`), будут добавлены как слоты, чтобы минимизировать объем памяти, используемой экземплярами класса. Все это реализуется средствами метакласса `AutoSlotProperties`, в котором имеется единственный метод:

```
class AutoSlotProperties(type):

    def __new__(mcl, classname, bases, dictionary):
        slots = list(dictionary.get("__slots__", []))
        for getter_name in [key for key in dictionary
                            if key.startswith("get_")]:
            if isinstance(dictionary[getter_name],
                          collections.Callable):
                name = getter_name[4:]
                slots.append("__" + name)
                getter = dictionary.pop(getter_name)
                setter_name = "set_" + name
                setter = dictionary.get(setter_name, None)
                if (setter is not None and
                    isinstance(setter, collections.Callable)):
                    del dictionary[setter_name]
                dictionary[name] = property(getter, setter)
        dictionary["__slots__"] = tuple(slots)
        return super().__new__(mcl, classname, bases, dictionary)
```

При вызове методу `__new__()` метакласса передаются имена метакласса и класса, список базовых классов и словарь класса, который должен быть создан. Поскольку перед созданием класса нам необходимо изменить словарь, следует переопределить не метод `__init__()`, а метод `__new__()`.

Реализация метода начинается с копирования коллекции `__slots__`, с созданием пустой коллекции, если коллекция `__slots__` отсутствовала. Попутно кортеж преобразуется в список, чтобы впоследствии имелась возможность изменять его. Из всех атрибутов, находящихся в словаре, мы выбираем те, что начинаются с префикса `"get_"` и являются вызываемыми, то есть те, которые представляют методы чтения. Для каждого метода чтения в список `slots` добавляется частное имя атрибута, который будет хранить соответствующие данные, например, при наличии метода `get_`*`name`*`()` в список `slots` добавляется имя `__`*`name`*. После этого из словаря извлекается и удаляется ссылка на метод чтения по его оригинальному имени (обе эти операции выполняются за один раз, с помощью вызова метода `dict.pop()`). То же самое выполняется для метода записи, если таковой присутствует, и затем создается новый элемент словаря с соответствующим именем свойства в качестве ключа, например, для метода чтения с именем `get_`*`name`*`()` свойство получит имя *`name`*. Значением элемента будет свойство с методами чтения и записи (который может отсутствовать), которые были найдены и удалены из словаря.

В конце оригинальный кортеж `__slots__` замещается модифицированным списком, в который были включены частные имена для каждого добавленного свойства, и вызывается метод базового класса, чтобы создать действительный класс, но уже с использованием модифицированного словаря. Обратите внимание, что в данном случае мы должны явно передать метакласс методу базового класса – это необходимо делать всегда, когда вызывается метод `__new__()`, потому что это метод класса, а не метод экземпляра.

В этом примере нам не потребовалось переопределять метод `__init__()`, потому что все необходимое было реализовано в методе `__new__()`, однако вполне возможно переопределить оба метода: `__new__()` и `__init__()` и в каждом из них выполнить свою часть работы.

Если использование механизма наследования и приема агрегирования сравнить с ручной дрелью, а использование декораторов и дескрипторов – с электрической дрелью, то использование метаклассов можно сравнить с лазерным лучом, дающим непревзойденную мощность и гибкость. Метаклассы не являются инструментом первой необходимости, исключая, разве что, разработчиков прикладных платформ, которым необходимо предоставить своим пользователям мощные средства, не заставляя их проходить многочисленные этапы, чтобы оценить предлагаемые преимущества.

Функциональное программирование

Функциональный стиль программирования – это подход к программированию, когда вычисления программируются путем комбинирования функций, которые не изменяют свои аргументы, не обращаются к переменным, определяющим состояние программы, и не изменяют их, а результаты своей работы поставляют в виде возвращаемых значений. Основное преимущество этого подхода к программированию состоит в том, что при его использовании (теоретически) намного проще разрабатывать функции по отдельности и проще отлаживать функциональные программы. Здесь также положительно сказывается тот факт, что функциональные программы не изменяют свое состояние, поэтому вполне возможно рассуждать об их функциях с математической точки зрения.

С функциональным программированием тесно связаны три понятия: *отображение*, *фильтрация* и *упрощение*. Отображение предполагает совместное использование функции и итерируемого объекта и получение нового итерируемого объекта (или списка), каждый элемент которого представляет результат вызова функции для соответствующего элемента в оригинальном итерируемом объекте. Понятие отображения поддерживается встроенной функцией `map()`, например:

```
list(map(lambda x: x ** 2, [1, 2, 3, 4]))    # вернет: [1, 4, 9, 16]
```

Функция `map()` принимает в виде аргументов функцию и итерируемый объект и для большей эффективности возвращает итератор, а не список. В данном примере мы принудительно преобразовали итерируемый объект в список, чтобы результат выглядел более понятно.

```
[x ** 2 for x in [1, 2, 3, 4]]              # вернет: [1, 4, 9, 16]
```

Часто вместо функции `map()` можно использовать выражения-генераторы. Здесь был использован генератор списков, чтобы избежать необходимости применять функцию `list()`, а чтобы получить генератор списков, оказалось достаточно заменить внешние круглые скобки квадратными.

Фильтрация предполагает совместное использование функции и итерируемого объекта и получение нового итерируемого объекта, в состав которого включаются все те элементы оригинального итерируемого объекта, для которых функция вернула значение `True`. Это понятие поддерживается встроенной функцией `filter()`:

```
list(filter(lambda x: x > 0, [1, -2, 3, -4]))     # вернет: [1, 3]
```

Функция `filter()` принимает в виде аргументов функцию и итерируемый объект и возвращает итератор.

```
[x for x in [1, -2, 3, -4] if x > 0]              # вернет: [1, 3]
```

Функцию `filter()` всегда можно заменить выражением-генератором или генератором списков.

Упрощение предполагает совместное использование функции и итерируемого объекта и получение в качестве результата отдельного значения. При этом используется следующий порядок работы: сначала функции передаются значения первого и второго элементов итерируемого значения, затем вычисленный результат и значение третьего элемента, затем вычисленный результат и значение четвертого элемента и т. д., пока не будут использованы все элементы. Это понятие поддерживается функцией `functools.reduce()` из модуля `functools`. Ниже приводятся две строки программного кода, выполняющие одни и те же вычисления:

```
functools.reduce(lambda x, y: x * y, [1, 2, 3, 4])     # вернет: 24
functools.reduce(operator.mul, [1, 2, 3, 4])           # вернет: 24
```

В модуле `operator` имеются функции, реализующие действия всех операторов языка Python, призванные упростить программирование в функциональном стиле. Здесь во второй строке была задействована функция `operator.mul()`, чтобы избежать необходимости создавать лямбда-функцию, выполняющую умножение, как это сделано в первой строке.

В языке Python имеется еще несколько встроенных функций, выполняющих упрощение: `all()`, принимающая итерируемый объект и возвращающая `True`, если для каждого элемента итерируемого объекта встроенная функция `bool()` возвращает значение `True`; `any()`, возвращающая `True`, если хотя бы для одного элемента итерируемого объекта будет получено значение `True`; `max()`, возвращающая элемент итерируемого объекта с наибольшим значением; `min()`, возвращающая элемент итерируемого объекта с наименьшим значением; `sum()`, возвращающая сумму значений элементов итерируемого объекта.

Теперь, когда мы познакомились с ключевыми понятиями, рассмотрим несколько примеров. Начнем с двух способов получить суммарный размер всех файлов в списке `files`:

```
functools.reduce(operator.add, (os.path.getsize(x) for x in files))
functools.reduce(operator.add, map(os.path.getsize, files))
```

Использование функции `map()` часто дает более компактный программный код, чем эквивалентный генератор списков или выражение-генератор, за исключением случаев, когда используется условное выражение. Здесь вместо выражения `lambda x, y: x + y` мы использовали функцию сложения `operator.add()`.

Если бы нам потребовалось определить суммарный размер только файлов с расширением *.py*, можно было бы отфильтровать все файлы, не являющиеся файлами с программным кодом на языке Python. Ниже приводятся три варианта реализации этого действия:

```
functools.reduce(operator.add, map(os.path.getsize,
                 filter(lambda x: x.endswith(".py"), files)))
functools.reduce(operator.add, map(os.path.getsize,
                 (x for x in files if x.endswith(".py"))))
functools.reduce(operator.add, (os.path.getsize(x)
                 for x in files if x.endswith(".py")))
```

Вероятно, второй и третий вариант выглядят более предпочтительными, потому что они не требуют создавать лямбда-функцию, но выбор между использованием выражения-генератора (или генератора-списков) и функциями `map()` и `filter()` – зачастую лишь вопрос личных предпочтений.

Использование функций `map()`, `filter()` и `functools.reduce()` часто позволяет устранить циклы, как было продемонстрировано в примерах выше. Эти функции особенно удобны, когда необходимо адаптировать программный код, написанный на функциональном языке программирования, при этом в языке Python обычно имеется возможность заменить функцию `map()` генератором списков, функцию `filter()` – генератором списков с условием и во многих случаях функцию `functools.reduce()` можно заменить такими встроенными функциями, как `all()`, `any()`, `max()`, `min()` и `sum()`. Например:

```
sum(os.path.getsize(x) for x in files if x.endswith(".py"))
```

При этом получается тот же результат, что и в трех предыдущих примерах, но программный код получился более компактным.

В дополнение к функциям, реализующим действия операторов языка Python, модуль `operator` также предоставляет функции `operator.attrgetter()` и `operator.itemgetter()`, первую из которых мы коротко рассматривали выше в этой главе. Обе они возвращают функции, которые затем могут вызываться для извлечения определенных атрибутов или элементов.

Функция operator.attrgetter(), стр. 428

Операция получения среза может использоваться для извлечения последовательности, составляющей часть списка, а операция получения среза с заданным шагом может использоваться для извлечения последовательности частей списка (например, каждого третьего элемента, с помощью инструкции `L[::3]`); точно так же функция `operator.itemgetter()` может использоваться для извлечения последовательности произвольных частей, например, `operator.itemgetter(4, 5, 6, 11, 18)(L)`. Функция, возвращаемая функцией `operator.itemgetter()`, не обязательно должна вызываться непосредственно, как показано в этом примере, – ее можно сохранить и передавать в виде аргумента функции `map()`, `filter()` или `functools.reduce()`, а также использовать в словарях, списках или генераторах множеств.

Когда необходимо выполнить сортировку, можно определить ключевую функцию. Эта функция может быть любой функцией, например,

лямбда-функцией, встроенной функцией или методом (таким как `str.lower()`), а также функцией, возвращаемой функцией `operator.attrgetter()`. Например, предположим, что список *L* хранит объекты с атрибутом `priority`, тогда отсортировать список в порядке приоритетов можно следующим способом: `L.sort(key=operator.attrgetter("priority"))`.

В дополнение к модулям `functools` и `operator`, упоминавшимся выше, для обеспечения поддержки функционального стиля программирования может использоваться модуль `itertools`. Например, хотя можно выполнить итерации по двум или более спискам, применив к ним операцию конкатенации, но также можно реализовать альтернативный вариант с помощью функции `itertools.chain()`, как показано ниже:

```
for value in itertools.chain(data_list1, data_list2, data_list3):
    total += value
```

Функция `itertools.chain()` возвращает итератор, который сначала дает последовательность значений из первой последовательности, затем последовательность значений из второй последовательности и т. д., пока не будут использованы все значения из всех последовательностей. В модуле `itertools` имеется множество других функций, а в описаниях к ним приводится множество маленьких, но полезных примеров, с которыми стоит ознакомиться.

Частично подготовленные функции

Частичная подготовка функций – это создание функции из существующей функции и некоторых аргументов, в результате чего получается новая функция, которая выполняет те же действия, что и оригинальная функция, но некоторые ее аргументы оказываются фиксированными и не могут передаваться вызывающим программным кодом. Ниже приводится очень простой пример:

```
enumerate1 = functools.partial(enumerate, start=1)
for lino, line in enumerate1(lines):
    process_line(i, line)
```

В первой строке создается новая функция, `enumerate1()`. Она служит оберткой вокруг существующей функции (`enumerate()`) с именованным аргументом (`start=1`), поэтому при обращении к функции `enumerate1()` будет вызвана оригинальная функция с фиксированным аргументом и со всеми остальными аргументами, заданными во время вызова, в данном случае – с аргументом `lines`. Здесь функция `enumerate1()` была использована для обеспечения нумерации строк, начиная с 1.

Использование частично подготовленных функций может упростить программный код, особенно, когда приходится вызывать одну и ту же функцию с одними и теми же аргументами снова и снова. Например, вместо того чтобы при обработке текстовых файлов в кодировке UTF-8

в каждом вызове функции open() указывать режим и кодировку, можно просто создать пару функций с фиксированными аргументами:

```
reader = functools.partial(open, mode="rt", encoding="utf8")
writer = functools.partial(open, mode="wt", encoding="utf8")
```

Теперь текстовые файлы можно открывать для чтения вызовом `reader(`*filename*`)` и для записи – вызовом `writer(`*filename*`)`.

Одной из наиболее типичных областей применения частично подготовленных функций является программирование графического интерфейса (о котором рассказывается в главе 13), где часто бывает удобно вызывать одну определенную функцию при нажатии на любую из множества кнопок. Например:

```
loadButton = tkinter.Button(frame, text="Load",
                            command=functools.partial(doAction, "load"))
saveButton = tkinter.Button(frame, text="Save",
                            command=functools.partial(doAction, "save"))
```

В данном примере используется библиотека графического интерфейса `tkinter`, которая поставляется как стандартная часть Python. Класс `tkinter.Button` используется для создания кнопок – в этом примере создаются две такие кнопки, обе они находятся в пределах одного и того же фрейма и на каждой отображается текст, указывающий их назначение. Во время создания каждой кнопки в аргументе `command` указывается функция, которая должна вызываться библиотекой `tkinter` при нажатии кнопки, в данном случае это функция `doAction()`. Здесь была использована частично подготовленная функция, чтобы гарантировать, что первым аргументом в вызове функции `soAction()` будет строка, определяющая, какая кнопка была нажата, благодаря чему `doAction()` сможет определить, какое действие следует выполнять.

Пример: valid.py

Дескрипторы, стр. 432

В этом разделе мы объединим дескрипторы с декораторами классов, чтобы реализовать мощный механизм создания атрибутов с проверкой.

Декораторы классов, стр. 438

До сих пор при необходимости обеспечить проверку корректности значения, записываемого в атрибут, мы опирались на свойства (то есть создавали методы чтения и записи). Недостаток такого подхода заключается в том, что программный код, реализующий проверку, необходимо добавлять в каждый класс для каждого атрибута, где такая проверка необходима. Было бы намного проще и удобнее, если бы имелась возможность добавлять в классы атрибуты со встроенной проверкой корректности. Ниже приводятся несколько примеров синтаксиса, который было бы желательно иметь:

```
@valid_string("name", empty_allowed=False)
@valid_string("productid", empty_allowed=False,
              regex=re.compile(r"[A-Z]{3}\d{4}"))
@valid_string("category", empty_allowed=False, acceptable=
        frozenset(["Consumables", "Hardware", "Software", "Media"]))
@valid_number("price", minimum=0, maximum=1e6)
@valid_number("quantity", minimum=1, maximum=1000)
class StockItem:

    def __init__(self, name, productid, category, price, quantity):
        self.name = name
        self.productid = productid
        self.category = category
        self.price = price
        self.quantity = quantity
```

Регулярные выражения, стр. 524

Все атрибуты класса `StockItem` требуют проверки. Например, атрибут `productid` может содержать только непустую строку, которая начинается с трех алфавитных символов верхнего регистра и заканчивается четырьмя цифрами. Атрибут `category` может содержать только непустую строку, которая должна иметь одно из указанных значений. И атрибут `quantity` может быть только числом в диапазоне от 1 до 1000 включительно. Если попытаться записать недопустимое значение, будет возбуждено исключение.

Декораторы классов, стр. 438

Проверка реализуется посредством объединения декораторов классов и дескрипторов. Как отмечалось выше, декораторы классов могут принимать только один аргумент – декорируемый класс. Поэтому здесь используется прием, продемонстрированный при первом обсуждении декораторов классов, в результате применения которого были созданы функции `valid_string()` и `valid_number()`, принимающие любые желаемые аргументы и возвращающие декоратор, который в свою очередь принимает класс и возвращает модифицированную версию класса.

Рассмотрим функцию `valid_string()`:

```
def valid_string(attr_name, empty_allowed=True, regex=None,
                 acceptable=None):
    def decorator(cls):
        name = "__" + attr_name
        def getter(self):
            return getattr(self, name)
        def setter(self, value):
            assert isinstance(value, str), (attr_name +
                                            " must be a string")
            if not empty_allowed and not value:
                raise ValueError("{0} may not be empty".format(
```

```
                            attr_name))
                if ((acceptable is not None and value not in acceptable) or
                    (regex is not None and not regex.match(value))):
                    raise ValueError("{0} cannot be set to {1}".format(
                            attr_name, value))
                setattr(self, name, value)
            setattr(cls, attr_name, GenericDescriptor(getter, setter))
            return cls
        return decorator
```

Функция начинается с того, что создает функцию-декоратор, которая принимает класс в виде единственного аргумента. Декоратор добавляет в декорируемый класс два атрибута: частный атрибут данных и дескриптор. Например, когда функция valid_string() вызывается с именем атрибута «productid», класс `StockItem` получает атрибут `__productid`, который будет хранить строку идентификатора продукта, и дескриптор `productid` атрибута, который будет использоваться для доступа к значению. Например, если создать экземпляр класса инструкцией `item = StockItem("TV", "TVA4312", "Electrical", 500, 1)`, мы сможем получать значение идентификатора продукта как `item.productid` и изменять его инструкцией, например, `item.productid = "TVB2100"`.

Функция чтения, создаваемая декоратором, просто использует глобальную функцию `getattr()`, возвращающую значение частного атрибута данных. Функция записи реализует проверку и в конце, для записи нового (и корректного) значения в атрибут данных, использует функцию `setattr()`. В действительности частный атрибут данных создается при первой попытке присвоить ему значение.

После создания функций чтения и записи снова вызывается функция `setattr()` – на этот раз, чтобы создать новый атрибут класса с заданным именем (например, `productid`) и с дескриптором типа `GenericDescriptor` в виде значения. В конце функция-декоратор возвращает модифицированный класс, а функция `valid_string()` возвращает функцию-декоратор.

Функция `valid_number()` по своей структуре идентична функции `valid_string()`, она отличается только принимаемыми аргументами и реализацией проверки в функции записи, поэтому мы не будем показывать ее здесь. (Полный программный код примера вы найдете в файле *Valid.py*.)

Последнее, что нам осталось описать, – это дескриптор `GenericDescriptor`; и это, как оказывается, самая простая часть примера:

```
class GenericDescriptor:

    def __init__(self, getter, setter):
        self.getter = getter
        self.setter = setter

    def __get__(self, instance, owner=None):
```

```
        if instance is None:
            return self
        return self.getter(instance)

    def __set__(self, instance, value):
        return self.setter(instance, value)
```

Дескриптор используется для хранения функций чтения и записи для каждого атрибута и просто передает всю работу по чтению и записи этим функциям.

В заключение

В этой главе мы получили массу дополнительных сведений о поддержке процедурного и объектно-ориентированного программирования в языке Python и приобрели некоторое представление о поддержке функционального программирования.

В первом разделе мы узнали, как создавать выражения-генераторы, и познакомились с функциями-генераторами поближе. Мы также узнали, как динамически импортировать модули и как получать доступ к функциональным возможностям таких модулей, а также динамически выполнять программный код. В этом разделе мы увидели примеры создания и использования рекурсивных функций и нелокальных переменных. Мы также узнали, как создавать собственные декораторы функций и методов и как определять и и использовать аннотации функций.

Во втором разделе главы мы исследовали различные дополнительные аспекты объектно-ориентированного программирования. Сначала мы больше узнали о доступе к атрибутам, например, с помощью специального метода `__getattr__()`. Затем мы познакомились с функторами и увидели, как они могут использоваться для получения функций с состоянием, что также может быть достигнуто посредством добавления свойств к функции или за счет использования замыканий – оба приема также рассматриваются в этой главе. Мы узнали, как использовать инструкцию `with` вместе с менеджерами контекста и как создавать собственные менеджеры контекста. Поскольку объекты файлов в языке Python, помимо всего прочего, являются еще и менеджерами контекста, мы можем выполнять операции с файлами с использованием структур `try with ... except`, чтобы гарантировать закрытие открытых файлов без необходимости реализовать блоки `finally`.

Затем во втором разделе мы перешли к описанию дополнительных особенностей объектно-ориентированного программирования, начав с описания дескрипторов. Дескрипторы могут использоваться самыми разными способами, и эта технология лежит в основе многих стандартных декораторов Python, таких как `@property` и `@classmethod`. Мы узнали, как создавать собственные дескрипторы, и увидели три различных примера их использования. Затем мы исследовали декораторы клас-

сов и увидели, что с их помощью можно модифицировать классы почти так же, как с помощью декораторов функций можно модифицировать функции.

В последних трех подразделах второго раздела мы познакомились с поддержкой в языке Python абстрактных базовых классов, множественным наследованием и метаклассами. Мы узнали, как создавать собственные классы, использующие стандартные абстрактные базовые классы, и как создавать собственные абстрактные классы. Мы также увидели, как использовать множественное наследование для объединения в одном классе возможностей нескольких классов. А из описания метаклассов мы узнали, как оказывать влияние на процесс создания и инициализации классов (в противоположность экземплярам классов).

В предпоследнем разделе были представлены некоторые функции и модули, которые в языке Python обеспечивают поддержку функционального программирования. Мы узнали, как использовать распространенные идиомы функционального программирования, такие как отображение, фильтрация и упрощение. Мы также увидели, как создавать частично подготовленные функции.

В последнем разделе было показано, как, объединив декораторы классов и дескрипторы, можно реализовать мощный и гибкий механизм создания атрибутов со встроенной проверкой значений.

Эта глава завершает описание самого языка программирования Python. Не все особенности языка были рассмотрены в этой и в предыдущих главах, но особенности, котрые не были охвачены нами, используются очень редко. Ни в одной из последующих глав не будет представлено новых особенностей языка, однако во всех этих главах будут использоваться модули из стандартной библиотеки, которые не были описаны прежде, и в некоторых из них будут использоваться приемы, продемонстрированные в этой и в предыдущих главах. Кроме того, в программах, которые будут демонстрироваться в следующих главах, отсутствуют ограничения, применявшиеся ранее (то есть ограничения на использование только тех аспектов языка, которые были представлены к текущему моменту), поэтому они являются наиболее характерными примерами в этой книге.

Упражнения

Ни одно из упражнений, которые приводятся здесь, не требует создания большого объема программного кода, но ни одно из них нельзя назвать легким!

1. Скопируйте программу magic-numbers.py и удалите ее функции `get_function()` и все функции `load_modules()`, за исключением какой-нибудь одной. Добавьте класс функтора `GetFunction` с двумя кэшами: один – для хранения найденных функций и другой – для хране-

ния функций, которые не были найдены (чтобы избежать повторного поиска функций в модулях, где эти функции отсутствуют). Единственное изменение в функции `main()` заключается в добавлении строки `get_function = GetFunction()` перед циклом и в использовании инструкции `with`, чтобы можно было отказаться от блока `finally`. Кроме того, проверьте, что функции в модуле являются вызываемыми объектами, но не с помощью функции `hasattr()`, а с помощью проверки на принадлежность абстрактному базовому классу `collections.Callable`. Определение класса можно уместить примерно в двенадцать строк программного кода. Решение приводится в файле *magic-numbers_ans.py*.

2. Создайте новый файл модуля и определите в нем три функции: `is_ascii()`, которая возвращает `True`, если все символы в заданной строке имеют числовые коды меньше 127; `is_ascii_punctuation()`, которая возвращает `True`, если все символы в заданной строке содержатся и в строке `string.punctuation`; `is_ascii_printable()`, которая возвращает `True`, если все символы в заданной строке содержатся и в строке `string.printable`. Последние две функции структурно идентичны друг другу. Каждая функция должна быть создана с использованием инструкции `lambda`, может занимать одну или две строки и должна быть написана в функциональном стиле. Обязательно добавьте для каждой функции строки документирования с доктестами и предусмотрите запуск доктестов при попытке запустить модуль. Для реализации каждой функции потребуется от трех до пяти строк программного кода, а с учетом доктестов общий размер модуля не должен превышать 25 строк. Решение приводится в файле *Ascii.py*.
3. Создайте новый файл модуля и определите в нем класс `Atomic` менеджера контекста. Этот класс должен работать подобно классу `AtomicList`, демонстрировавшемуся в этой главе, за исключением того, что он должен работать не только со списками, но и с любыми типами изменяемых коллекций. Метод `__init__()` должен проверять тип контейнера, и, вместо того чтобы хранить флаг выбора между поверхностной и глубокой копиями, он должен в зависимости от значения флага присваивать соответствующую функцию атрибуту `self.copy` и вызывать функцию копирования в методе `__enter__()`. Метод `__exit__()` имеет немного более сложную реализацию, потому что замена содержимого списка выполняется иначе, чем замена содержимого словарей и множеств, и здесь нельзя использовать инструкцию присваивания, потому что она никак не отразится на оригинальном контейнере. Определение самого класса можно уместить примерно в тридцать строк, однако вам необходимо также добавить доктесты. Решение приводится в файле *Atomic.py*, длина которого составляет около ста пятидесяти строк, включая доктесты.

- Делегирование работы процессам
- Делегирование работы потокам

9

Процессы и потоки

С тех пор, как многоядерные процессоры получили широкое распространение, еще более важной и более практически значимой стала проблема распределения вычислительной нагрузки таким образом, чтобы получить максимальную отдачу от всех имеющихся ядер. На практике используются два основных подхода к распределению нагрузки. Один из них заключается в одновременном выполнении нескольких процессов, а другой – в одновременном выполнении нескольких потоков управления. В этой главе будет продемонстрировано, как использовать оба подхода.

Преимущество выполнения нескольких процессов, то есть запуск автономных программ, заключается в том, что каждый процесс работает независимо от других. Тем самым все бремя разрешения конфликтов ложится на операционную систему. Недостаток такого подхода заключается в неудобстве организации взаимодействий и совместного использования данных между вызывающей программой и отдельными процессами. В системах UNIX запуск отдельных процессов может выполняться с использованием парадигмы ветвления процессов, но для кросс-платформенных программ должно использоваться другое решение. Простейшее решение, которое будет показано здесь, заключается в том, что вызывающая программа передает данные запускаемым ею процессам и оставляет за ними возможность самостоятельно воспроизвести результаты. Наиболее гибкое решение, существенно упрощающее двусторонний обмен данными, заключается в использовании механизмов сетевых взаимодействий. Конечно, во многих ситуациях в таком взаимодействии нет никакой необходимости, когда вполне достаточно запустить одну или более программ с помощью управляющей программы.

Альтернативой выполнению работы независимыми процессами является создание многопоточных программ, которые распределяют рабо-

ту между независимыми потоками выполнения. Преимуществом такого подхода является простота совместного использования данных (если при этом гарантируется, что в каждый конкретный момент времени доступ к данным имеет только один поток выполнения), но в этом случае все бремя разрешения конфликтов ложится на плечи программиста. В языке Python имеется отличная поддержка создания многопоточных программ, позволяющая свести к минимуму работу, которую нам необходимо выполнить самостоятельно. Тем не менее многопоточные программы намного сложнее однопоточных программ и требуют значительно большей аккуратности при их создании и сопровождении.

В первом разделе этой главы мы создадим две маленькие программы. Первая программа будет запускаться пользователем, а вторая – первой программой, причем вторая программа будет запускаться в виде отдельного процесса. Второй раздел начнется с введения в многопоточное программирование. После этого мы создадим многопоточную программу, реализующую ту же функциональность, что и две программы из первого раздела, с целью продемонстрировать различия между решениями, основанными на использовании нескольких процессов и нескольких потоков выполнения. А затем мы рассмотрим еще одну многопоточную программу, более сложную, чем первую, которая выполняет работу несколькими потоками и собирает воедино полученные результаты.

Делегирование работы процессам

В определенных ситуациях программы с необходимой функциональностью уже имеются, и требуется автоматизировать их использование. Сделать это можно с помощью модуля `subprocess`, который предоставляет средства запуска других программ, передачи им любых параметров командной строки и в случае необходимости – возможность обмена данными с ними с помощью каналов. Один очень простой пример такой программы мы уже видели в главе 5, когда использовали функцию `subprocess.call()` для очистки консоли способом, зависящим от типа платформы. Однако эти средства могут также использоваться для создания пар программ «родитель-потомок», в которых родительская программа запускается пользователем, а она в свою очередь запускает столько экземпляров дочерней программы, сколько потребуется, причем каждой выдается отдельное задание. Именно этот прием мы рассмотрим в данном разделе.

В главе 3 мы рассматривали очень простую программу *grepword.py*, которая отыскивает слово, указанное в командной строке, в файлах, имена которых перечисляются вслед за словом. В этом разделе мы разработаем более сложную версию, способную рекурсивно отыскивать файлы во вложенных подкаталогах и делегировать работу дочерним процессам, число которых зависит от наших потребностей. На экран

будет выводиться простой список имен файлов (с путями), в которых будет обнаружено искомое слово.

Родительская программа находится в файле *grepword-p.py*, а дочерняя – в файле *grepword-p-child.py*. Взаимоотношения между этими двумя программами во время работы схематически изображены на рис. 9.1.

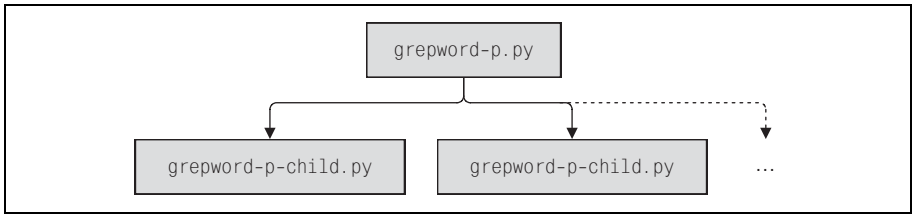

Рис. 9.1. *Родительская и дочерние программы*

Основу программы *grepword-p.py* составляет функция `main()`, которую мы рассмотрим, разделив ее на три части:

```
def main():
    child = os.path.join(os.path.dirname(__file__),
                         "grepword-p-child.py")
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    files_per_process = len(filelist)  // opts.count
    start, end = 0, files_per_process + (len(filelist) % opts.count)
    number = 1
```

Функция `get_files()`, стр. 399

Функция начинается с определения имени дочерней программы. Затем сохраняются параметры командной строки, полученные от пользователя. Функция `parse_options()` в своей работе использует модуль `optparse`. Она возвращает именованный кортеж `opts`, который определяет, должна ли программа рекурсивно выполнять поиск во вложенных подкаталогах, и количество используемых процессов (значение по умолчанию 7), максимальное количество которых было выбрано нами произвольно и равно 20. Кроме того, функция возвращает искомое слово и список имен (файлов и каталогов), полученных из командной строки. Функция `get_files()` возвращает список файлов, которые необходимо прочитать.

Получив все сведения, необходимые для решения задачи, мы определяем, сколько файлов должно быть обработано каждым из процессов. Переменные `start` и `end` будут использоваться, чтобы определить часть списка `filelist`, которая будет передаваться очередному дочернему процессу для обработки. Едва ли стоит ожидать, что число файлов

будет кратным числу процессов, поэтому для первого дочернего процесса число файлов увеличивается на величину остатка. Переменная number используется исключительно для нужд отладки – чтобы при выводе результатов можно было видеть, какая строка какому процессу принадлежит.

```
pipes = []
while start < len(filelist):
    command = [sys.executable, child]
    if opts.debug:
        command.append(str(number))
    pipe = subprocess.Popen(command, stdin=subprocess.PIPE)
    pipes.append(pipe)
    pipe.stdin.write(word.encode("utf8") + b"\n")
    for filename in filelist[start:end]:
        pipe.stdin.write(filename.encode("utf8") + b"\n")
    pipe.stdin.close()
    number += 1
    start, end = end, end + files_per_process
```

Для каждой части start:end списка filelist создается командная строка, состоящая из имени выполняемого файла интерпретатора Python (которое хранится в атрибуте sys.executable), имени файла дочерней программы, которая должна быть запущена, и параметров командной строки – в данном случае просто номер дочернего процесса, при условии, что выполнение идет в отладочном режиме. Если файл дочерней программы содержит корректную строку «shebang» или в операционной системе настроена взаимосвязь между расширением имени файла и открывающей его программой, можно было бы сразу учесть эту информацию и не беспокоиться о включении имени выполняемого файла интерпретатора. Но мы предпочитаем приведенный подход, потому что он гарантирует, что дочерняя программа будет выполняться той же версией интерпретатора Python, что и родительская.

Получив строку с командой, мы создаем объект subprocess.Popen, передаем ему эту команду для выполнения (в виде списка строк) и в данном случае задаем возможность записывать данные в поток стандартного ввода процесса. (Точно так же возможно читать данные из потока стандартного вывода, определив похожий именованный аргумент.) После этого мы записываем в поток стандартного ввода процесса искомое слово, сопровождая его символом перевода строки, а затем все имена файлов из соответствующей части списка filelist. Модуль subprocess читает и записывает байты, а не строки, поэтому мы должны кодировать строки при записи (и декодировать байты при чтении), используя соответствующую кодировку, и в данном случае мы выбрали UTF-8. Закончив передачу списка файлов дочернему процессу, мы закрываем поток стандартного ввода и идем дальше.

Нет никакой необходимости сохранять ссылку на каждый дочерний процесс (на каждой новой итерации цикла в переменную pipe будет за-

писываться новая ссылка на объект `subprocess.Popen`), потому что каждый процесс выполняется независимо, но мы сохраняем их в списке, чтобы иметь возможность прерывать их работу. Кроме того, мы не выполняем сбор информации, которая выводится дочерними процессами; вместо этого мы позволяем им выполнять вывод результатов на консоль. Это означает, что результаты, выводимые несколькими процессами, могут выводиться вперемешку. (В упражнениях вам будет предоставлена возможность предотвратить такое перемешивание.)

```
while pipes:
    pipe = pipes.pop()
    pipe.wait()
```

После запуска всех дочерних процессов мы ожидаем, пока каждый из них завершит свою работу. В UNIX-системах это гарантирует такую, может быть, не очень существенную особенность: после того как все процессы завершатся, управление передается командной строке (в противном случае нам было бы необходимо нажать клавишу Enter после завершения всех процессов). Другое преимущество такого подхода состоит в следующем: если работа программы прерывается (например, нажатием комбинации клавиш Ctrl+C), то все дочерние процессы, которые к этому моменту продолжают работу, будут прерваны и завершены с исключением `KeyboardInterrupt`, которое нельзя перехватить. В противном случае главная программа завершится (потеряв возможность прерывать работу дочерних процессов) и дочерние процессы продолжат работу (пока их не остановит команда `kill` или менеджер задач).

Ниже приводится полный программный код (за исключением комментариев и инструкций `import`) дочерней программы *grepword-p-child.py*, разбитый на две части:

```
BLOCK_SIZE = 8000

number = "{0}: ".format(sys.argv[1]) if len(sys.argv) == 2 else ""
word = sys.stdin.readline().rstrip()
```

Программа начинается с того, что запоминает свой номер или пустую строку, если она выполняется не в отладочном режиме. После этого она читает первую строку, содержащую искомое слово. Эта и вся последующая информация читается как строки.

```
for filename in sys.stdin:
    filename = filename.rstrip()
    previous = ""
    try:
        with open(filename, "rb") as fh:
            while True:
                current = fh.read(BLOCK_SIZE)
                if not current:
                    break
```

```
                current = current.decode("utf8", "ignore")
                if (word in current or
                    word in previous[-len(word):] +
                        current[:len(word)]):
                    print("{0}{1}".format(number, filename))
                    break
                if len(current) != BLOCK_SIZE:
                    break
                previous = current
        except EnvironmentError as err:
            print("{0}{1}".format(number, err))
```

Все строки, кроме первой, являются именами файлов (с путями). Для каждого имени программа открывает соответствующий файл, читает его содержимое и выводит имя файла, если в нем обнаруживается искомое слово. Вполне возможно, что некоторые файлы могут иметь очень большой размер, что может привести к проблеме нехватки памяти, особенно, когда параллельно выполняются 20 дочерних процессов и все они читают большие файлы. Мы ликвидируем эту проблему, читая каждый файл блоками, всякий раз сохраняя предыдущий прочитанный блок, чтобы гарантировать, что учтен случай попадания единственного вхождения искомого слова на границу двух блоков. Еще одно преимущество чтения файлов блоками состоит в том, что если искомое слово находится в начале файла, мы можем завершить его чтение, не читая весь файл целиком, поскольку нам достаточно знать, что слово присутствует в файле, и не важно, в каком месте внутри файла оно встречается.

Кодировки символов, стр. 112

Чтение файлов выполняется в двоичном режиме, поэтому мы должны преобразовывать каждый блок в строку, прежде чем можно будет выполнять поиск, так как искомое слово является строкой. Мы исходим из предположения, что во всех файлах содержится текст в кодировке UTF-8, но в некоторых случаях это предположение может оказаться неверным. Более сложная версия программы могла бы сначала пытаться определить фактическую кодировку, затем закрывать файл и повторно открывать его уже с корректной кодировкой. Как уже отмечалось в главе 2, существует по меньшей мере два пакета автоматического определения кодировки файлов, которые доступны в каталоге пакетов Python Package Index, *pypi.python.org/pypi*. (Может показаться заманчивым декодировать искомое слово в объект `bytes` и сравнивать объект `bytes` с объектом `bytes`, но такой прием не дает полной надежности, так как некоторые символы могут иметь более одного допустимого представления в кодировке UTF-8.)

Модуль `subprocess` предлагает гораздо более широкие возможности, чем было использовано здесь, включая эквиваленты обратным апострофам и конвейерам командной оболочки, а также функциям `os.system()` и функциям порождения дочерних процессов.

В следующем разделе мы рассмотрим многопоточную версию программы *grepword-p.py*, благодаря чему мы сможем сравнить ее с версией, которая запускает дочерние процессы. Мы также увидим более сложную многопоточную программу, которая распределяет работу между потоками выполнения и собирает результаты воедино, что дает больший контроль над тем, как они будут выводиться.

Делегирование работы потокам выполнения

Создание двух или более потоков выполнения в языке Python производится достаточно просто. Сложности появляются, когда возникает необходимость использовать одни и те же данные в разных потоках. Представьте, что два потока совместно пользуются одним и тем же списком. Один поток может приступить к выполнению итераций по списку, используя инструкцию `for x in L`, а второй поток в это же время удаляет несколько элементов где-нибудь в середине списка. В лучшем случае это будет приводить к неожиданному аварийному завершению программы, а в худшем – к получению неверных результатов.

Одно из типичных решений этой проблемы заключается в использовании механизма блокировки. Например, один поток может сначала получить блокировку и только *потом* начинать итерации по списку – в это время любой другой поток будет заблокирован, ожидая снятия блокировки. В действительности все не так просто. Связь между блокировкой и данными, которые она запирает, существует только в нашем сознании. Если один поток уже получил блокировку и ее попытается запросить второй поток, он окажется заблокированным, пока первый поток не освободит блокировку. Реализуя доступ к данным через получение блокировки, мы можем гарантировать, что в каждый конкретный момент доступ к данным будет иметь только один поток, хотя сама защита носит косвенный характер.

Одна из проблем, связанных с блокировками, заключается в том, что существует риск появления ситуации взаимоблокировки. Предположим, что *поток №1* получает блокировку A, получая доступ к элементу совместно используемых данных a, а затем пытается получить блокировку B, чтобы получить возможность доступа к элементу совместно используемых данных b, но не может этого сделать, потому что в это же время *поток №2* уже имеет блокировку B для доступа к элементу совместно используемых данных b и в свою очередь пытается получить блокировку A, чтобы получить доступ к данным a. То есть *поток №1*, имея блокировку A, пытается получить блокировку B, в то время как *поток №2*, имея блокировку B, пытается получить блокировку A.

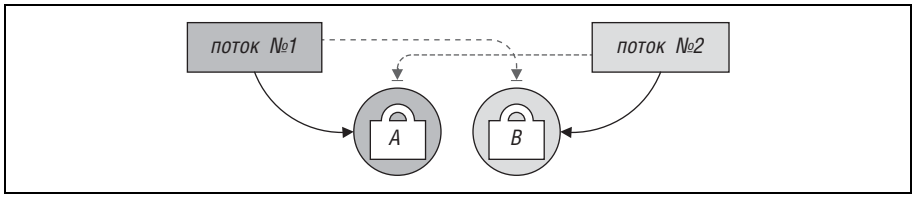

Рис. 9.2. *Взаимоблокировка: два или более потоков пытаются перекрестно получить блокировки*

В результате оба потока оказываются заблокированными, и программа оказывается в состоянии клинча, как показано на рис. 9.2.

Изобразить данный конкретный случай взаимоблокировки оказалось достаточно просто, но выявить ситуации взаимоблокировки на практике бывает намного сложнее из-за того что они не всегда очевидны. Некоторые библиотеки организации реализации механизмов многопоточной обработки данных способны распознавать потенциальные ситуации взаимоблокировки и предупреждать о них, но, чтобы избежать их, все равно требуется участие человека.

Существует один простой, но действенный прием, позволяющий избежать взаимоблокировок, который заключается в строгом соблюдении порядка получения блокировок. Например, если правило гласит, что перед получением блокировки *B* всегда должна запрашиваться блокировка *A* и в потоке выполнения необходимо получить блокировку *B*, то, согласно правилу, поток сначала должен получить блокировку *A*. Тем самым гарантируется, что состояние взаимоблокировки, описанное выше, никогда не возникнет, так как оба потока будут сначала пытаться получить блокировку *A*, и первый, кому это удастся, сможет затем запросить блокировку *B* – при условии, что все потоки выполнения следуют установленному правилу.

Еще одна проблема, связанная с блокировкой, состоит в том, что, когда несколько потоков одновременно ожидают освобождения блокировки, они остаются заблокированными и не выполняют никакой полезной работы. Эту ситуацию до некоторой степени можно смягчить, изменив стиль программирования так, чтобы минимизировать объемы работ, выполняемых в контексте блокировки.

В любой программе на языке Python имеется по крайней мере один поток выполнения – главный поток. Чтобы создать дополнительные потоки, необходимо импортировать модуль `threading` и с его помощью создать необходимое число потоков. Создать потоки можно двумя способами: вызвать функцию `threading.Thread()` и передать ей вызываемый объект или создать свой подкласс класса `threading.Thread`. В этой главе будут продемонстрированы оба способа. Прием, основанный на создании подкласса, обладает большей гибкостью и реализуется доста-

точно просто. Подклассы могут переопределить метод `__init__()` (при этом они *обязаны* вызывать реализацию базового класса) и *обязаны* переопределить метод `run()` – в этом методе выполняется вся работа потока. Метод `run()` *никогда* не должен вызываться нашим программным кодом – поток запускается вызовом метода `start()`, а этот метод сам вызовет метод `run()`, когда он будет готов к этому. Ни один из других методов класса `threading.Thread` не должен переопределяться, хотя добавление новых методов не возбраняется.

Пример: многопоточная программа поиска слова

В этом подразделе мы рассмотрим программный код программы *grepword-t.py*. Эта программа выполняет ту же самую работу, что и программа *grepword-p.py*, но в отличие от нее распределяет работу не между несколькими процессами, а между несколькими потоками выполнения. Схематическое изображение программы приводится на рис. 9.3.

Одна из интересных особенностей программы состоит в том, что она вообще не имеет никаких блокировок. Такое стало возможным благодаря тому, что единственным совместно используемым элементом данных является список имен файлов, который реализован в виде объекта класса `queue.Queue`. Особенность класса `queue.Queue` заключается в том, что он сам выполняет все необходимые блокировки, поэтому всякий раз, когда мы выполняем добавление или удаление элементов, мы можем опираться на внутреннюю реализацию очереди, которая сама *упорядочивает* доступ. В контексте потоков такое упорядочение означает, что в каждый конкретный момент времени доступом к данным обладает только один поток выполнения. Еще одно преимущество использования класса `queue.Queue` состоит в том, что нам не требуется распределять работу между потоками – мы просто добавляем в очередь элементы, которые требуется обработать, а рабочие потоки сами принимаются за работу, когда они будут готовы к этому.

Класс `queue.Queue` работает по принципу «первым пришел, первым ушел». Кроме того, в модуле `queue` имеется реализация списков `queue.LifoQueue`, которая работает по принципу «последним пришел, первым ушел», а также реализация очереди `queue.PriorityQueue`, которая состоит из двухэлементных кортежей (приоритет, элемент данных), где эле-

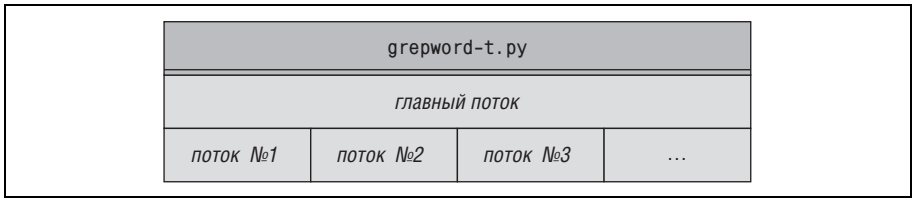

Рис. 9.3. *Многопоточная программа*

менты с наименьшим значением приоритета обрабатываются первыми. Для всех очередей можно определить максимальный размер – если будет достигнут максимальный размер, очередь блокирует все последующие попытки добавить элементы, пока из нее не будут удалены существующие элементы.

Мы рассмотрим программу *grepword-t.py*, разделив ее на три части, и начнем с полного определения функции `main()`:

```
def main():
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    work_queue = queue.Queue()
    for i in range(opts.count):
        number = "{0}: ".format(i + 1) if opts.debug else ""
        worker = Worker(work_queue, word, number)
        worker.daemon = True
        worker.start()
    for filename in filelist:
        work_queue.put(filename)
    work_queue.join()
```

Получение параметров командной строки и списка имен файлов выполняется точно так же, как и раньше. После сбора всей необходимой информации создается очередь типа `queue.Queue` и затем выполняются циклы по числу создаваемых потоков (по умолчанию 7). Для каждого потока подготавливается строка, содержащая порядковый номер потока при работе в отладочном режиме (пустая строка – в противном случае), и затем создается экземпляр класса `Worker` (подкласс класса `threading.Thread`) – описание его свойства `daemon` приводится чуть ниже. После этого поток запускается на выполнение. В этот момент времени для него еще нет работы, поскольку очередь заданий еще пуста, поэтому поток тут же окажется заблокированным на попытке получить задание из очереди.

После того как будут созданы и запущены все потоки, функция выполняет итерации по всем именам файлов, добавляя каждый из них в очередь заданий. Как только в очередь будет добавлен первый файл, один из потоков сможет получить его и начать поиск слова, и это будет происходить, пока все потоки не получат по файлу для выполнения своей работы. Как только поток завершит обработку файла, он тут же получит из очереди следующий файл, и так до тех пор, пока не будут обработаны все файлы.

Обратите внимание, что такая организация распределения работы отличается от принятой в программе *grepword-p.py*, где каждому дочернему процессу выделялся фрагмент списка, после чего дочерние процессы последовательно обрабатывают файлы в своих списках. В случаях, подобных этому, применение потоков выполнения дает более высокую производительность. Например, если первые пять файлов в списке имеют очень большой размер, а остальные файлы очень маленькие, то

каждый большой файл будет обрабатываться отдельным потоком, так как каждый поток извлекает из очереди только одно задание за раз, что приведет к более равномерному распределению нагрузки. В программе *gerpword-p.py*, напротив, все большие файлы достанутся первому дочернему процессу, а остальным процессам – маленькие файлы, поэтому на долю первого процесса может выпасть значительный объем работы, в то время как остальные процессы могут быстро завершиться, сделав, по сути, не так много.

Программа будет продолжать работу, пока имеется хотя бы один запущенный поток выполнения. Это порождает определенную проблему, так как с технической точки зрения даже после выполнения всех заданий потоки все равно считаются выполняющимися. Решение этой проблемы состоит в том, чтобы превратить потоки в демонов. Это позволит программе завершиться, как только в ней не останется ни одного работающего потока, не являющегося демоном. Главный поток не является демоном, поэтому, как только главный поток завершится, программа завершит работу каждого потока-демона и завершится сама. Конечно, теперь может возникнуть противоположная проблема – после создания и запуска всех потоков нам необходимо гарантировать, что главный поток программы не завершится, пока не будет выполнена вся работа. Добиться этого можно вызовом метода `queue.Queue. join()` – этот метод блокирует вызывающий его поток, пока очередь не опустеет.

Ниже приводится начало определения класса `Worker`:

```
class Worker(threading.Thread):

    def __init__(self, work_queue, word, number):
        super().__init__()
        self.work_queue = work_queue
        self.word = word
        self.number = number

    def run(self):
        while True:
            try:
                filename = self.work_queue.get()
                self.process(filename)
            finally:
                self.work_queue.task_done()
```

В методе `__init__()` в обязательном порядке должен вызываться метод `__init__()` базового класса. Аргумент `work_queue` – это та самая очередь заданий `queue.Queue`, которая совместно используется всеми потоками.

Метод `run()` выполняет бесконечный цикл. Это типичная организация работы потока-демона, и в этом определенно есть смысл, так как заранее неизвестно, сколько файлов предстоит обработать потоку. В каждой итерации вызывается метод `queue.Queue.get()`, чтобы получить имя следующего файла для обработки. Этот вызов заблокирует выполнение потока, если очередь окажется пуста, и нам не требуется преду-

сматривать свои блокировки, так как все необходимые действия выполняются классом `queue.Queue` автоматически. Получив файл, поток обрабатывает его, после чего необходимо сообщить очереди, что данное задание было выполнено, вызовом метода `queue.Queue.task_done()`, что совершенно необходимо для обеспечения корректной работы метода `queue.Queue.join()`.

Мы не показали реализацию функции `process()`, потому что, кроме строки с инструкцией `def`, ее программный код, начиная со строки `previous = ""` и до конца, остался тем же, что и в программе *grepword-pchild.py* (на стр. 471).

Последнее, что хотелось бы отметить: в состав примеров к книге включена программа *grepword-m.py*, практически идентичная программе *grepword-t.py*, рассматривавшейся здесь, но в ней вместо модуля `threading` используется модуль `multiprocessing`. Ее программный код имеет ровно три отличия: первое – она импортирует модуль `multiprocessing` вместо модулей `queue` и `threading`; второе – класс `Worker` наследует класс `multiprocessing.Process`, а не `threading.Thread` и третье – очередь заданий в ней имеет тип `multiprocessing.JoinableQueue`, а не `queue.Queue`.

Модуль `multiprocessing` реализует функциональность, напоминающую потоки, используя механизм ветвления в системах, поддерживающих его (UNIX), и порождает дочерние процессы в системах, которые не поддерживают ветвление (Windows), благодаря чему механизм блокировок оказывается необходимым не всегда, а сами процессы будут выполняться на всех ядрах процессора, доступных операционной системе. Пакет предоставляет несколько способов передачи данных между процессами, включая применение очереди, которая может использоваться для распределения работы между процессами точно так же, как очередь `queue.Queue` использовалась для распределения заданий между потоками.

Главное преимущество версии программы, в которой для обработки запускаются отдельные процессы, состоит в том, что на многоядерных процессорах она потенциально способна обеспечить более высокую производительность, поскольку процессы могут выполняться на всех доступных ядрах. Сравните это со стандартным интерпретатором Python (написанным на языке C и иногда называемым CPython), в котором имеется глобальная блокировка интерпретатора (Global Interpreter Lock, GIL), что означает, что в каждый конкретный момент времени может выполняться только один поток программного кода Python. Это ограничение является особенностью конкретной реализации и необязательно применяется в других реализациях интерпретатора, таких как Jython.[1]

[1] Краткое объяснение, для чего применяется глобальная блокировка интерпретатора, приводится по адресу: *www.python.org/doc/faq/library/#can-t-weget-rid-of-the-global-interpreter-lock* и *docs.python.org/api/threads.html*.

Пример: многопоточная программа поиска дубликатов файлов

Пример второй многопоточной программы по своей структуре напоминает первую, но она более сложная. В ней используются две очереди – одна для заданий и вторая для результатов, а кроме того, имеется отдельный поток обработки результатов, который выводит их по мере поступления. Кроме того, демонстрируется прием создания подкласса класса `threading.Thread` и создание потока вызовом функции `threading.Thread()`, а также используется блокировка для упорядочения доступа к совместно используемым данным (тип `dict`).

Программа *findduplicates-t.py* является расширенной версией программы *finddup.py*, которая приводилась в главе 5. Она выполняет итерации по всем файлам в текущем каталоге (или в каталоге по указанному пути) и рекурсивно выполняет обход подкаталогов. Она сравнивает размеры всех файлов с одинаковыми именами (точно так же, как программа *finddup.py*); для файлов с одинаковыми именами и одинаковыми размерами она вычисляет контрольную сумму по алгоритму MD5 (Message Digest), чтобы убедиться, что файлы действительно являются идентичными, и сообщает о таких файлах.

Сначала мы рассмотрим функцию `main()`, разделив ее на четыре части:

```
def main():
    opts, path = parse_options()
    data = collections.defaultdict(list)
    for root, dirs, files in os.walk(path):
        for filename in files:
            fullname = os.path.join(root, filename)
            try:
                key = (os.path.getsize(fullname), filename)
            except EnvironmentError:
                continue
            if key[0] == 0:
                continue
            data[key].append(fullname)
```

Каждый ключ словаря со значениями по умолчанию `data` представляет собой кортеж из двух элементов (размер и имя файла), где имя файла не включает путь к нему, а значение представляет собой список имен файлов (с путями к ним). Любой элемент, у которого в списке имеется более одного имени файла, потенциально может содержать дубликаты. Словарь заполняется в процессе итераций по всем файлам в заданном пути, при этом пропускаются все файлы, для которых невозможно получить размер (возможно, из-за отсутствия необходимых прав доступа или потому что они не являются обычными файлами), а также файлы, имеющие нулевой размер (поскольку все файлы нулевого размера можно считать идентичными).

```
work_queue = queue.PriorityQueue()
results_queue = queue.Queue()
md5_from_filename = {}
for i in range(opts.count):
    number = "{0}: ".format(i + 1) if opts.debug else ""
    worker = Worker(work_queue, md5_from_filename, results_queue,
                    number)
    worker.daemon = True
    worker.start()
```

Собрав исходные данные, можно приступать к созданию рабочих потоков. Но сначала создается очередь заданий и очередь результатов. Очередь заданий представляет собой очередь с приоритетами, поэтому она всегда возвращает элемент с наименьшим значением приоритета (или, в нашем случае, файлы с наименьшими размерами). Затем создается словарь, в котором каждый ключ – имя файла (включая путь к нему), а значение – контрольная сумма MD5. Назначение словаря состоит в том, чтобы избежать повторного вычисления контрольной суммы для одного и того же файла (поскольку эта процедура является достаточно дорогостоящей).

Создав коллекцию для хранения совместно используемых данных, функция выполняет количество циклов, соответствующее количеству создаваемых потоков (по умолчанию семь циклов). Класс `Worker` напоминает одноименный класс, созданный в предыдущем примере, только на этот раз его экземплярам передаются две очереди и словарь контрольных сумм MD5. Как и прежде, каждый рабочий поток немедленно запускается на выполнение и каждый из них тут же блокируется, пока в очереди не появятся свободные задания.

```
results_thread = threading.Thread(
                    target=lambda: print_results(results_queue))
results_thread.daemon = True
results_thread.start()
```

Вместо создания отдельного подкласса `threading.Thread` для обработки результатов мы создали функцию, которая передается функции `threading.Thread()`. Эта функция возвращает отдельный поток, который вызовет указанную функцию сразу же после запуска. Мы передаем функции очередь с результатами (которая, конечно же, пока пустая), поэтому поток тут же оказывается заблокированным.

К этому моменту мы создали все рабочие потоки и поток обработки результатов, причем все они оказались заблокированы в ожидании появления заданий.

```
for size, filename in sorted(data):
    names = data[size, filename]
    if len(names) > 1:
        work_queue.put((size, names))
```

```
    work_queue.join()
    results_queue.join()
```

Теперь выполняются итерации по элементам словаря `data`, и для всех кортежей, состоящих из двух элементов (размер, имя), для которых имеется список из двух или более потенциальных дубликатов файлов, размер и имена файлов с путями добавляются в виде элементов в очередь заданий. Поскольку очередь является экземпляром класса из модуля `queue`, нам не нужно беспокоиться о блокировках.

В заключение выполняется присоединение к очереди заданий и к очереди с результатами, чтобы заблокировать главный поток программы до того момента, пока обе очереди не опустеют. Тем самым гарантируется, что программа будет продолжать работать, пока не будут выполнены все задания и не будут выведены все результаты, после чего программа завершает свою работу.

```
def print_results(results_queue):
    while True:
        try:
            results = results_queue.get()
            if results:
                print(results)
        finally:
            results_queue.task_done()
```

Эта функция передается в виде аргумента функции `threading.Thread()` и вызывается, когда запускается данный поток. Функция выполняет бесконечный цикл, потому что она используется в потоке, который работает в режиме демона. Эта функция просто извлекает результаты (многострочный текст) и выводит непустые строки, пока очередь с результатами не опустеет.

Начало определения класса `Worker` похоже на определение одноименного класса, который приводился выше:

```
class Worker(threading.Thread):

    Md5_lock = threading.Lock()

    def __init__(self, work_queue, md5_from_filename, results_queue,
                 number):
        super().__init__()
        self.work_queue = work_queue
        self.md5_from_filename = md5_from_filename
        self.results_queue = results_queue
        self.number = number

    def run(self):
        while True:
            try:
                size, names = self.work_queue.get()
                self.process(size, names)
```

```
            finally:
                self.work_queue.task_done()
```

Различия заключаются в том, что в этой версии у нас имеется больше совместно используемых данных и наша функция `process()` вызывается с другими аргументами. Нам не требуется беспокоиться об организации доступа к очередям, так как они гарантируют упорядочение доступа; при обращении к другим данным, в данном случае – к словарю `md5_from_filename`, мы должны управлять доступом, используя блокировку. Мы сделали блокировку атрибутом класса, потому что нам требуется, чтобы все экземпляры класса `Worker` использовали одну и ту же блокировку, то есть когда один экземпляр получает блокировку, все остальные экземпляры при попытке получить ее блокируются.

Теперь рассмотрим функцию `process()`, разделив ее на две части:

```
def process(self, size, filenames):
    md5s = collections.defaultdict(set)
    for filename in filenames:
        with self.Md5_lock:
            md5 = self.md5_from_filename.get(filename, None)
        if md5 is not None:
            md5s[md5].add(filename)
        else:
            try:
                md5 = hashlib.md5()
                with open(filename, "rb") as fh:
                    md5.update(fh.read())
                md5 = md5.digest()
                md5s[md5].add(filename)
                with self.Md5_lock:
                    self.md5_from_filename[filename] = md5
            except EnvironmentError:
                continue
```

Сначала создается пустой словарь со значениями по умолчанию, в котором каждый ключ является значением контрольной суммы MD5, а значение – множеством имен файлов, которые имеют соответствующее значение контрольной суммы. Затем выполняются итерации по всем файлам и для каждого файла извлекается его контрольная сумма MD5, если она уже была вычислена ранее; в противном случае она вычисляется.

Менеджеры контекста, стр. 428

Независимо от того, получаем ли мы доступ к словарю `md5_from_filename` для чтения или для записи, мы делаем это в контексте блокировки. Экземпляры класса `threading.Lock()` являются менеджерами контекста, которые приобретают блокировку на входе и освобождают ее на выходе. Инструкции `with` блокируются, пока блокировка `Md5_Lock` не будет освобождена, если какой-либо другой поток уже приобрел ее. В первой инструкции `with`,

после приобретения блокировки, из словаря извлекается значение контрольной суммы MD5 (или None, если она еще не вычислялась для текущего файла). Если в качестве контрольной суммы получено значение None, ее необходимо вычислить, и в этом случае она записывается в словарь md5_from_filename, чтобы предотвратить повторное ее вычисление.

Обратите внимание, что каждый раз мы стараемся минимизировать объем работы, выполняемой в контексте блокировки, чтобы свести продолжительность блокировки остальных потоков к минимуму – в данном случае в контексте блокировки выполняется единственное обращение к словарю.

Строго говоря, мы вообще можем не использовать блокировки при использовании интерпретатора CPython, так как глобальная блокировка интерпретатора (GIL) эффективно синхронизирует все попытки обращения к словарю. Однако при разработке программы мы решили не полагаться на реализацию интерпретатора с глобальной блокировкой и поэтому явно используем блокировку.

Глобальная блокировка GIL, стр. 478

```
        for filenames in md5s.values():
            if len(filenames) == 1:
                continue
            self.results_queue.put("{0}Duplicate files ({1:n} bytes):"
                                   "\n\t{2}".format(self.number, size,
                                      "\n\t".join(sorted(filenames))))
```

В конце выполняется цикл по элементам локального словаря md5s и для каждого множества, содержащего более одного имени файла, в очередь с результатами добавляется многострочный текст. Добавляемый текст содержит номер рабочего потока (по умолчанию пустая строка), размер файла в байтах и все имена файлов дубликатов. Нам не требуется использовать блокировку при обращении к очереди с результатами, так как она является экземпляром класса queue.Queue, который автоматически выполняет блокировку.

Классы из модуля queue существенно упрощают создание многопоточных приложений, а на случай, когда нам требуется использовать блокировки явно, модуль threading может предложить множество возможных вариантов. В данном примере мы использовали простейшую блокировку типа threading.Lock, но имеются и другие разновидности блокировок, включая threading.RLock (блокировка, которая может получаться повторно потоком, который уже владеет ею), threading.Semaphore (блокировка, которая может использоваться для защиты заданного числа ресурсов) и threading.Condition, которая обеспечивает условие ожидания.

Глобальная блокировка GIL, стр. 478

При использовании нескольких потоков выполнения часто можно получать более простые решения, чем при использовании модуля `subprocess`, но, к сожалению, многопоточные программы на языке Python не всегда позволяют добиться производительности, какую можно получить при реализации обработки данных несколькими процессами. Как уже отмечалось ранее, камнем преткновения здесь становится стандартная реализация интерпретатора Python, поскольку интерпретатор CPython в каждый конкретный момент времени может выполнять программный код на языке Python только на одном процессоре, даже когда в программе выполняется сразу несколько потоков.

Попытка решить эту проблему была предпринята в модуле `multiprocessing`, и, как уже говорилось ранее, среди примеров к этой книге имеется программа *grepword-m.py*, использующая этот модуль, которая отличается от рассматривавшейся выше версии всего тремя строками. Похожее преобразование можно выполнить и в программе *finddduplicates-t.py*, рассматривавшейся в этом подразделе, но на практике применять этот модуль не рекомендуется. Несмотря на то, что модуль `multiprocessing` для упрощения подобных преобразований предлагает API (Application Programming Interface – прикладной программный интерфейс), близко совпадающий с API модуля `threading`, тем не менее эти два API не являются идентичными и имеют некоторые различия. Кроме того, чисто механический переход с использования модуля `threading` на использование модуля `multiprocessing` скорее всего возможен только для небольших и простых программ, таких как *grepword-t.py*, поэтому лучше разрабатывать программы, изначально опираясь на модуль `multiprocessing`. (В примерах к книге имеется программа *findduplicates-m.py*, она выполняет ту же работу, что и программа *findduplicates-t.py*, но делает это несколько иначе и использует модуль `multiprocessing`.)

В процессе разработки находится еще одно решение – версия интерпретатора CPython с поддержкой выполнения многопоточных программ. Самые свежие сведения о ходе разработки можно получить по адресу *www.code.google.com/p/python-threadsafe*.

В заключение

В этой главе было показано, как создавать программы, запускающие другие программы с помощью модуля `subprocess`, входящего в состав стандартной библиотеки. Программам, запускаемым с помощью модуля `subprocess`, можно передавать аргументы командной строки, поставлять данные через их стандартный поток ввода и получать результаты через их стандартный поток вывода (и через стандартный поток вывода сообщений об ошибках). Возможность создавать дочерние процессы

позволяет получить максимальную выгоду от наличия многоядерных процессоров и перекладывать работу по обеспечению параллельного выполнения нескольких процессов на плечи операционной системы. Недостаток такого подхода состоит в том, что при необходимости в нескольких процессах совместно использовать некоторые данные или синхронизировать их работу необходимо конструировать некоторый механизм взаимодействия, например, на основе разделяемой памяти (использованием модуля `mmap`), разделяемых файлов или на основе сетевых взаимодействий, что совершенно очевидно влечет за собой дополнительные хлопоты.

В этой главе также было показано, как создавать многопоточные программы. К сожалению, такие программы не имеют возможности использовать все преимущества наличия многоядерного процессора (если они работают под управлением стандартной реализации интерпретатора CPython), поэтому для языка Python использование нескольких процессов часто является более практичным решением, если дело касается производительности. Тем не менее мы видели, что модуль `queue` и механизмы блокировок языка Python, такие как `threading.Lock`, делают создание многопоточных программ простым делом, и что в случае простых программ, где достаточно применения объектов модуля `queue`, таких как `queue.Queue` и `queue.PriorityQueue`, можно вообще отказаться от явного использования блокировок.

Несмотря на то, что многопоточное программирование несомненно приобретает все большее распространение, тем не менее многопоточные программы доставляют больше хлопот при разработке, отладке и сопровождении, чем однопоточные. Однако в многопоточных программах проще организовать взаимодействие между потоками, например, посредством совместно используемых данных (при помощи классов из модуля `queue` или с использованием блокировок) и синхронизацию потоков (например, для сбора результатов), чем в случае нескольких процессов. Наличие нескольких потоков выполнения может оказаться весьма полезным в программах с графическим интерфейсом, которые должны производить длительные вычисления и сохранять возможность отклика на действия пользователя, включая возможность отменить выполнение задачи. Но в случае использования удобного механизма взаимодействий между процессами, такого как разделяемая память или очередь, доступная нескольким процессам, предлагаемая пакетом `multiprocessing`, использование нескольких процессов может оказаться более предпочтительной альтернативой многопоточным программам.

В следующей главе будет продемонстрирован еще один пример многопоточной программы сервера, которая обрабатывает каждый запрос клиента в отдельном потоке и использует блокировки для защиты совместно используемых данных.

Упражнения

1. Скопируйте и модифицируйте программу *grepword-p.py* так, чтобы дочерние процессы в ней ничего не выводили, а главная программа собирала бы результаты и после завершения всех дочерних процессов сортировала и выводила полученные данные. Для этого достаточно будет изменить только функцию `main()`, добавив три строки и изменив три строки. Для реализации этого упражнения придется проявить внимание и смекалку, и, возможно, вам потребуется прочитать документацию с описанием модуля `subprocess`. Решение приводится в файле *grepword-p_ans.py*.
2. Напишите многопоточную программу, которая читает содержимое файлов, имена которых перечислены в командной строке (и рекурсивно – содержимое файлов во всех каталогах, перечисленных в командной строке). Все файлы, которые являются файлами XML (то есть начинающиеся с символов «`<?xml`»), следует проанализировать с помощью парсера XML, и для каждого воспроизвести список уникальных тегов, используемых в файле, или вывести сообщение, если возникла какая-либо ошибка. Ниже приводится пример вывода программы:

```
./data/dvds.xml is an XML file that uses the following tags:
    dvd
    dvds
./data/bad.aix is an XML file that has the following error:
    mismatched tag: line 7889, column 2
./data/incidents.aix is an XML file that uses the following tags:
    airport
    incident
    incidents
    narrative
```

Самый простой способ написать такую программу состоит в том, чтобы модифицировать копию программы *findduplicates-t.py*, хотя вы, конечно, можете написать программу с самого начала. Небольшие изменения придется внести в метод `__init__()` и `__run__()` класса `Worker`, а метод `process()` придется переписать полностью (но для этого потребуется всего около двадцати строк). В функцию `main()` программы потребуется внести несколько упрощений, а функция `print_results()` будет выводить однострочный текст. Сообщение о порядке использования также потребуется изменить, чтобы оно выглядело, как показано ниже:

```
Usage: xmlsummary.py [options] [path]
outputs a summary of the XML files in path; path defaults to .

Options:
  -h, --help       show this help message and exit
  -t COUNT, --threads=COUNT
```

```
                    the number of threads to use (1..20) [default 7]
  -v, --verbose
  -d, --debug
(Порядок использования: xmlsummary.py [параметры] [путь] выводит
сведения о файлах XML в пути path; по умолчанию используется путь .

Параметры:
  -h, --help        показать это сообщение и выйти
  -t COUNT, --threads=COUNT
                    Число используемых потоков (1..20) [по умолчанию 7]
  -v, --verbose
  -d, --debug
)
```

Обязательно попробуйте запустить программу в отладочном режиме, чтобы проверить, как запускаются потоки и что каждый из них выполняет свою часть работы. Решение приводится в файле *xmlsummary.py*, в котором чуть больше 100 строк и не используются явные блокировки.

10

- Создание клиента TCP
- Создание сервера TCP

Сети

Сети позволяют компьютерным программам взаимодействовать друг с другом, даже если они выполняются на разных машинах. Для одних программ, таких как веб-броузеры, работа в сети является основным видом их деятельности, тогда как для других работа в сети является всего лишь дополнением к их функциональным возможностям – например, выполнение удаленных операций и регистрация событий или получение и передача данных другим машинам. Большинство сетевых программ работают либо по схеме точка-точка (когда одна и та же программа выполняется на разных машинах), либо по более общей схеме клиент/сервер (программы-клиенты отправляют запросы серверу).

В этой главе мы создадим простое приложение, работающее по схеме клиент/сервер. Такие приложения обычно состоят из двух отдельных программ: программы сервера, ожидающей поступления запросов и отвечающей на них, и одного или более клиентов, которые отправляют запросы серверу и получают от него ответы. Для обеспечения нормальной работы клиентов необходимо знать, как подключиться к серверу, то есть необходимо знать IP-адрес сервера и номер порта.[1] Кроме того, и клиент и сервер должны передавать и принимать данные в понятных им форматах.

Низкоуровневый модуль `socket` (на котором основаны все высокоуровневые сетевые модули в языке Python) поддерживает как адреса IPv4, так и адреса IPv6. Он также поддерживает наиболее широко используемые сетевые протоколы, включая UDP (User Datagram Protocol – про-

[1] Существует также возможность выполнять подключения с помощью механизма обнаружения служб, например, с использованием Bonjour API – соответствующие модули можно найти в каталоге пакетов Python Package Index, *pypi.python.org/pypi*.

токол пользовательских дейтаграмм) – легковесный, но ненадежный протокол, не обладающий поддержкой постоянных соединений, выполняющий передачу данных пакетами (дейтаграммами), но не гарантирующий их доставку адресату, и TCP (Transmission Control Protocol – протокол управления передачей) – надежный протокол, поддерживающий установку постоянных соединений и ориентированный на потоковый режим передачи данных. С помощью протокола TCP можно передавать и принимать любые объемы данных – сокет отвечает за деление данных на пакеты достаточно маленького размера, чтобы их можно было отправлять по сети, а также за их восстановление на другом конце соединения.

Протокол UDP часто используется в инструментах мониторинга, которые обеспечивают непрерывное чтение поступающих данных и для которых потеря некоторых пакетов не является существенной. Иногда этот протокол используется для передачи потокового аудио или видео, когда потеря отдельных кадров считается вполне допустимым явлением. Протоколы FTP и HTTP построены на базе протокола TCP, и обычно приложения, работающие в схеме клиент/сервер, используют протокол TCP, потому что им необходимы постоянные соединения и надежность, которые обеспечиваются протоколом TCP. В этой главе мы будем разрабатывать приложение, работающее в схеме клиент/сервер, поэтому мы будем использовать протокол TCP.

Еще нам необходимо определиться с тем, как будут выполняться отправка и получение данных – в виде текстовых строк или в виде блоков двоичных данных, а во втором случае – в каком формате. В этой главе мы будем использовать блоки двоичных данных, где первые четыре байта (кодированные как целые числа с использованием модуля `struct`) будут определять длину последующего блока данных в двоичном формате – в виде законсервированных объектов, которые воспроизводятся модулем `pickle`. Преимущество такого подхода состоит в том, что мы можем использовать один и тот же программный код для передачи и приема в *любых* приложениях, поскольку в виде законсервированных объектов можно сохранять практически любые данные. Недостаток такого подхода заключается в том, что и клиент и сервер должны уметь работать с законсервированными объектами, поэтому обе программы должны быть написаны на языке Python или обладать доступом к интерпретатору Python, например, Jython в Java или Boost.Python в C++. И, конечно, при использовании модуля `pickle` не следует забывать о проблеме безопасности.

Модуль pickle, стр. 341

Пример, который мы будем рассматривать, представляет собой программу регистрации автомобиля. На сервере хранится вся регистраци-

онная информация (номерной знак, количество посадочных мест, пробег и владелец). Программа-клиент будет использоваться для получения информации об автомобиле, позволяя изменять величину пробега или владельца, а также создавать новые регистрационные записи. Одновременно может использоваться любое число клиентских программ, и они не будут блокировать друг друга, даже если два клиента одновременно обратятся к серверу. Это обеспечивается тем, что сервер будет обслуживать каждого клиента в отдельном потоке. (Мы также увидим, насколько просто можно организовать обработку клиентов в отдельных процессах.)

Исключительно ради демонстрации мы будем запускать сервер и клиентов на одной и той же машине. Это означает, что будет использоваться «локальный» IP-адрес (хотя, если сервер выполняется на другой машине, клиенту можно передать ее IP-адрес в виде аргумента командной строки и они смогут взаимодействовать друг с другом при условии, что этому не будут препятствовать системы сетевой защиты). Кроме того, мы произвольно выбрали номер порта 9653. Номер порта должен быть больше 1023, и желательно, чтобы он находился в диапазоне от 5001 до 32787, хотя вполне допустимыми считаются номера портов вплоть до 65535.

Сервер может получать пять типов запросов: `GET_CAR_DETAILS`, `CHANGE_MILEAGE`, `CHANGE_OWNER`, `NEW_REGISTRATION` и `SHUTDOWN` и отправлять соответствующие ответы для каждого из них. В ответ передаются запрошенные данные, подтверждение выполнения запрошенного действия или признак ошибки.

Клиент TCP

Программа-клиент находится в файле *car_registration.py*. Ниже приводится пример сеанса взаимодействия (с уже запущенным сервером и слегка отредактированным меню, чтобы уместить его по ширине книжной страницы):

```
(C)ar (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]:
License: 024 hyr
License: 024 HYR
Seats:   2
Mileage: 97543
Owner:   Jack Lemon
(C)ar (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]: m
License [024 HYR]:
Mileage [97543]: 103491
Mileage successfully changed
```

Здесь жирным шрифтом выделены данные, которые были введены пользователем. Там, где ввод пользователя отсутствует, подразумевается, что пользователь нажал клавишу Enter, принимая значение по

умолчанию. В данном случае пользователь запросил информацию о конкретном автомобиле и затем обновил величину пробега.

С сервером могут работать столько клиентов, сколько мы пожелаем, и когда пользователь завершает работу клиентской программы, это никак не сказывается на работе сервера. Но если остановить сервер, то клиент, действиями которого была выполнена остановка, завершит свою работу, а все остальные клиенты при следующей попытке подключиться к серверу получат сообщение «Connection refused» (попытка соединения отвергнута) и их работа будет завершена. В более сложных приложениях возможность останавливать сервер может предоставляться только определенным пользователям и, возможно, с определенных машин, но мы включили такую возможность в клиентскую программу, чтобы показать, как это можно реализовать.

Теперь приступим к изучению программного кода, начав с функции `main()` и пользовательского интерфейса, а реализацию сетевых взаимодействий рассмотрим в конце примера.

```
def main():
    if len(sys.argv) > 1:
        Address[0] = sys.argv[1]
    call = dict(c=get_car_details, m=change_mileage, o=change_owner,
                n=new_registration, s=stop_server, q=quit)
    menu = ("(C)ar Edit (M)ileage Edit (O)wner (N)ew car "
            "(S)top server (Q)uit")
    valid = frozenset("cmonsq")
    previous_license = None
    while True:
        action = Console.get_menu_choice(menu, valid, "c", True)
        previous_license = call[action](previous_license)
```

Глобальный список `Address` хранит IP-адрес и номер порта в двух элементах `["localhost", 9653]`, где IP-адрес может быть переопределен, если он передается в виде аргумента командной строки. Словарь `call` отображает пункты меню на соответствующие им функции.

Ветвление с использованием словарей, стр. 395

Модуль `Console` – один из тех, что поставляются вместе с книгой; он содержит некоторые полезные функции получения значений, вводимых пользователем в консоли, такие как `Console.get_string()` и `Console.get_integer()`, – они похожи на функции, разработанные в первых главах, и были объединены в модуль, чтобы их было проще использовать в разных программах.

Для удобства пользователей мы запоминаем последний введенный номерной знак, чтобы его можно было использовать в качестве значения по умолчанию, потому что выполнение большинства команд начинается с запроса номерного знака соответствующего автомобиля. Как только пользователь сделает свой выбор, вызывается соответствующая функция, которой передается номерной знак, и ожидается, что

каждая функция будет возвращать использовавшийся номер. Так как цикл выполняется бесконечно, программа должна завершаться одной из функций; это будет показано ниже.

```
def get_car_details(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is not None:
        print("License: {0}\nSeats: {1[0]}\nMileage: {1[1]}\n"
              "Owner: {1[2]}".format(license, car))
    return license
```

Эта функция используется для получения информации о конкретном автомобиле. Поскольку большинству функций требуется запросить номер автомобиля у пользователя и часто необходимо получить некоторые сведения об автомобиле, мы вынесли эту функциональность в отдельную функцию `retrieve_car_details()` – она возвращает кортеж из двух элементов, содержащий номер автомобиля, введенный пользователем, и именованный кортеж `CarTuple`, хранящий количество посадочных мест, пробег и владельца (или предыдущий номер автомобиля и значение `None`, если был введен неопознанный номер). Здесь функция просто выводит полученную информацию и возвращает использовавшийся номер автомобиля, который будет применяться в качестве значения по умолчанию при вызове следующей функции, которой потребуется этот номер.

```
def retrieve_car_details(previous_license):
    license = Console.get_string("License", "license",
                                 previous_license)
    if not license:
        return previous_license, None
    license = license.upper()
    ok, *data = handle_request("GET_CAR_DETAILS", license)
    if not ok:
        print(data[0])
        return previous_license, None
    return license, CarTuple(*data)
```

Это первая функция, реализующая взаимодействие по сети. Она вызывает функцию `handle_request()`, которую мы рассмотрим немного ниже. Функция принимает `handle_request()` любые данные, полученные в качестве аргументов, отправляет их серверу и возвращает все, что было получено от сервера. Функция `handle_request()` ничего не знает об отправляемых и получаемых данных, она просто реализует сетевую службу.

В приложении регистрации автомобилей, в соответствии с принятым протоколом запросов, первым аргументом всегда посылается имя команды, которая должна быть выполнена сервером, а в остальных аргументах – параметры команды, в данном случае это номер автомобиля. Согласно протоколу ответов сервер всегда возвращает кортеж, первым элементом которого является логический флаг, свидетельствующий

об успехе операции. Если флаг имеет значение False, то во втором элементе кортежа содержится текст сообщения об ошибке. Если флаг имеет значение True, то кортеж будет состоять либо из двух элементов, где во втором элементе содержится текст подтверждения, либо из *n* элементов, где второй и все последующие элементы хранят запрошенные данные.

Далее, если номер автомобиля не будет опознан, переменная ok получит значение False, и тогда функция просто выведет текст сообщения об ошибке из data[0], а вызывающей программе будет возвращен предыдущий номер автомобиля. В противном случае будет возвращен текущий номер (который теперь становится предыдущим номером) и кортеж CarTuple, созданный из списка data (число мест, пробег, владелец).

```
def change_mileage(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is None:
        return previous_license
    mileage = Console.get_integer("Mileage", "mileage",
                                  car.mileage, 0)
    if mileage == 0:
        return license
    ok, *data = handle_request("CHANGE_MILEAGE", license, mileage)
    if not ok:
        print(data[0])
    else:
        print("Mileage successfully changed")
    return license
```

Эта функция следует той же схеме, что и get_car_details(), за исключением того, что после получения информации она обновляет один из элементов. Здесь фактически выполняется два сетевых запроса, поскольку функция retrieve_car_details() вызывает функцию handle_request() для получения информации об автомобиле. Здесь нам необходимо убедиться в корректности номера автомобиля и получить текущее значение пробега, которое будет использоваться как значение по умолчанию. В данном случае ответ сервера всегда будет представлять собой кортеж из двух элементов, во втором элементе которого будет содержаться либо текст сообщения об ошибке, либо None.

Мы не будем рассматривать функцию change_owner(), поскольку структурно она полностью повторяет функцию change_mileage(). Мы также не будем рассматривать функцию new_registration(), которая отличается лишь тем, что не пытается получить информацию об автомобиле вначале (так как вводится информация о новом автомобиле) и запрашивает у пользователя полный набор сведений. Для нас в этих функциях нет ничего нового и ничего, что было бы связано с программированием сетевых взаимодействий.

```
def quit(*ignore):
    sys.exit()
```

```
def stop_server(*ignore):
    handle_request("SHUTDOWN", wait_for_reply=False)
    sys.exit()
```

Если пользователь выбирает команду завершения программы, мы производим выход вызовом функции `sys.exit()`. Каждая функция, соответствующая некоторому пункту меню, получает в виде аргумента предыдущий номер автомобиля, но в данном случае в нем нет никакой необходимости. Мы не можем просто написать `def quit():`, потому что в результате будет создана функция, не ожидающая аргументов, и при вызове такой функции с предыдущим номером автомобиля будет возбуждено исключение `TypeError`, свидетельствующее о том, что функция получила аргумент, который она не ожидала получить. Поэтому мы просто определили параметр `*ignore`, который способен принять любое число позиционных аргументов. Само имя `ignore` ничего не значит для интерпретатора и используется исключительно для того, чтобы показать тому, кто будет сопровождать программу, что аргументы игнорируются функцией.

Если пользователь выбирает команду остановки сервера, мы с помощью функции `handle_request()` извещаем сервер и указываем при этом, что не ждем ответа. Сразу после отправки данных функция `handle_request()` возвращает управление, не ожидая ответа, и мы завершаем работу программы вызовом функции `sys.exit()`.

```
def handle_request(*items, wait_for_reply=True):
    SizeStruct = struct.Struct("!I")
    data = pickle.dumps(items, 3)

    try:
        with SocketManager(tuple(Address)) as sock:
            sock.sendall(SizeStruct.pack(len(data)))
            sock.sendall(data)
            if not wait_for_reply:
                return

            size_data = sock.recv(SizeStruct.size)
            size = SizeStruct.unpack(size_data)[0]
            result = bytearray()
            while True:
                data = sock.recv(4000)
                if not data:
                    break
                result.extend(data)
                if len(result) >= size:
                    break
        return pickle.loads(result)
    except socket.error as err:
        print("{0}: is the server running?".format(err))
        sys.exit(1)
```

Эта функция реализует в программе все сетевые взаимодействия. Она начинается с создания объекта `struct.Struct`, который хранит одно целое число без знака с сетевым порядком следования байтов, после чего создается законсервированный объект, содержащий все полученные элементы данных. Функция ничего не знает и не делает никаких предположений об этих элементах данных. Обратите внимание, что здесь явно определяется протокол консервирования 3, чтобы гарантировать, что сервер и клиент будут использовать одну и ту же версию протокола, даже если на стороне сервера или на стороне клиента была обновлена версия интерпретатора Python.

Если бы нам было необходимо обеспечить возможность дальнейшего развития протокола обмена, мы могли бы предусмотреть передачу его номера версии (как мы делали это с двоичными форматами сохранения данных на диск). Это можно реализовать как на сетевом уровне, так и на уровне данных. На сетевом уровне мы могли бы предусмотреть передачу номера версии в виде второго целого числа без знака, то есть передавать длину блока данных и номер версии протокола. На уровне данных мы могли бы следовать соглашению, согласно которому законсервированный объект всегда является списком (или словарем), в котором первый элемент (или элемент с ключом «version») содержит номер версии. (Вы получите шанс добавить версию протокола, когда будете решать упражнения.)

Класс `SocketManager` – это наш собственный менеджер контекста, представляющий собой сокет, готовый к использованию, – мы рассмотрим его чуть ниже. Метод `socket.socket.sendall()` отправляет все переданные ему данные, выполняя за кулисами столько вызовов `socket.socket.send()`, сколько потребуется. Программа всегда передает серверу два элемента данных: длину законсервированного объекта и сам объект. Если аргумент `wait_for_reply` имеет значение `False`, функция не ждет получения ответа от сервера и немедленно возвращает управление – менеджер контекста гарантирует, что сокет будет закрыт до того, как функция фактически вернет управление.

После отправки данных (когда требуется получить ответ) вызывается метод `socket.socket.recv()`, который принимает ответ. Этот метод блокирует выполнение программы, пока не примет данные. Первым вызовом принимаются четыре байта – столько отводится под целое число со значением размера законсервированного объекта, следующего за числом. Здесь с помощью объекта `struct.Struct` выполняется распаковывание байтов в целое число `size`. Затем создается пустой объект `bytearray` и производится попытка получить входящий законсервированный объект блоками размером до 4 000 байтов. Как только будет прочитано `size` байтов (или если данные были исчерпаны до этого момента), производится выход из цикла и распаковывание данных с помощью функции `pickle.loads()` (которая принимает объект `bytes` или `bytearray`), после чего данные возвращаются вызывающей программе.

В данном случае известно, что данные всегда имеют форму кортежа, так как это определяется протоколом взаимодействия с сервером приложения регистрации автомобилей, но функция handle_request() ничего не знает и не делает никаких предположений о типе получаемых данных.

Если при попытке выполнить сетевое взаимодействие что-то пойдет не так, например, сервер окажется не запущен или по каким-либо причинам соединение будет разорвано, будет возбуждено исключение socket.error. В этом случае исключение будет перехвачено, клиентская программа выведет сообщение об ошибке и завершит работу.

```
class SocketManager:

    def __init__(self, address):
        self.address = address

    def __enter__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, *ignore):
        self.sock.close()
```

Объект address – это кортеж из двух элементов (IP-адрес и номер порта); он определяется при создании менеджера контекста. Как только инструкция with задействует менеджер контекста, он создаст сокет и попытается установить соединение, при этом выполнение программы будет заблокировано, пока соединение не будет установлено или пока сокет не возбудит исключение. Первый аргумент функции инициализации объекта socket.socket() определяет семейство адресов – в данном случае используется семейство socket.AF_INET (IPv4), однако имеются и другие семейства – например, socket.AF_INET6 (IPv6), socket.AF_UNIX и socket.AF_NETLINK. Второй аргумент – это обычно либо socket.SOCK_STREAM (TCP), как в данном случае, либо socket.SOCK_DGRAM (UDP).

Когда поток управления покидает область видимости инструкции with, вызывается метод __exit__() объекта контекста. Мы не заботимся о том, возникло исключение или нет (поэтому мы игнорируем аргументы с информацией об исключении), и просто закрываем сокет. Поскольку метод возвращает None (False – в логическом контексте), любое возникшее исключение продолжит свое распространение – на этот случай мы предусмотрели соответствующий блок except в функции handle_request(), который обработает любое исключение, возникшее в сокете.

Сервер TCP

Поскольку в большинстве случаев программный код реализации серверов следует одному и тому же шаблону, вместо низкоуровневого моду-

ля socket мы будем использовать высокоуровневый модуль socketserver, который выполнит всю рутинную работу за нас. Все, что от нас требуется, это реализовать класс обработчика запросов с методом handle(), который будет использоваться для чтения запросов и записи ответов. Модуль socketserver сам выполняет все необходимые взаимодействия, обслуживая запросы на соединение либо по очереди, либо передавая их своим отдельным потокам или процессам, причем все это делается абсолютно прозрачно, что избавляет нас от необходимости погружаться в детали низкоуровневой обработки.

Программа-сервер для данного приложения находится в файле *car_registration_server.py*.[1] Эта программа содержит определение очень простого класса Car, который хранит информацию о количестве мест, пробеге и владельце в виде свойств (первое из них доступно только для чтения). Класс не хранит информацию о номерном знаке, потому что номер хранится в словаре, где он используется в качестве ключа.

Начнем с того, что рассмотрим функцию main(), затем коротко познакомимся с тем, как сервер загружает данные, далее рассмотрим создание класса сервера и, наконец, реализацию класса обработчика запросов, обрабатывающего запросы клиентов.

```
def main():
    filename = os.path.join(os.path.dirname(__file__),
                            "car_registrations.dat")
    cars = load(filename)
    print("Loaded {0} car registrations".format(len(cars)))
    RequestHandler.Cars = cars
    server = None
    try:
        server = CarRegistrationServer(("", 9653), RequestHandler)
        server.serve_forever()
    except Exception as err:
        print("ERROR", err)
    finally:
        if server is not None:
            server.shutdown()
            save(filename, cars)
            print("Saved {0} car registrations".format(len(cars)))
```

Регистрационная информация об автомобилях хранится в том же каталоге, что и сама программа. В объект cars записывается ссылка на словарь, ключами которого являются строки с номерами автомобилей, а значениями – объекты типа Car. Обычно серверы ничего не выводят на экран, потому что обычно они запускаются и останавливают-

[1] При первом запуске сервера в операционной системе Windows может появиться диалог брандмауэра, сообщающий о том, что Python заблокирован – щелкните на кнопке Разблокировать (Unblock), чтобы дать серверу возможность работать.

ся автоматически, а выполняются в фоновом режиме. По этой причине они, как правило, сообщают о своем состоянии посредством записи сообщений в файлы журналов (например, с помощью модуля `logging`). Здесь мы решили выводить на экран сообщения при запуске и остановке, чтобы упростить тестирование и экспериментирование.

Наш класс обработчика запросов должен иметь возможность обращаться к словарю `cars`, но мы не можем передавать словарь экземплярам этого класса, так как они будут создаваться сервером без нашего участия – по одному обработчику на каждый запрос. Поэтому мы записываем ссылку на словарь в атрибут класса `RequestHandler.Cars`, который обеспечит доступ к словарю всем экземплярам класса.

При создании экземпляра сервера ему передаются адрес и порт для выполнения сетевых взаимодействий, а также класс `RequestHandler` – сам класс, а не его экземпляр. Пустая строка адреса соответствует любому доступному адресу IPv4 (включая текущий сетевой адрес машины и локальный адрес `localhost`). Затем серверу сообщается, что он должен выполнять обслуживание запросов без остановки. Когда сервер останавливается (как это происходит, мы увидим немного ниже), он сохраняет словарь `cars`, поскольку информация в нем могла быть изменена клиентами.

```
def load(filename):
    try:
        with contextlib.closing(gzip.open(filename, "rb")) as fh:
            return pickle.load(fh)
    except (EnvironmentError, pickle.UnpicklingError) as err:
        print("server cannot load data: {1}".format(err))
        sys.exit(1)
```

Загрузка выполняется очень просто благодаря тому что здесь используется менеджер контекста из модуля `contextlib`, входящего в состав стандартной библиотеки, который гарантирует закрытие файла независимо от того, произошло исключение или нет. Другой способ добиться того же эффекта состоит в том, чтобы использовать собственный менеджер контекста. Например:

```
class GzipManager:

    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.fh = gzip.open(self.filename, self.mode)
        return self.fh

    def __exit__(self, *ignore):
        self.fh.close()
```

При использовании собственного менеджера `GzipManager` инструкция `with` приобретает вид:

```
with GzipManager(filename, "rb") as fh:
```

Функция `save()` (здесь она не показана) по своей структуре очень похожа на функцию `load()`, только она открывает файл в двоичном режиме для записи, сохраняет данные с помощью функции `pickle.dump()` и ничего не возвращает.

```
class CarRegistrationServer(socketserver.ThreadingMixIn,
                            socketserver.TCPServer): pass
```

Множественное наследование, стр. 449

Это полное определение нашего собственного класса сервера. Если появится необходимость создать сервер, который будет обслуживать запросы в отдельных процессах, а не в потоках, достаточно будет лишь унаследовать класс `socketserver.ForkingMixIn` вместо класса `socketserver.ThreadingMixIn`. Термин *mixin* (*смесь*) часто используется для описания классов, специально предназначенных для множественного наследования. Классы из модуля `socketserver` могут использоваться для создания самых разнообразных серверов, включая серверы UDP и серверы UDP и TCP в операционной системе UNIX, посредством наследования соответствующей пары базовых классов.

Обратите внимание, что класс-смесь из модуля `socketserver` всегда должен наследоваться первым. Это гарантирует, что методы класса-смеси будут пользоваться преимуществом перед методами второго наследуемого класса при наличии одноименных методов в обоих классах, поскольку интерпретатор Python выполняет поиск методов в базовых классах в том порядке, в каком они перечислены в определении класса, и использует первый найденный метод.

Для обработки каждого запроса сервер создает обработчик запросов (используя переданный ему класс). Наш собственный класс `RequestHandler` реализует методы для обработки каждого типа запросов плюс обязательный метод `handle()`, который вызывается классом сервера. Но прежде чем перейти к рассмотрению этих методов, познакомимся с объявлением класса и с атрибутами класса.

```
class RequestHandler(socketserver.StreamRequestHandler):

    CarsLock = threading.Lock()
    CallLock = threading.Lock()
    Call = dict(
            GET_CAR_DETAILS=(
                    lambda self, *args: self.get_car_details(*args)),
            CHANGE_MILEAGE=(
                    lambda self, *args: self.change_mileage(*args)),
            CHANGE_OWNER=(
                    lambda self, *args: self.change_owner(*args)),
            NEW_REGISTRATION=(
```

```
                lambda self, *args: self.new_registration(*args)),
        SHUTDOWN=lambda self, *args: self.shutdown(*args))
```

Мы наследуем класс `socketserver.StreamRequestHandler`, потому что будем использовать потоковый (TCP) сервер. Для создания серверов UDP можно использовать класс `socketserver.DatagramRequestHandler`, а для обеспечения низкоуровневого доступа можно было бы наследовать класс `socketserver.BaseRequestHandler`.

Словарь `RequestHandler.Cars` – это атрибут класса, который добавляется в функции `main()`. Он хранит все регистрационные данные. Добавление дополнительных атрибутов в объекты (такие как классы или экземпляры) может производиться за пределами объявления класса (в данном случае – в функции `main()`) без каких-либо ограничений (при условии, что объект имеет атрибут `__dict__`), что может быть очень удобно. Так как заранее известно, что класс зависит от этого атрибута, можно было бы добавить строку `Cars = None` в объявление, чтобы зафиксировать существование переменной.

Почти каждый метод, обрабатывающий запросы, должен иметь доступ к словарю `Cars`, но нам следует гарантировать, что обращение к нему никогда не будет происходить из двух методов (из двух разных потоков) одновременно, в противном случае это может привести к повреждению словаря или к аварийному завершению программы. Чтобы избежать этих неприятностей, мы предусмотрели блокировку, оформленную в виде атрибута класса. С ее помощью мы сможем гарантировать, что в каждый конкретный момент времени только один поток будет иметь доступ к словарю.[1] (Принципы разработки многопоточных программ, включая использование блокировок, рассматриваются в главе 9.)

Словарь `Call` – это еще один атрибут класса. Каждый ключ это словаря является именем операции, которую может выполнять сервер, а каждое значение – это функция, выполняющая соответствующую операцию. Мы не можем использовать методы непосредственно, как это было сделано с функциями в словаре, обслуживающем меню в клиентской программе, потому что на уровне класса отсутствует ссылка `self`. Чтобы решить эту проблему, мы использовали функции-обертки, которые получают значение ссылки `self` в момент вызова и затем вызывают соответствующий метод, передавая ему ссылку `self` и любые другие аргументы. Как вариант, словарь `Call` можно было бы создать *после* определения всех методов. Это позволило бы создать такие записи, как `GET_CAR_DETAILS=get_car_details`, и интерпретатор Python сумел бы

[1] Глобальная блокировка интерпретатора (Global Interpreter Lock, GIL) гарантирует синхронизированный доступ к словарю `Cars`, но, как отмечалось ранее, мы не можем полагаться на нее, так как она является особенностью реализации CPython.

Глобальная блокировка GIL, стр. 478

отыскать метод get_car_details(), так как словарь создается после того, как все методы были определены. Но мы предпочли использовать первый вариант, так как он более явный и не зависит от того, в каком месте класса находится определение словаря.

Несмотря на то, что после создания класса словарь Call всегда будет использоваться только для чтения, тем не менее, учитывая, что словарь является изменяемым объектом, мы решили обеспечить дополнительный уровень безопасности и определили блокировку, которая гарантирует, что в каждый конкретный момент времени доступ к нему будет иметь только один поток. (Она также не является обязательной из-за глобальной блокировки, присутствующей в реализации CPython.)

Глобальная блокировка GIL, стр. 478

```
def handle(self):
    SizeStruct = struct.Struct("!I")
    size_data = self.rfile.read(SizeStruct.size)
    size = SizeStruct.unpack(size_data)[0]
    data = pickle.loads(self.rfile.read(size))

    try:
        with self.CallLock:
            function = self.Call[data[0]]
        reply = function(self, *data[1:])
    except Finish:
        return
    data = pickle.dumps(reply, 3)
    self.wfile.write(SizeStruct.pack(len(data)))
    self.wfile.write(data)
```

Всякий раз, когда клиент выполняет запрос, создается новый поток с новым экземпляром класса RequestHandler, после чего вызывается метод handle() экземпляра. Внутри этого метода данные, полученные от клиента, можно прочитать с помощью объекта файла self.rfile, а отправка данных клиенту может быть выполнена с помощью объекта файла self.wfile. Оба эти объекта предоставляются модулем socketserver уже открытыми и готовыми к использованию.

Объект типа struct.Struct – это целочисленный счетчик байтов, который необходим, чтобы прочитать формат «длина плюс данные», используемый при обмене данными между клиентами и сервером.

Сначала выполняется чтение первых четырех байтов и распаковывание их в целое число size, чтобы узнать размер законсервированного объекта, отправленного клиентом. Затем выполняется чтение size байтов и распаковывание их в переменную data. Операция чтения блокирует дальнейшее выполнение, пока данные не будут прочитаны. В данном случае известно, что данные всегда поступают в виде кортежа, первый элемент которого определяет выполняемую операцию,

а остальные элементы являются параметрами этой операции, потому что это определено протоколом, установленным для клиентов.

Внутри блока `try` мы получаем лямбда-функцию, соответствующую запрошенной операции. Доступ к словарю `Call` выполняется под защитой блокировки, хотя вполне возможно, что мы проявляем излишнюю осторожность. Как обычно, мы стараемся выполнять как можно меньше действий в области видимости блокировки – в данном случае мы всего лишь отыскиваем в словаре ссылку на требуемую функцию. Как только функция будет получена, она тут же вызывается и ей передаются в виде первого аргумента – ссылка `self` и в виде последующих аргументов – остальное содержимое кортежа `data`. Здесь производится вызов функции, поэтому ссылка `self` не передается ей интерпретатором. Это не имеет большого значения, так как мы сами передаем ссылку `self`, а внутри лямбда-функции полученная ссылка `self` используется для вызова метода обычным способом. В результате производится вызов метода `self.`*`method`*`(*data[1:])`, где *`method`* – это метод, соответствующий операции, указанной в `data[0]`.

Если выбрана операция остановки сервера, в методе `shutdown()` возбуждается наше собственное исключение `Finish` – в этой ситуации известно, что клиент не ожидает ответа, поэтому мы просто можем вернуть управление. Но для остальных операций выполняется консервирование (с использованием протокола 3) объекта с результатами работы метода, соответствующего запрошенной операции, и запись – сначала размера законсервированного объекта, а затем самого объекта с данными.

```
def get_car_details(self, license):
    with self.CarsLock:
        car = copy.copy(self.Cars.get(license, None))
    if car is not None:
        return (True, car.seats, car.mileage, car.owner)
    return (False, "This license is not registered")
```

Этот метод начинается с того, что пытается получить блокировку для доступа к данным и блокируется, пока не получит ее. Затем с помощью метода `dict.get()`, которому вторым аргументом передается значение `None`, он получает сведения об автомобиле с указанным номером или `None`. Информация тут же копируется, и инструкция `with` завершается. Этим обеспечивается максимально короткое время, в течение которого блокировка будет занята. Операция чтения не изменяет словарь, но, так как мы имеем дело с изменяемой коллекцией, вполне возможно, что какой-нибудь другой поток в то же самое время попробует изменить словарь – использование блокировки устраняет такую опасность. Теперь, когда мы находимся вне области действия блокировки, у нас имеется объект с копией информации об автомобиле (или `None`), с которой можно продолжить работу, не блокируя другие потоки.

Все методы, выполняющие операции с регистрационной информацией, возвращают кортеж, первым элементом которого является логический флаг – признак успешного выполнения операции, а количество и значения остальных элементов зависят от конкретного метода. Ни один из этих методов даже понятия не имеет и не делает никаких предположений о данных, возвращаемых клиенту, кроме того, что эти данные представляют собой «кортеж, первым элементом которого является логический флаг», так как все сетевые взаимодействия сосредоточены в методе `handle()`.

```
def change_mileage(self, license, mileage):
    if mileage < 0:
        return (False, "Cannot set a negative mileage")
    with self.CarsLock:
        car = self.Cars.get(license, None)
        if car is not None:
            if car.mileage < mileage:
                car.mileage = mileage
                return (True, None)
        return (False, "Cannot wind the odometer back")
    return (False, "This license is not registered")
```

В этом методе мы можем выполнить одну проверку, не прибегая к использованию блокировки. Но если величина пробега неотрицательна, необходимо приобрести блокировку и получить ссылку на объект с информацией о соответствующем автомобиле, и если требуемый объект существует (например, если указан верный номер автомобиля), необходимо остаться в области действия блокировки, чтобы изменить величину пробега в соответствии с запросом клиента, в противном случае вернуть кортеж с сообщением об ошибке. Если для запрошенного номера автомобиля в словаре отсутствует информация (`car == None`), метод выходит из области видимости инструкции `with` и возвращает кортеж с сообщением об ошибке.

На первый взгляд, проверку величины пробега можно выполнить на стороне клиента, чтобы уменьшить объем сетевого трафика – например, клиент мог бы получать сообщение об ошибке при вводе отрицательной величины пробега (или такая возможность просто предотвращалась бы). Но даже если мы обяжем сторону клиента реализовать такую проверку, мы по-прежнему должны проверять данные на стороне сервера, так как нельзя быть полностью уверенным, что клиентская программа не содержит ошибок. И хотя программа-клиент получает величину пробега автомобиля, чтобы использовать ее в качестве значения по умолчанию, мы не должны считать величину пробега, указанную пользователем, верной (даже если она больше текущей величины), потому что какой-нибудь другой клиент мог увеличить это значение к текущему моменту времени. Из всего этого следует, что окончательная проверка данных может выполняться только на стороне сервера и только под защитой блокировки.

Метод change_owner() очень похож на предыдущий, поэтому мы не будем приводить его здесь.

```
def new_registration(self, license, seats, mileage, owner):
    if not license:
        return (False, "Cannot set an empty license")
    if seats not in {2, 4, 5, 6, 7, 8, 9}:
        return (False, "Cannot register car with invalid seats")
    if mileage < 0:
        return (False, "Cannot set a negative mileage")
    if not owner:
        return (False, "Cannot set an empty owner")
    with self.CarsLock:
        if license not in self.Cars:
            self.Cars[license] = Car(seats, mileage, owner)
            return (True, None)
    return (False, "Cannot register duplicate license")
```

И снова, прежде чем обратиться к словарю с регистрационными данными, мы можем выполнить множество проверок. Если все данные имеют допустимые значения, необходимо получить блокировку. Если номер автомобиля отсутствует в словаре RequestHandler.Cars (а он должен отсутствовать, так как при создании новой регистрационной записи должен указываться незарегистрированный номер), создается новый объект Car и сохраняется в словаре. Все это должно выполняться под защитой блокировки, потому что мы не должны допускать возможность добавления одного и того же номера автомобиля в момент времени между проверкой присутствия номера автомобиля в словаре RequestHandler.Cars и добавлением нового объекта Car в словарь.

```
def shutdown(self, *ignore):
    self.server.shutdown()
    raise Finish()
```

Если запрошена операция остановки сервера, вызывается метод shutdown() сервера – это приведет к прекращению приема последующих запросов, но сам сервер продолжит обработку уже принятых запросов. После этого мы возбуждаем собственное исключение, чтобы известить метод handler(), что работу сервера следует прекратить. Это приведет к тому, что метод handler() вернет управление вызывающей программе, не посылая ответ клиенту.

В заключение

В этой главе было показано, что сетевые клиенты и серверы реализуются довольно просто благодаря наличию в стандартной библиотеке языка Python сетевых модулей и модулей struct и pickle.

В первом разделе мы разработали клиентскую программу и создали в ней единственную функцию handle_request(), способную отправлять

на сервер и получать от сервера произвольные законсервированные объекты с данными в универсальном формате «длина плюс законсервированный объект». Во втором разделе мы увидели, как можно создать подкласс сервера, используя классы из модуля `socketserver`, и как реализовать класс обработчика запросов для обслуживания запросов клиентов. Здесь также все сетевые взаимодействия были сосредоточены в единственном методе `handle()`, который принимает от клиентов и отправляет клиентам произвольные законсервированные объекты с данными.

Модули `socket` и `socketserver`, а также многие другие модули из стандартной библиотеки, такие как `asyncore`, `asynchat` и `ssl`, предоставляют намного больше функциональных возможностей, чем мы использовали здесь. Но если окажется, что средств сетевых взаимодействий, имеющихся в стандартной библиотеке, недостаточно или они недостаточно высокоуровневые, то стоит обратить внимание сетевую платформу `Twisted`, созданную сторонними разработчиками (*www.twistedmatrix.com*), как на возможную альтернативу.

Упражнения

В упражнения предлагается модифицировать программы клиента и сервера, описанные в этой главе. Модификации не требуют ввода больших объемов программного кода, но требуют проявить долю внимания, чтобы найти правильное решение.

1. Скопируйте программы *car_registration_server.py* и *car_registration.py* и измените их так, чтобы они использовали версию протокола обмена на сетевом уровне. Реализовать это можно, например, передавая два целых числа (длина, версия протокола) вместо одного.

   Для этого потребуется добавить или изменить порядка десяти строк в функции `handle_request()` в клиентской программе, а также добавить или изменить порядка шестнадцати строк в методе `handle()` в серверной программе, включая программный код, обрабатывающий ситуацию, когда номер версии протокола не соответствует ожидаемому.

2. Скопируйте программу *car_registration_server.py* (или используйте ту, что была разработана в упражнении 1) и модифицируйте ее так, чтобы она выполняла новую операцию `GET_LICENSES_STARTING_WITH`. Операция должна принимать один строковый параметр. Метод, реализующий операцию, должен всегда возвращать кортеж из двух элементов (`True`, *список номеров*). Ситуация ошибки в данном случае невозможна, так как отсутствие совпадений не является ошибкой, и в этом случае клиенту должны возвращаться значение `True` и пустой список.

   Извлечение номеров автомобилей (ключей словаря `RequestHandler.Cars`) должно выполняться под защитой блокировки, но все ос-

тальные действия должны выполняться за пределами блокировки, чтобы минимизировать время блокировки словаря. Один из эффективных способов поиска соответствующих номеров автомобилей заключается в том, чтобы отсортировать список ключей, затем с помощью модуля `bisect` отыскать первое совпадение и потом выполнять итерации, начиная с найденного элемента. Другое возможное решение состоит в том, чтобы выполнить итерации по списку номеров, выбирая те, что начинаются с указанной строки, возможно, с применением генератора списков.

Помимо дополнительной инструкции `import`, необходимо будет добавить пару строк программного кода, записывающего ссылку на реализацию операции в словарь `Call`. Размер метода реализации операции не будет превышать десятка строк. Это совсем несложно, но потребует проявить определенное внимание. Решение, использующее модуль `bisect`, приводится в файле *car_registration_server_ans.py*.

3. Скопируйте программу *car_registration.py* (или используйте ту, что была разработана в упражнении 1) и модифицируйте ее так, чтобы задействовать достоинства нового сервера (*car_registration_server_ans.py*). Это подразумевает внесение изменений в функцию `retrieve_car_details()`, чтобы в случае ввода неправильного номера автомобиля запросить у пользователя начальные символы номерного знака и предложить ему выбрать номер из полученного списка. Ниже приводится пример сеанса взаимодействия пользователя с программой с использованием новой функции (сервер уже запущен, меню немного отредактировано, чтобы уместить его по ширине книжной страницы, и ввод пользователя выделен жирным шрифтом):

```
(C)ar (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]:
License: da 4020
License: DA 4020
Seats: 2
Mileage: 97181
Owner: Jonathan Lynn
(C)ar (M)ileage (O)wner (N)ew car (S)top server (Q)uit [c]:
License [DA 4020]: z
This license is not registered
Start of license: z
No licence starts with Z
Start of license: a
(1) AO4 4HE
(2) A37 4791
(3) ABK3035
Enter choice (0 to cancel): 3
License: ABK3035
Seats: 5
Mileage: 17719
Owner: Anthony Jay
```

Для решения этого упражнения необходимо будет удалить одну строку и добавить порядка двадцати строк. Потребуется немного поломать голову, чтобы предоставить пользователю возможность выйти или продолжить на каждом этапе. Обязательно протестируйте новые функциональные возможности в разных ситуациях (отсутствие номеров, начинающихся с указанной строки; имеется всего один номер, начинающийся с указанной строки; имеется два или более номеров, начинающихся с указанной строки). Решение приводится в файле *car_registration_ans.py*.

11

- Базы данных DBM
- Базы данных SQL

Программирование приложений баз данных

Большинство разработчиков программного обеспечения под термином *база данных* подразумевают СУРБД (система управления реляционными базами данных). Для хранения данных эти системы используют таблицы (по своему строению подобные электронным таблицам), строки которых соответствуют записям, а столбцы – полям. Манипулирование данными, хранящимися в этих таблицах, производится с помощью инструкций, написанных на языке SQL (Structured Query Language – язык структурированных запросов). Язык Python включает в себя API (Application Programming Interface – прикладной программный интерфейс) для работы с базами данных SQL и обычно распространяется с поддержкой базы данных SQLite 3.

Существует еще одна разновидность баз данных – *DBM* (Database Manager – система управления базами данных), в которой данные хранятся в виде произвольного числа элементов ключ-значение. В стандартной библиотеке Python имеется несколько интерфейсов для работы с разными реализациями DBM, включая характерные для операционной системы UNIX. Базы данных DBM работают по принципу словарей в языке Python, за исключением того, что обычно они хранятся на диске, а не в памяти, а их ключами и значениями всегда являются объекты типа `bytes`, размер которых может ограничиваться. В первом разделе этой главы рассматривается модуль `shelve`, предоставляющий удобный интерфейс DBM, позволяя использовать строковые ключи и любые (поддающиеся консервированию) объекты в качестве значений.

Если имеющихся в наличии баз данных DBM и SQLite окажется недостаточно, в каталоге пакетов Python Package Index, *pypi.python.org/pypi*, можно найти множество пакетов, предназначенных для работы

с различными базами данных, включая bsddb DBM («Berkeley DB»), объектно-реляционные отображения, такие как SQLAlchemy (*www.sqlalchemy.org*), и интерфейсы к популярным клиент/серверным базам данных, таким как DB2, Informix, Ingres, MySQL, ODBC и PostgreSQL.

В этой главе мы реализуем две версии программы ведения списка фильмов на дисках DVD, в котором будут храниться название фильма, год выхода, продолжительность в минутах и имя режиссера. Первая версия программы для хранения информации использует базу данных DBM (с помощью модуля `shelve`), а вторая версия – базу данных SQLite. Обе программы могут также загружать и сохранять информацию в простом формате XML, обеспечивая возможность, например, экспортировать сведения о фильмах из одной программы и импортировать их в другую. Версия, использующая базу данных SQL, обладает немного более широкими возможностями, чем версия, использующая базу данных DBM, и имеет более ясную организацию.

Базы данных DBM

Модуль `shelve` представляет собой обертку вокруг DBM, позволяя нам взаимодействовать с базой данных DBM, как с обычным словарем, в котором в качестве ключей допускается использовать только строки, а в качестве значений – объекты, допускающие возможность консервирования. За кулисами модуль `shelve` преобразует ключи и значения в объекты типа `bytes` и обратно.

Тип данных bytes, стр. 344

Поскольку модуль `shelve` основан на использовании лучшей из доступных баз данных DBM, есть вероятность, что файл DBM, сохраненный на одной машине, не будет читаться на другой, если на другой машине отсутствует поддержка той же самой DBM. Наиболее типичное решение такой проблемы состоит в том, чтобы обеспечить возможность импорта и экспорта данных в формате XML для файлов, которые должны быть переносимыми с машины на машину. Именно это мы и реализуем в программе *dvds-dbm.py* в этом разделе.

В качестве ключей мы будем использовать названия фильмов на дисках DVD, а в качестве значений – кортежи, в которых будут храниться имя режиссера, год и продолжительность фильма. Благодаря модулю `shelve` нам не придется выполнять каких-либо преобразований данных, и мы можем воспринимать объект DBM как обычный словарь.

Так как по своей структуре программа похожа на интерактивные программы, управляемые с помощью меню, которые мы уже видели ранее, мы сосредоточимся исключительно на аспектах, связанных с программированием DBM. Ниже приводится фрагмент из функции `main()`, где был опущен программный код, выполняющий обработку меню:

```
db = None
try:
    db = shelve.open(filename, protocol=pickle.HIGHEST_PROTOCOL)
    ...
finally:
    if db is not None:
        db.close()
```

Здесь открывается (или создается, если он еще не существует) указанный файл DBM в режиме для чтения и для записи. Значение каждого элемента сохраняется в файле в виде объекта, законсервированного с использованием указанного протокола консервирования. Существующие элементы можно будет прочитать, даже если они были сохранены с использованием меньшего номера протокола, поскольку интерпретатор в состоянии определять правильный номер протокола при чтении законсервированных объектов. В конце функции файл DBM закрывается, в результате происходит очистка внутреннего кэша DBM, производится запись всех изменений на диск и собственно закрытие файла.

Программа предоставляет возможность добавлять, редактировать, просматривать, импортировать и экспортировать данные. Мы пропустим процедуры импортирования и экспортирования данных в формате XML, поскольку они очень похожи на те, что мы рассматривали в главе 7. Точно так же мы опустим большую часть программного кода реализации пользовательского интерфейса, кроме операции добавления, потому что мы видели его прежде в других контекстах.

```
def add_dvd(db):
    title = Console.get_string("Title", "title")
    if not title:
        return
    director = Console.get_string("Director", "director")
    if not director:
        return
    year = Console.get_integer("Year", "year", minimum=1896,
                               maximum=datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   minimum=0, maximum=60*48)
    db[title] = (director, year, duration)
    db.sync()
```

Этой функции, как и любой другой, вызываемой из меню программы, передается в качестве единственного параметра объект DBM (`db`). Большая часть функции связана с получением данных о диске DVD и только в последней строке производится сохранение элемента ключ-значение в файле DBM, где в качестве ключа используется название фильма, а в качестве значения – кортеж с именем режиссера, годом выпуска и продолжительностью (который консервируется средствами модуля `shelve`).

Для сохранения непротиворечивости, свойственной языку Python, механизмы DBM предоставляют тот же самый API, что и словари, поэтому нам не придется осваивать новый синтаксис помимо функции `shelve.open()`, которую мы уже видели выше, и метода `shelve.Shelf.sync()`, который используется для очистки внутреннего кэша модуля `shelve` и синхронизации данных, находящихся в дисковом файле с последними изменениями, – в данном случае просто добавляется новый элемент.

```
def edit_dvd(db):
    old_title = find_dvd(db, "edit")
    if old_title is None:
        return
    title = Console.get_string("Title", "title", old_title)
    if not title:
        return
    director, year, duration = db[old_title]
    ...
    db[title] = (director, year, duration)
    if title != old_title:
        del db[old_title]
    db.sync()
```

Чтобы отредактировать сведения о диске, пользователь должен сначала выбрать диск, с которым он будет работать. Для этой операции требуется получить только название, так как названия служат ключами, а значения хранят остальные данные. Необходимая для этого функциональность будет востребована и в других местах (например, при удалении DVD), поэтому мы вынесли ее в отдельную функцию `find_dvd()`, которую мы рассмотрим следующей. Если диск найден, мы получаем от пользователя изменения, используя существующие значения как значения по умолчанию, чтобы повысить скорость взаимодействия. (Мы опустили большую часть программного кода, выполняющего взаимодействие с пользователем, так как в большинстве своем он остался тем же, что используется в функции добавления нового диска.) В конце мы сохраняем данные точно так же, как и в функции добавления. Если название не изменялось, эта операция будет иметь эффект перезаписи значения, ассоциированного с ключом, но если название изменилось, будет создана новая пара ключ-значение, и в этом случае необходимо удалить оригинальный элемент.

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    while True:
        matches = []
        start = Console.get_string(message, "title")
        if not start:
            return None
        for title in db:
            if title.lower().startswith(start.lower()):
```

```
                matches.append(title)
        if len(matches) == 0:
            print("There are no dvds starting with", start)
            continue
        elif len(matches) == 1:
            return matches[0]
        elif len(matches) > DISPLAY_LIMIT:
            print("Too many dvds start with {0}; try entering "
                  "more of the title".format(len(matches)))
            continue
        else:
            for i, match in enumerate(sorted(matches, key=str.lower)):
                print("{0}: {1}".format(i + 1, match))
            which = Console.get_integer("Number (or 0 to cancel)",
                                        "number", minimum=1, maximum=len(matches))
            return matches[which - 1] if which != 0 else None
```

Чтобы упростить и максимально ускорить поиск названия, пользователю предлагается ввести один или несколько начальных символов названия. Получив начало названия, функция выполняет итерации по данным в DBM и создает список найденных совпадений. Если имеется всего одно совпадение, оно возвращается, а если имеется несколько совпадений (но не более чем целочисленное значение `DISPLAY_LIMIT`, которое устанавливается где-то в другом месте программы), то они выводятся в алфавитном порядке без учета регистра символов, с порядковыми номерами перед ними, чтобы пользователь мог сделать выбор простым вводом числа. (Функция `Console.get_integer()` принимает 0, даже если значение аргумента `minimum` больше нуля, благодаря чему значение 0 может использоваться как признак отмены операции. Эту особенность поведения можно отключить, для чего достаточно передать аргумент `allow_zero=False`. Мы не можем использовать для отмены простое нажатие клавиши Enter, потому что ввод пустой строки рассматривается как ввод значения по умолчанию.)

```
def list_dvds(db):
    start = ""
    if len(db) > DISPLAY_LIMIT:
        start = Console.get_string("List those starting with "
                                   "[Enter=all]", "start")
    print()
    for title in sorted(db, key=str.lower):
        if not start or title.lower().startswith(start.lower()):
            director, year, duration = db[title]
            print("{0} ({1}) {2} minute{3}, by {4}".format(
                  title, year, duration, Util.s(duration), director))
```

Вывод списка всех дисков (или только тех, названия которых начинаются с определенной подстроки) реализуется простым обходом всех элементов в базе данных.

Функция `Util.s()` определена как `s = lambda x: "" if x == 1 else "s"`; здесь она возвращает символ «s», если продолжительность фильма превышает одну минуту.

```
def remove_dvd(db):
    title = find_dvd(db, "remove")
    if title is None:
        return
    ans = Console.get_bool("Remove {0}?".format(title), "no")
    if ans:
        del db[title]
        db.sync()
```

Удаление диска заключается в том, чтобы отыскать диск, который пользователь желает удалить, запросить подтверждение и в случае его получения выполнить удаление элемента из DBM.

Теперь мы знаем, как открыть (или создать) файл DBM с помощью модуля `shelve`, как добавлять в него элементы, редактировать элементы, выполнять итерации по элементам и удалять элементы.

К сожалению, в нашей базе данных имеется один недостаток. Имена режиссеров могут повторяться, что легко может приводить к несоответствиям, например, имя режиссера Danny DeVito для одного фильма может быть введено, как «Danny De Vito», а для другого, как «Danny deVito». Одно из решений этой проблемы состоит в том, чтобы создавать два файла DBM. Главный – с названиями в качестве ключей и значениями (год, продолжительность, идентификатор режиссера) и файл с режиссерами, ключами в котором являются идентификаторы режиссеров (например, целые числа), а значениями – имена. Мы устраним этот недостаток в следующем разделе, где версия программы, работающей с базой данных SQL, будет использовать две таблицы: в одной будет храниться информация о дисках, а во второй – о режиссерах.

Базы данных SQL

Интерфейсы к наиболее популярным базам данных SQL доступны в виде модулей сторонних разработчиков, а по умолчанию в составе Python поставляется модуль `sqlite3` (и база данных SQLite 3), поэтому к созданию приложений баз данных можно приступать сразу же. База данных SQLite – это облегченная база данных SQL, в которой отсутствуют многие особенности, которые имеются, например, в PostgreSQL, но ее очень удобно использовать для создания прототипов, и во многих случаях предоставляемых ею возможностей оказывается вполне достаточно.

С целью упростить миграцию с одной базы данных на другую, в PEP 249 (Python Database API Specification v2.0) дается спецификация API, которая называется DB-API 2.0, которой должны следовать интерфейсы к базам данных; модуль `sqlite3`, к примеру, следует этой специфи-

кации, но не все модули сторонних разработчиков соблюдают ее. Спецификацией API определяются два основных объекта – объект соединения и объект курсора, а API, который они должны поддерживать, приводится в табл. 11.1 и в табл. 11.2. В случае с модулем `sqlite3` его объекты соединения и курсора предоставляют множество дополнительных атрибутов и методов сверх требований, предъявляемых спецификацией DB-API 2.0.

Версия программы, использующая базу данных SQL, находится в файле *dvds-sql.py*. Программа предусматривает хранение имен режиссеров отдельно от информации о дисках, чтобы избежать повторений, и предлагает дополнительный пункт меню, дающий пользователю получить список режиссеров. Структура двух таблиц показана на рис. 11.1. Размер этой программы составляет чуть меньше 300 строк, тогда как размер программы *dvds-dbm.py* из предыдущего раздела составляет чуть меньше 200 строк. Такое различие в основном обусловлено необходимостью использовать запросы SQL вместо простых операций со словарем, а также необходимостью создавать таблицы в базе данных при первом запуске программы.

Таблица 11.1. Методы объекта соединения в соответствии со спецификацией DB-API 2.0

Синтаксис	Описание
`db.close()`	Закрывает соединение с базой данных (представленной объектом `db`, который возвращается вызовом функции `connect()`)
`db.commit()`	Подтверждает любую, ожидающую подтверждения, транзакцию в базе данных и ничего не делает, если база данных не поддерживает транзакции
`db.cursor()`	Возвращает объект курсора базы данных, посредством которого могут выполняться запросы
`db.rollback()`	Откатывает любую, ожидающую подтверждения, транзакцию в базе данных до состояния, в котором база данных находилась на момент начала транзакции, и ничего не делает, если база данных не поддерживает транзакции

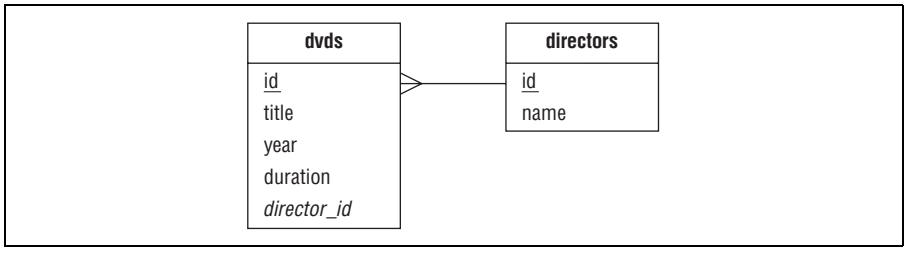

***Рис. 11.1.** Структура базы данных дисков DVD*

Таблица 11.2. Методы и атрибуты объекта курсора в соответствии со спецификацией DB-API 2.0

Синтаксис	Описание
`c.arraysize`	Число строк (доступно для чтения/записи), которое будет возвращено методом `fetchmany()`, если аргумент `size` не определен
`c.close()`	Закрывает курсор `c` — эта операция выполняется автоматически, когда поток управления покидает область видимости курсора
`c.description`	Последовательность (только для чтения), представленная кортежем из 7 элементов (`name, type_code, display_size, internal_size, precision, scale, null_ok`), описывающая очередной столбец курсора `c`
`c.execute(sql, params)`	Выполняет запрос SQL, находящийся в строке `sql`, замещая каждый символ-заполнитель соответствующим параметром из последовательности или отображения *params*, если имеется
`c.executemany (sql, seq_of_params)`	Выполняет запрос SQL по одному разу для каждого элемента в последовательности последовательностей или отображений *seq_of_params*; этот метод не должен использоваться для выполнения операций, создающих наборы результатов (таких как инструкции `SELECT`)
`c.fetchall()`	Возвращает последовательность всех строк, которые еще не были извлечены (это могут быть все строки)
`c.fetchmany(size)`	Возвращает последовательность строк (каждая строка сама по себе является последовательностью); по умолчанию аргумент *size* принимает значение `c.arraysize`
`c.fetchone()`	Возвращает следующую строку из набора результатов, полученных в результате запроса, или `None`, если все результаты были исчерпаны. Возбуждает исключение, если набор результатов отсутствует
`c.rowcount`	Доступный только для чтения счетчик строк для последней операции (такой как `SELECT`, `INSERT`, `UPDATE` или `DELETE`) или -1, если счетчик недоступен или не имеет смысла

Функция `main()` напоминает одноименную функцию из предыдущей версии программы, только на этот раз она вызывает нашу функцию `connect()`, чтобы установить соединение с базой данных.

```
def connect(filename):
    create = not os.path.exists(filename)
    db = sqlite3.connect(filename)
    if create:
        cursor = db.cursor()
        cursor.execute("CREATE TABLE directors ("
            "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
```

```
                "name TEXT UNIQUE NOT NULL)")
            cursor.execute("CREATE TABLE dvds ("
                "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
                "title TEXT NOT NULL, "
                "year INTEGER NOT NULL, "
                "duration INTEGER NOT NULL, "
                "director_id INTEGER NOT NULL, "
                "FOREIGN KEY (director_id) REFERENCES directors)")
            db.commit()
        return db
```

Функция `sqlite3.connect()` возвращает объект базы данных, открывает указанный файл базы данных или создает пустой файл базы данных, если файл не существует. Вследствие этого перед вызовом функции `sqlite3.connect()` необходимо проверить существование файла, чтобы знать, будет ли создаваться новая база данных, потому что в этом случае необходимо создать таблицы, которые используются программой. Все запросы к базе данных выполняются с помощью объекта курсора, который можно получить вызовом метода `cursor()` объекта базы данных.

Обратите внимание, что в обеих таблицах присутствует поле ID с ограничением `AUTOINCREMENT` – это означает, что SQLite автоматически будет записывать в поля ID уникальные числа, поэтому при добавлении новых записей можно переложить заботу о заполнении этих полей на SQLite.

База данных SQLite поддерживает ограниченный диапазон типов данных – по сути, только логические значения, числа и строки, но этот диапазон может быть расширен с помощью «адаптеров», либо предопределенных – для таких типов данных, как даты и время, либо создаваемых самостоятельно, которые могут использоваться для представления любых желаемых типов данных. В нашей программе этого не требуется, но в случае необходимости вы можете обратиться к описанию модуля `sqlite3`, где описываются все подробности. Синтаксис определения внешнего ключа, который мы использовали, возможно, не совпадает с синтаксисом определения внешнего ключа в других базах данных, но в любом случае – это просто описание наших намерений, поскольку база данных SQLite, в отличие от многих других, соблюдает ссылочную целостность. Еще одна особенность `sqlite3`, которая действует по умолчанию, заключается в неявной поддержке транзакций, поэтому в модуле отсутствует явный метод «запуска транзакции».

```
def add_dvd(db):
    title = Console.get_string("Title", "title")
    if not title:
        return
    director = Console.get_string("Director", "director")
    if not director:
        return
```

```
    year = Console.get_integer("Year", "year", minimum=1896,
                               maximum=datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor = db.cursor()
    cursor.execute("INSERT INTO dvds "
                   "(title, year, duration, director_id) "
                   "VALUES (?, ?, ?, ?)",
                   (title, year, duration, director_id))
    db.commit()
```

Эта функция начинается так же, как эквивалентная ей функция из программы *dvds-dbm.py*, отличия начинаются после сбора всей необходимой информации. Имя режиссера, введенное пользователем, может присутствовать, а может отсутствовать в таблице `directors`, поэтому мы предусмотрели функцию `get_and_set_director()`, которая добавляет имя режиссера, если оно еще отсутствует в базе данных, и в любом случае возвращает его идентификатор для вставки в таблицу `dvds`. Собрав все данные, функция выполняет инструкцию `INSERT` языка SQL. Здесь не требуется указывать идентификатор записи, потому что база данных SQLite подставит его автоматически.

Знаки вопроса в тексте запроса используются в качестве символов-заполнителей. Каждый знак `?` замещается соответствующим значением из последовательности, следующей за строкой с инструкцией SQL. Имеется также возможность использовать именованные символы-заполнители, она будет продемонстрирована в функции редактирования записи. Несмотря на то, что существует возможность отказаться от использования символов-заполнителей простым форматированием строки SQL, внедряя в нее необходимые данные, тем не менее мы рекомендуем всегда использовать символы-заполнители, а бремя корректного кодирования и экранирования служебных символов в элементах данных перекладывать на модуль базы данных. Еще одно преимущество использования символов-заполнителей состоит в том, что они повышают уровень безопасности, предотвращая возможность инъекции в запрос злонамеренного кода SQL.

```
def get_and_set_director(db, director):
    director_id = get_director_id(db, director)
    if director_id is not None:
        return director_id
    cursor = db.cursor()
    cursor.execute("INSERT INTO directors (name) VALUES (?)",
                   (director,))
    db.commit()
    return get_director_id(db, director)
```

Эта функция возвращает идентификатор указанного имени режиссера и в случае необходимости добавляет новую запись в таблицу `directors`.

Если была добавлена новая запись, идентификатор режиссера извлекается с помощью повторного вызова функции `get_director_id()`.

```
def get_director_id(db, director):
    cursor = db.cursor()
    cursor.execute("SELECT id FROM directors WHERE name=?",
                   (director,))
    fields = cursor.fetchone()
    return fields[0] if fields is not None else None
```

Функция `get_director_dvd()` возвращает идентификатор для заданного имени режиссера или `None`, если такого имени в базе данных не существует. Здесь используется метод `fetchone()`, потому что для данного имени может существовать либо одна запись, либо ни одной. (Дубликатов записей не может существовать, потому что поле `name` в таблице `directors` имеет ограничение `UNIQUE`, и в любом случае, прежде чем добавить новое имя режиссера в таблицу, мы проверяем его существование.) Методы извлечения записей всегда возвращают последовательность полей (или `None`, если все записи были извлечены), даже если, как в данном случае, выполняется попытка извлечь единственное поле.

```
def edit_dvd(db):
    title, identity = find_dvd(db, "edit")
    if title is None:
        return
    title = Console.get_string("Title", "title", title)
    if not title:
        return
    cursor = db.cursor()
    cursor.execute("SELECT dvds.year, dvds.duration, directors.name "
                   "FROM dvds, directors "
                   "WHERE dvds.director_id = directors.id AND "
                   "dvds.id=:id", dict(id=identity))
    year, duration, director = cursor.fetchone()
    director = Console.get_string("Director", "director", director)
    if not director:
        return
    year = Console.get_integer("Year", "year", year, 1896,
                               datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   duration, minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor.execute("UPDATE dvds SET title=:title, year=:year, "
                   "duration=:duration, director_id=:director_id "
                   "WHERE id=:id", locals())
    db.commit()
```

Чтобы отредактировать запись с информацией о диске, ее сначала необходимо отыскать. Если запись будет найдена, пользователю предоставляется возможность изменить название фильма. Затем извлекаются значения остальных полей, которые будут использоваться в качест-

ве значений по умолчанию, чтобы минимизировать ввод с клавиатуры, так как пользователю достаточно будет просто нажать клавишу Enter, чтобы принять значение по умолчанию. В этой функции используются именованные символы-заполнители (в форме *:name*), и поэтому значения для них должны поставляться в виде отображения. Для инструкции SELECT был использован вновь созданный словарь, а для инструкции UPDATE — словарь, полученный вызовом функции locals(). В обоих случаях можно было бы создавать новый словарь, и тогда для инструкции UPDATE вместо вызова функции locals() можно было бы указать словарь dict(title=title, year=year, duration=duration, director_id=director_id, id=identity).

Как только у нас будут значения всех полей и пользователь внесет все необходимые изменения, из базы данных извлекается идентификатор режиссера (при этом в случае необходимости вставляется новая запись) и затем выполняется запись обновленных данных в базу. Мы предприняли упрощенный подход, выполняя обновление сразу всех полей записи, вместо того чтобы выявлять и обновлять только те, которые действительно изменились.

При использовании базы данных DBM название диска использовалось в качестве ключа, поэтому при изменении названия создавался новый элемент ключ-значение, а прежний элемент удалялся. В данном же случае запись имеет уникальный идентификатор (поле id), который определяется в момент ее добавления, поэтому можно без всяких ограничений изменять значение любого другого поля без необходимости выполнять какие-либо дополнительные действия.

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    cursor = db.cursor()
    while True:
        start = Console.get_string(message, "title")
        if not start:
            return (None, None)
        cursor.execute("SELECT title, id FROM dvds "
                       "WHERE title LIKE ? ORDER BY title",
                       (start + "%"))
        records = cursor.fetchall()
        if len(records) == 0:
            print("There are no dvds starting with", start)
            continue
        elif len(records) == 1:
            return records[0]
        elif len(records) > DISPLAY_LIMIT:
            print("Too many dvds ({0}) start with {1}; try entering "
                  "more of the title".format(len(records), start))
            continue
        else:
            for i, record in enumerate(records):
```

```
            print("{0}: {1}".format(i + 1, record[0]))
        which = Console.get_integer("Number (or 0 to cancel)",
                                    "number", minimum=1, maximum=len(records))
        return records[which - 1] if which != 0 else (None, None)
```

Эта функция играет ту же роль, что и функция `find_dvd()` из программы *dvds-dbm.py*, и возвращает кортеж из двух элементов (название, идентификатор диска) или (`None`, `None`) в зависимости от того, была ли найдена требуемая запись. Вместо того чтобы выполнять итерации по всем данным, здесь используется оператор шаблонного символа SQL (`%`), поэтому из базы данных извлекаются только необходимые записи. А поскольку ожидается, что число записей будет невелико, выполняется извлечение сразу всех записей в последовательность последовательностей. Если будет найдено более одной записи, соответствующей условию поиска, или не настолько много, чтобы все они одновременно не поместились на экране, функция выводит их, предваряя каждую запись порядковым номером, чтобы пользователь смог сделать выбор вводом числа, как это делалось в программе *dvds-dbm.py*.

```
def list_dvds(db):
    cursor = db.cursor()
    sql = ("SELECT dvds.title, dvds.year, dvds.duration, "
           "directors.name FROM dvds, directors "
           "WHERE dvds.director_id = directors.id")
    start = None
    if dvd_count(db) > DISPLAY_LIMIT:
        start = Console.get_string("List those starting with "
                                   "[Enter=all]", "start")
        sql += " AND dvds.title LIKE ?"
    sql += " ORDER BY dvds.title"
    print()
    if start is None:
        cursor.execute(sql)
    else:
        cursor.execute(sql, (start + "%",))
    for record in cursor:
        print("{0[0]} ({0[1]}) {0[2]} minutes, by {0[3]}".format(
              record))
```

Чтобы получить сведения о каждом диске, создается запрос `SELECT`, выполняющий объединение двух таблиц. Если число записей в базе данных (возвращается функцией `dvd_count()`) оказывается больше, чем может поместиться на экране, в предложение `WHERE` добавляется второй элемент, вводящий дополнительное ограничение. Затем запрос выполняется и производится обход всех записей в результирующем наборе данных. Каждая запись представляет собой последовательность, в которой порядок следования полей определяется запросом `SELECT`.

```
def dvd_count(db):
    cursor = db.cursor()
```

```
    cursor.execute("SELECT COUNT(*) FROM dvds")
    return cursor.fetchone()[0]
```

Эти строки программного кода были оформлены в виде отдельной функции, потому что они требуются в нескольких функциях.

Мы опустили программный код функции `list_directors()`, так как по своей структуре она очень похожа на функцию `list_dvds()`, только гораздо проще, потому что она выводит список записей, состоящих из единственного поля (`name`).

```
def remove_dvd(db):
    title, identity = find_dvd(db, "remove")
    if title is None:
        return
    ans = Console.get_bool("Remove {0}?".format(title), "no")
    if ans:
        cursor = db.cursor()
        cursor.execute("DELETE FROM dvds WHERE id=?", (identity,))
        db.commit()
```

Эта функция вызывается, когда пользователь выбирает операцию удаления записи, и она очень похожа на эквивалентную функцию в программе *dvds-dbm.py*.

На этом мы заканчиваем обзор программы *dvds-sql.py*, и теперь мы знаем, как создавать таблицы в базе данных, как выбирать записи, как выполнять итерации по выбранным записям, а также как вставлять, изменять и удалять записи. С помощью метода `execute()` можно выполнять любые инструкции SQL, какие только поддерживаются используемой базой данных.

База данных SQLite обладает гораздо более широкими возможностями, чем было использовано здесь, включая режим автоматического подтверждения транзакций (и другие операции управления транзакциями), и возможность создавать функции, которые могут выполняться внутри запросов SQL. Имеется также возможность создавать фабричные функции, управляющие представлением возвращаемых записей (например, в виде словарей или собственных типов данных, вместо последовательностей полей). Дополнительно имеется возможность создавать базы данных SQLite в памяти, для чего достаточно передать строку `":memory"` в качестве имени файла.

В заключение

В главе 7 были продемонстрированы различные способы сохранения на диск и загрузки данных с диска, и в этой главе было показано, как можно взаимодействовать с типами данных, сохраняющих информацию на диске, а не в памяти.

В случае использования файлов DBM очень удобным в использовании оказывается модуль `shelve`, так как он позволяет хранить элементы данных в виде строка-объект. В случае необходимости иметь более полный контроль имеется возможность прямого взаимодействия с используемыми базами данных DBM. Одна замечательная особенность баз данных DBM вообще и модуля `shelve` в частности состоит в том, что они используют API словаря, обеспечивая простоту получения, добавления, редактирования и удаления элементов, и позволяют легко перевести программу, применяющую словари, на использование DBM. Одно маленькое неудобство применения DBM заключается в том, что в случае реляционных данных каждая таблица элементов ключ-значение должна храниться в отдельном файле DBM, тогда как база данных SQLite хранит все данные в одном файле.

База данных SQLite прекрасно подходит для разработки прототипов программ, которые будут работать с базами данных SQL, и во многих случаях она может использоваться как основная база данных, особенно благодаря тому, что она включается в состав дистрибутива Python. В этой главе было продемонстрировано, как получать объект базы данных с помощью функции `connect()` и как выполнять запросы SQL (такие как `CREATE TABLE`, `SELECT`, `INSERT`, `UPDATE` и `DELETE`) с помощью метода `execute()` объекта курсора.

Язык Python предлагает широкий диапазон средств хранения данных на диске или в памяти, начиная от двоичных файлов, файлов XML и законсервированных объектов и заканчивая базами данных DBM и SQL, что позволяет выбирать нужное средство в зависимости от ситуации.

Упражнение

Напишите интерактивную консольную программу обслуживания списка закладок. Для каждой закладки должны храниться два элемента данных: адрес URL и имя. Ниже приводится пример сеанса работы с программой:

```
Bookmarks (bookmarks.dbm)
(1) Programming in Python 3..... http://www.qtrac.eu/py3book.html
(2) PyQt........................ http://www.riverbankcomputing.com
(3) Python...................... http://www.python.org
(4) Qtrac Ltd................... http://www.qtrac.eu
(5) Scientific Tools for Python.... http://www.scipy.org

(A)dd (E)dit (L)ist (R)emove (Q)uit [l]: e
Number of bookmark to edit: 2
URL [http://www.riverbankcomputing.com]:
Name [PyQt]: PyQt (Python bindings for GUI library)
```

Программа должна давать пользователю возможность добавлять, редактировать, выводить список и удалять закладки. Чтобы обеспечить максимальную простоту выбора закладки при выполнении операций

редактирования и удаления, закладки должны выводиться в виде списка с порядковыми номерами и пользователю должна предлагаться возможность ввести номер закладки для редактирования или удаления. Данные должны сохраняться в файле DBM с применением модуля `shelve`, где имена должны играть роль ключей, а адреса URL – значений. По своей структуре программа очень похожа на программу *dvds-dbm.py*, за исключением функции `find_bookmark()`, которая получится намного проще, чем функция `find_dvd()`, так как она будет получать от пользователя только порядковый номер закладки и выполнять поиск имени по этому номеру.

В качестве дополнительного удобства, если пользователь явно не указывает протокол, добавляемые или редактируемые адреса URL следует предварять строкой `http://`.

Вся программа может уместиться менее чем в 100 строк (предполагается, что будут использоваться функции `Console.get_string()` и подобные ей из модуля `Console`). Решение приводится в файле *bookmarks.py*.

12

- Язык регулярных выражений в Python
- Модуль для работы с регулярными выражениями

Регулярные выражения

Регулярное выражение – это компактная форма записи представления о коллекции строк. Огромная мощь регулярных выражений обусловлена тем, что единственное регулярное выражение может представлять неограниченное число строк, отвечающих требованиям регулярного выражения. Регулярные выражения определяются с помощью мини-языка, который совершенно не похож на язык Python, но в состав Python входит модуль `re`, с помощью которого можно создавать и использовать регулярные выражения.[1]

Регулярные выражения используются для достижения следующих четырех основных целей:

- Проверка: проверка соответствия фрагментов текста некоторым критериям, например, наличие символа обозначения валюты и последующих за ним цифр.
- Поиск: поиск подстрок, которые могут иметь несколько форм, например, поиск подстрок «pet.png», «pet.jpg», «pet.jpeg» или «pet.svg», чтобы при этом не обнаруживались подстроки «carpet.png» и подобные ей.
- Поиск и замена: замена всего, что совпадает с регулярным выражением, и замена на указанную строку, например, поиск подстроки

[1] Существует превосходная книга о регулярных выражениях: Jeffrey E. F. Friedl «Mastering Regular Expressions» (Джеффри Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008). Она не имеет явного отношения к языку Python, но модуль `re`, входящий в состав Python, предлагает функциональность, во многом похожую на ту, что предлагается механизмом регулярных выражений языка Perl, который подробно рассматривается в этой книге.

«устройство передвижения, движимое мускульной силой» и замена подстрокой «велосипед».

- Разбиение строк: разбиение строки по точкам совпадения с регулярным выражением, например, разбиение строки по подстроке «: » или «=».

Самым простым выражением является выражение (например, обычный символ), за которым может следовать квантификатор. Более сложные регулярные выражения могут состоять из любого числа выражений с квантификаторами и могут включать проверки и дополнительные флаги.

В первом разделе этой главы будут представлены и описаны все ключевые концепции, имеющие отношение к регулярным выражениям, и будет продемонстрирован чистый синтаксис регулярных выражений с минимальными ссылками на язык Python. Затем, во втором разделе, будет показано, как использовать регулярные выражения в контексте программирования на языке Python, с привлечением сведений из предыдущих разделов. Читатели, знакомые с регулярными выражениями и желающие узнать, как они работают в языке Python, могут сразу перейти ко второму разделу (на стр. 538). Глава полностью охватывает язык регулярных выражений, предлагаемый модулем `re`, включая все проверки и флаги. Мы будем выделять регулярные выражения в тексте **`жирным шрифтом`**, места совпадений – `шрифтом с подчеркиванием`, а захваченные символы – `с подчеркиванием и затенением`.

Язык регулярных выражений в Python

В этом разделе мы разобьем рассмотрение языка регулярных выражений на четыре подраздела. В первом подразделе демонстрируется, как находить совпадения с отдельными символами или с группами символов, например, совпадение с символом a, или совпадение с символом b, или совпадение с символом a или b. Во втором подразделе демонстрируется, как находить совпадения с применением квантификаторов, например, единственное совпадение, или не менее одного совпадения, или столько совпадений, сколько вообще возможно. В третьем подразделе демонстрируется, как группировать подвыражения и как сохранять текст совпадений. И, наконец, в последнем подразделе демонстрируется, как воздействовать на работу регулярных выражений с использованием проверок и флагов.

Символы и классы символов

Простейшее регулярное выражение – это обычные литералы символов, такие как `a` или `5`. В отсутствие явного квантификатора такое выражение подразумевает «совпадение с одним вхождением». Например, регулярное выражение `tune` состоит из четырех выражений, каждое из них неявно определяет одно совпадение, поэтому оно будет сов-

падать с одним символом *t*, за которым следует один символ *u*, за которым следует один символ *n*, за которым следует один символ *e*, то есть оно будет совпадать со строками tune и attuned.

Экранированные последовательности, стр. 86

Большинство символов могут использоваться как литералы, но некоторые из них имеют «специальное назначение» в языке регулярных выражений и потому должны экранироваться символом обратного слеша (\), когда они используются как литералы. К специальным относятся символы \.^$?+*{}[]()|. В пределах регулярных выражений можно также использовать большинство стандартных экранированных последовательностей языка Python, например, \n – для обозначения символа перевода строки, \t – для обозначения символа табуляции, а также экранированные последовательности с шестнадцатеричными кодами символов *\xHH*, *\uHHHH* и *\UHHHHHHHH*.

Во многих случаях вместо совпадения с единственным символом бывает необходимо отыскать совпадение с одним из множества символов. Реализовать это можно с помощью *символьного класса* – один или более символов, заключенные в квадратные скобки. (Символьные классы не имеют никакого отношения к классам в языке Python – это просто термин регулярных выражений, используемый для обозначения «множества символов».) Символьный класс – это выражение и, как и любое другое выражение, в отсутствие явного квантификатора соответствует точно одному символу (который может быть любым символом из данного символьного класса). Например, регулярное выражение r[ea]d совпадает с red и radar, но не со словом read. Точно так же, чтобы отыскать совпадение с единственной цифрой, можно использовать регулярное выражение [0123456789]. Для удобства можно указывать диапазон символов с помощью символа дефиса; так, выражение [0-9] также будет соответствовать цифре. Имеется возможность инвертировать значение символьного класса, указывая символ «крышки» после открывающей квадратной скобки; так, выражение [^0-9] будет соответствовать любому символу, но *не* цифре.

Обратите внимание, что в символьном классе все специальные символы, кроме символа \, теряют свое специальное значение, но, что касается символа ^, то он приобретает новое значение (инверсия), когда является первым символом в символьном классе; в противном случае он просто обозначает символ «крышки». Кроме того, символ дефиса обозначает диапазон символов, только если он не является первым символом – в этом случае он просто обозначает символ дефиса.

Так как некоторые наборы символов требуются достаточно часто, для них предусматриваются краткие формы записи – они перечислены в табл. 12.1. За одним исключением, эти сокращенные формы могут использоваться внутри символьных классов, например, регулярное выражение [\dA-Fa-f] соответствует шестнадцатеричной цифре. Ис-

ключение составляет символ точки, который за пределами символьного класса обозначает набор символов, а внутри символьного класса – сам символ точки.

Таблица 12.1. Сокращенные формы символьных классов

Символ	Значение
.	Соответствует любому символу, за исключением символа перевода строки, или любому символу, если используется флаг `re.DOTALL`, или символу `.` внутри символьного класса
`\d`	Соответствует цифре Юникода или `[0-9]`, если используется флаг `re.ASCII`
`\D`	Соответствует нецифровому символу Юникода или `[^0-9]`, если используется флаг `re.ASCII`
`\s`	Соответствует любому пробельному символу Юникода или `[ \t\n\r\f\v]`, если используется флаг `re.ASCII`
`\S`	Соответствует любому символу Юникода, не являющемуся пробельным, или `[^ \t\n\r\f\v]`, если используется флаг `re.ASCII`
`\w`	Соответствует символу «слова» Юникода или `[a-zA-Z0-9_]`, если используется флаг `re.ASCII`
`\W`	Соответствует любому символу не-«слова» Юникода или `[^a-zA-Z0-9_]`, если используется флаг `re.ASCII`

Флаги, стр. 533

Квантификаторы

Квантификаторы записываются в виде {*m*,*n*}, где *m* и *n* – это минимальное и максимальное число совпадений с выражением, при которых соответствие выражению с квантификатором будет считаться найденным. Например, оба выражения `e{1,1}e{1,1}` и `e{2,2}` соответствуют слову `feel`, но не соответствуют слову `felt`.

Записывать квантификатор после каждого выражения было бы слишком утомительно, а сами регулярные выражения при этом было бы трудно читать. К счастью, язык регулярных выражений поддерживает несколько удобных сокращений. Если в квантификаторе указывается только одно число, оно обозначает и минимум и максимум, то есть выражение `e{2}` – это то же самое, что выражение `e{2,2}`. И, как уже отмечалось в предыдущем разделе, если квантификатор не указан явно, предполагается, что он равен единице (например, `{1,1}`, или `{1}`), то есть выражение `ee` – это то же самое, что выражение `e{1,1}e{1,1}` или `e{1}e{1}`; оба выражения `e{2}` и `ee` соответствуют слову `feel`, но не соответствуют слову `felt`.

Бывает удобно использовать различные минимальное и максимальное значения. Например, чтобы найти совпадение со словами `travelled` и `traveled` (обе формы записи являются допустимыми) можно было бы

использовать выражение `travel{1,2}ed` или `travell{0,1}ed`. Квантификатор `{0,1}` используется настолько часто, что для него появилось собственное сокращение – `?`; поэтому другой способ записи (к тому же чаще использующийся на практике) этого регулярного выражения выглядит так: `travell?ed`.

Имеются еще два сокращения для квантификаторов: `+` – обозначает `{1,`*n*`}` («не менее одного»); `*` – обозначает `{0,`*n*`}` («любое число»). В обоих случаях *n* обозначает максимально допустимое для квантификатора число, которое обычно не менее 32 767. Все квантификаторы перечислены в табл. 12.2.

Таблица 12.2. Квантификаторы регулярных выражений

Синтаксис	Значение
e? или *e*{0,1}	Максимальный, соответствует нулевому или большему числу вхождений выражения *e*
e?? или *e*{0,1}?	Минимальный, соответствует нулевому или большему числу вхождений выражения *e*
e+ или *e*{1,}	Максимальный, соответствует одному или больше вхождению выражения *e*
e+? или *e*{1,}?	Минимальный, соответствует одному или больше вхождению выражения *e*
*e** или *e*{0,}	Максимальный, соответствует нулевому или большему числу вхождений выражения *e*
*e**? или *e*{0,}?	Минимальный, соответствует нулевому или большему числу вхождений выражения *e*
e{*m*}	Соответствует точно *m* вхождениям выражения *e*
e{*m*,}	Максимальный, соответствует по меньшей мере *m* вхождениям выражения *e*
e{*m*,}?	Минимальный, соответствует по меньшей мере *m* вхождениям выражения *e*
e{,*n*}	Максимальный, соответствует не более чем *n* вхождениям выражения *e*
e{,*n*}?	Минимальный, соответствует не более чем *n* вхождениям выражения *e*
e{*m*,*n*}	Максимальный, соответствует не менее чем *m* и не более чем *n* вхождениям выражения *e*
e{*m*,*n*}?	Минимальный, соответствует не менее чем *m* и не более чем *n* вхождениям выражения *e*

Квантификатор `+` очень удобен. Например, для поиска соответствий целым числам можно было бы использовать выражение `\d+`, так как оно совпадает с одной или более цифрами. Данное регулярное выражение может отыскать два совпадения в строке `4588.91`, например,

4588.91 и 4588.91. Иногда длительное удержание клавиши приводит к появлению опечаток. Мы могли бы использовать выражение `bevel+ed` для поиска допустимых форм написания beveled и bevelled, а также опечатки bevellled. Если бы потребовалось считать допустимой только форму записи с одним символом *l* и отыскать только формы записи с двумя или более символами *l*, мы могли бы применить регулярное выражение `bevell+ed`.

Квантификатор `*` используется реже – просто потому, что он может приводить к получению неожиданных результатов. Например, предположим, что требуется отыскать строки с комментариями в файлах с программным кодом на языке Python. Для этого мы могли бы попытаться использовать регулярное выражение `#*`. Но такое выражение будет соответствовать любым строкам, даже пустым, потому что фраза «совпадение с любым числом символов #» подразумевает и нулевое число совпадений. Если вы плохо знакомы с регулярными выражениями, возьмите себе за правило вообще не использовать квантификатор `*`, а если вы используете его (или квантификатор `?`), то обязательно убедитесь, что хотя бы одно подвыражение в регулярном выражении использует ненулевой квантификатор, то есть используется хотя бы один квантификатор, отличный от `*` и `?`, так как оба они могут находить соответствие при нулевом числе совпадений.

Часто возможно заменить квантификатор `*` на квантификатор `+`, и наоборот. Например, отыскать соответствие слову «tasselled», в котором присутствует хотя бы один символ *l*, можно с помощью выражения `tassel*ed` или `tassel+ed`, а соответствие формам записи с двумя или более символами *l* – с помощью `tasselll*ed` или `tassell+ed`.

Регулярное выражение `\d+` будет соответствовать строке 136. Но почему оно соответствует всем цифрам, а не только первой? По умолчанию все квантификаторы являются *жадными* (или *максимальными*) – они стремятся соответствовать как можно большему числу символов. Любой квантификатор можно сделать нежадным (или *минимальным*), добавив после него символ `?`. (Знак вопроса имеет два разных значения – когда он употребляется самостоятельно, он интерпретируется как сокращенная форма записи квантификатора `{0,1}`, а когда следует за квантификатором – он говорит о том, что стоящий перед ним квантификатор является минимальным.) Например, выражение `\d+?` обнаружит соответствие в строке 136 в трех разных местах: 136, 136 и 136. Другое выражение: `\d??` соответствует нулевому или большему числу цифр, но предпочтение будет отдано нулевому числу совпадений, потому что используется минимальный квантификатор – он порождает ту же проблему, что и квантификатор `*`, и может находить соответствие «с ничем», то есть будет соответствовать вообще любому тексту.

Минимальные квантификаторы могут с успехом использоваться для быстрого и приблизительного синтаксического анализа документов в формате XML и HTML. Например, для поиска всех тегов изображе-

ний можно было бы использовать выражение <img.*> (соответствует одному символу «<», за которым следует символ «i», затем символ «m», затем символ «g», затем ноль или более любых символов, за исключением символа перевода строки, и затем символ «>»), но оно будет давать неверные результаты, потому часть .* является жадной и будет соответствовать всему подряд, включая закрывающую угловую скобку тега «>», и остановится только по достижении последнего символа «>» в тексте.

Сами собой напрашиваются три варианта решения (кроме использования нормального парсера). Одно из них – выражение <img[^>]*> (соответствует подстроке <img, за которой следует любое число любых символов, отличных от >, и затем следует закрывающая угловая скобка тега >), другое – выражение <img.*?> (соответствует подстроке <img, за которой следует любое минимальное число любых символов, поэтому поиск соответствия будет остановлен сразу же по достижении символа >, затем следует >) и третье решение является комбинацией двух предыдущих – <img[^>]*?>. Но ни одно из них не является правильным, потому что все они будут соответствовать строке <img>, которая не является допустимым тегом. Так как известно, что тег изображения должен иметь атрибут src, можно записать более точное регулярное выражение <img\s+[^>]*?src=\w+[^>]*?>. Это выражение соответствует всем литералам символов <img, за которыми следует один или более пробельных символов, затем минимально ноль или более любых символов, за исключением > (чтобы пропустить любые атрибуты, такие как alt), затем атрибут src (символы src=, затем хотя бы один символ «слова»), затем любой (включая нулевое число) символ, за исключением >, чтобы пропустить любые другие атрибуты, и, наконец, закрывающая угловая скобка >.

Группировка и сохранение

На практике часто бывают необходимы регулярные выражения, соответствующие любой из двух или более альтернатив, и нередко требуется сохранить совпадение или какую-то его часть для последующей обработки. Кроме того, иногда требуется, чтобы квантификатор применялся к нескольким выражениям. Все это может быть реализовано с помощью операции группировки (), а выбор альтернативных вариантов можно реализовать с помощью конструкции выбора |.

Конструкция выбора особенно удобна, когда необходимо отыскать совпадение с одной из нескольких альтернатив. Например, регулярное выражение aircraft|airplane|jet будет соответствовать любому тексту, содержащему «aircraft», «airplane» или «jet». Тот же результат можно получить с помощью регулярного выражения air(craft|plane)|jet. В данном случае круглые скобки используются для группировки выражений, таким образом здесь имеются два внешних выражения: air(craft|plane) и jet. Первое из них имеет внутреннее выражение

craft|plane, а так как ему предшествует выражение air, первое внешнее выражение может соответствовать только слову «aircraft» или «airplane».

Круглые скобки преследуют две разные цели – сгруппировать выражения и сохранить текст, совпавший с выражением. Мы будем использовать термин *группировка* для обозначения сгруппированного выражения как выполняющего сохранение, так и не выполняющего, и *сохранение* или *группировка с сохранением* – для обозначения сохраняющей группы. Регулярное выражение (aircraft|airplane|jet) не только будет обнаруживать соответствие любому из трех выражений, но еще будет сохранять любое совпадение для последующего использования. Сравните это с регулярным выражением (air(craft|plane)|jet), в котором будет сохранено два фрагмента, если будет обнаружено совпадение с первым выражением («aircraft» или «airplane» – первое сохранение, и «craft» или «plane» – второе сохранение), и одно сохранение, если будет обнаружено совпадение со вторым выражением («jet»). Имеется возможность отключить эффект сохранения, поместив вслед за открывающей круглой скобкой символы ?:; так, например, регулярное выражение (air(?:craft|plane)|jet) при любом совпадении будет иметь только одно сохранение («aircraft», или «airplane» или «jet»).

Сгруппированное выражение – это обычное выражение, и потому к нему могут применяться квантификаторы. Как и для любого другого выражения, отсутствие квантификатора подразумевает точно одно совпадение. Например, при чтении текстового файла со строками в формате *ключ=значение*, где *ключ* содержит только алфавитно-цифровые символы, регулярное выражение (\w+)=(.+) будет соответствовать любой строке, в которой присутствуют непустой ключ и непустое значение. (Не забывайте, что . соответствует всему, чему угодно, за исключением символов перевода строки.) И для каждой совпавшей строки будет выполнено два сохранения. Первым сохранением будет ключ, а вторым – значение.

Например, регулярному выражению поиска пар *ключ=значение* будет соответствовать вся строка topic= physical geography с двумя сохранениями, выделенными серым фоном. Обратите внимание, что второе сохранение включает в себя несколько пробельных символов и что пробельный символ перед знаком = является недопустимым. Чтобы обеспечить большую гибкость в отношении пробелов и одновременно удалить нежелательные пробелы из сохранений, мы могли бы усовершенствовать регулярное выражение, создав более длинную версию:

```
[ \t]*(\w+)[ \t]*=[ \t]*(.+)
```

Этому регулярному выражению будет соответствовать та же строка, что и прежде, а также строки, где символ = с обеих сторон окружен пробелами; но в первом сохранении будут отсутствовать начальные и завершающие пробелы, а во втором сохранении будут отсутствовать начальные пробелы. Например: topic = physical geography.

Флаги регулярных выражений, стр. 535

Мы сделали все возможное, чтобы обеспечить совпадение с пробельными символами за пределами сохраняющих круглых скобок и одновременно обеспечить соответствие строк, в которых вообще отсутствуют пробелы. Мы не использовали символ `\s` для поиска совпадений с пробельными символами, потому что ему также соответствует символ перевода строки (`\n`); в противном случае это могло бы привести к некорректному совпадению сразу с несколькими строками (например, при использовании флага `re.MULTILINE`). И для поиска значения мы не использовали символ `\S`, соответствующий непробельным символам, потому что мы допускаем наличие пробелов внутри значений (например, внутри текста предложения). Чтобы исключить завершающие пробелы из второго сохранения, нам потребовалось бы более сложное регулярное выражение – мы увидим его в следующем подразделе.

Обратиться к захваченному тексту можно с помощью *обратных ссылок*, то есть с помощью ссылок на предшествующие группы.[1] Синтаксически обратные ссылки в регулярных выражениях имеют вид `\`*`i`*, где *i* – это порядковый номер сохраняющей группы. Нумерация сохраняющих групп начинается с единицы и увеличивается на единицу с каждой новой (начинающей сохраняющую группу) открывающей круглой скобкой, слева направо. Например, в упрощенном варианте поиск повторяющихся слов можно было бы выполнять с помощью регулярного выражения `(\w+)\s+\1`, которому соответствует «слово», за которым следует по меньшей мере один пробельный символ и затем то же самое слово, что было захвачено. (Сохранение с порядковым номером 0 создается автоматически, без применения круглых скобок – оно хранит все соответствие, то есть то, что мы показываем шрифтом с подчеркиванием.) Позднее мы увидим более сложный способ поиска повторяющихся слов.

В длинных или сложных регулярных выражениях для ссылки на сохранения часто удобнее использовать имена, а не порядковые номера. Это позволяет упростить сопровождение, так как добавление или удаление сохраняющих круглых скобок может приводить к изменению номеров, но не будет оказывать влияния на имена. Чтобы присвоить имя сохраняющей группе, вслед за открывающей круглой скобкой следует указать `?P<`*`name`*`>`. Например, в регулярном выражении `(?P<key>\w+)=(?P<value>.+)` имеется две сохраняющих группы – с именами `"key"` и `"value"`. Для обращения к именованным сохраняющим группам внутри регулярных выражений используется синтаксис `(?P=`*`name`*`)`. На-

[1] Обратите внимание, что обратные ссылки не могут использоваться внутри символьных классов, то есть внутри `[]`.

пример, регулярному выражению (?P<word>\w+)\s+(?P=word), где используется сохраняющая группа с именем "word", будут соответствовать пары повторяющихся слов.

Проверки и флаги

Одна из проблем, которая свойственна многим регулярным выражениям, которые мы видели выше, заключается в том, что они могут находить больше соответствий, чем предполагается, и даже могут находить соответствия там, где они не предполагались. Например, регулярному выражению aircraft|airplane|jet будут соответствовать слова «waterjet» и «jetski», кроме «jet». Такого рода проблемы могут быть решены с помощью проверки. Проверка не соответствует никакому тексту, она просто требует выполнения некоторого условия в точке, где выполняется проверка.

Проверка \b (граница слова) требует, чтобы символ, предшествующий ей, был символом «слова» (\w), а символ, следующий за ней, не был символом «слова» (\W), и наоборот. Например, регулярное выражение jet будет находить в тексте the jet and jetski are noisy два соответствия: the jet and jetski are noisy, а регулярное выражение \bjet\b будет находить только одно соответствие: the jet and jetski are noisy. Возвращаясь к оригинальному регулярному выражению, мы могли бы записать его как \baircraft\b|\bairplane\b|\bjet\b или, более просто, – как \b(?:aircraft|airplane|jet)\b, то есть граница слова, несохраняющее выражение, граница слова.

Поддерживаются многие другие проверки, как показано в табл. 12.3. Мы могли бы использовать проверки для улучшения регулярного выражения, отыскивающего пары *ключ=значение*, например, заменив его выражением ^(\w+)=([^\n]+) и установив флаг re.MULTILINE, чтобы гарантировать, что каждая пара *ключ=значение* будет извлекаться из единственной строки и не будет происходить объединения нескольких строк. (Перечень флагов приводится в табл. 12.4 на стр. 535, их синтаксис описывается в конце этого подраздела, а примеры использования – в начале следующего раздела.) А если бы нам потребовалось удалять начальные и конечные пробельные символы и использовать именованные сохраняющие группы, то полное регулярное выражение могло бы выглядеть так:

```
^[ \t]*(?P<key>\w+)[ \t]*=[ \t]*(?P<value>[^\n]+)(?<![ \t])
```

Флаги регулярных выражений, стр. 535

Хотя это регулярное выражение решает очень простую задачу, оно выглядит достаточно сложным. Один из способов упростить его сопровождение состоит в использовании комментариев. Сделать это можно, добавляя встроенные комментарии, используя синтаксис *(?#текст комментария)*, но на практике такие комментарии легко могут еще больше осложнить понимание регулярного

выражения. Лучшее решение заключается в использовании флага `re.VERBOSE` – он позволяет использовать внутри регулярных выражений пробельные символы и обычные комментарии языка Python с одним ограничением – когда необходимо описать совпадение с пробельным символом, следует использовать либо символ `\s`, либо символьный класс, такой как `[ ]`. Ниже приводится регулярное выражение, отыскивающее пары *ключ=значение*, с комментариями:

```
^[ \t]*           # начало строки и необязательные начальные пробелы
(?P<key>\w+)      # текст ключа
[ \t]*=[ \t]*     # знак равенства с возможными пробелами, окружающими его
(?P<value>[^\n]+) # текст значения
(?<![ \t])        # негативная ретроспективная проверка для исключения
                  # завершающих пробелов
```

Таблица 12.3. Проверки регулярных выражений

Символ	Значение
`^`	Соответствует началу текста; кроме того, с флагом `re.MULTILINE` соответствует позиции сразу после каждого символа перевода строки
`$`	Соответствует концу текста; кроме того, с флагом `re.MULTILINE` соответствует позиции перед каждым символом перевода строки
`\A`	Соответствует началу текста
`\b`	Соответствует границе «слова»; поведение проверки зависит от флага `re.ASCII` – внутри символьных классов обозначает символ забоя (backspace)
`\B`	Соответствует границе не-«слова»; поведение проверки зависит от флага `re.ASCII`
`\Z`	Соответствует концу текста
`(?=`*e*`)`	Совпадение обнаруживается, если текст справа от позиции проверки соответствует выражению *e*, при этом изменение позиции поиска в тексте не происходит – эта проверка называется *опережающей проверкой*, или *позитивной опережающей проверкой*
`(?!`*e*`)`	Совпадение обнаруживается, если текст справа от позиции проверки не соответствует выражению *e*, при этом изменение позиции поиска в тексте не происходит – эта проверка называется *негативной опережающей проверкой*
`(?<=`*e*`)`	Совпадение обнаруживается, если текст слева от позиции проверки соответствует выражению *e*, эта проверка называется *позитивной ретроспективной проверкой*
`(?<!`*e*`)`	Совпадение обнаруживается, если текст слева от позиции проверки не соответствует выражению *e*, эта проверка называется *негативной ретроспективной проверкой*

Флаги регулярных выражений, стр. 535

Таблица 12.4. Флаги модуля регулярных выражений

Флаг	Значение
`re.A` или `re.ASCII`	При наличии этого флага проверки `\b`, `\B`, `\s`, `\S`, `\w` и `\W` действуют так, как если бы они применялись к тексту, содержащему только символы ASCII; по умолчанию действие этих проверок основано на спецификации Юникода
`re.I` или `re.IGNORECASE`	Поиск совпадений выполняется без учета регистра символов
`re.M` или `re.MULTILINE`	При наличии этого флага символ `^` соответствует началу текста и позиции сразу же после каждого символа перевода строки, а символ `$` соответствует позиции перед каждым символом перевода строки и концу текста
`re.S` или `re.DOTALL`	При наличии этого флага символ `.` соответствует любому символу, включая символ перевода строки
`re.X` или `re.VERBOSE`	Позволяет включать в регулярные выражения пробелы и комментарии

В контексте программирования на языке Python регулярные выражения, подобные этому, обычно записываются в виде «сырых» строк, заключенных в тройные кавычки, – «сырые» строки используются, чтобы избежать необходимости дублировать символы обратного слеша, а тройные кавычки – чтобы регулярное выражение можно было записывать в нескольких строках.

«Сырые» строки, стр. 85

В дополнение к проверкам, обсуждавшимся до сих пор, существуют дополнительные проверки, которые заглядывают по тексту вперед (или назад) от точки проверки, чтобы узнать, соответствует ли (или не соответствует) определенное нами выражение. Выражения, которые могут использоваться в ретроспективных проверках, должны иметь фиксированную длину (то есть в них не могут использоваться квантификаторы `?`, `+` и `*`, а интервальные квантификаторы должны задавать фиксированный размер, например, `{3}`).

В случае регулярного выражения, отыскивающего совпадения со строками в формате *ключ=значение*, негативная обратная проверка для совпадения требует, чтобы символы, *предшествующие* позиции проверки, не были пробелами или символами табуляции. Вследствие этого последний символ «значения» сохраняется в сохраняющей группе, а завершающие пробелы и символы табуляции – нет (это не относится к пробелам и символам табуляции внутри сохраненного текста).

Рассмотрим другой пример. Предположим, что мы читаем многострочный текст, содержащий имена «Helen Patricia Sharman», «Jim Sharman», «Sharman Joshi», «Helen Kelly» и т. д., и нам необходимо отыскать текст «Helen Patricia», но только если он относится к имени

«Helen Patricia Sharman». Простейший способ заключается в том, чтобы использовать регулярное выражение `\b(Helen\s+Patricia)\s+Sharman\b`. Однако того же самого можно добиться с помощью опережающей проверки, например, `\b(Helen\s+Patricia)(?=\s+Sharman\b)`. Этому выражению будет соответствовать текст «Helen Patricia», только если он предшествует границе слова, за которым следуют пробельные символы и слово «Sharman» и далее следует граница слова.

Чтобы сохранить определенное имя в самых разных его разновидностях («Helen», «Helen P.» или «Helen Patricia»), можно было бы создать немного более сложное регулярное выражение, например, `\b(Helen(?:\s+(?:P\.|Patricia))?)\s+(?=Sharman\b)`. Ему соответствует граница слова, за которой следует имя в одной из форм, но только если вслед за ним следует некоторое число пробельных символов, затем слово «Helen Patricia Sharman» и граница слова.

Обратите внимание, что сохранение выполняется только двумя синтаксическими конструкциями – `(`*e*`)` и `(?P<`*name*`>`*e*`)`. Других форм сохраняющих группировок в круглых скобках не существует. Это особенно важно для опережающих и ретроспективных проверок, так как они лишь проверяют последующий или предшествующий текст – сами они не являются частью совпадения, а только проверяют факт наличия или отсутствия совпадения. Это также важно для последних двух синтаксических конструкций в круглых скобках, которые мы теперь рассмотрим.

Ранее мы уже видели, что получить доступ к сохраненному тексту внутри регулярного выражения можно либо по номеру группы (например, `\1`), либо по имени (например, `(?P=`*name*`)`). Однако имеется возможность принимать решение о наличии соответствия в зависимости от наличия предыдущего совпадения. Для этого используются синтаксические конструкции `(?(`*id*`)`*yes_exp*) и `(?(`*id*`)`*yes_exp*`|`*no_exp*). Здесь *id* – это имя или номер предыдущей сохраняющей группы. Если для указанной группы совпадение было обнаружено, здесь будет выполняться проверка на совпадение с выражением *yes_exp*. Если для указанной группы совпадение не было обнаружено, то здесь будет выполняться проверка на совпадение с выражением *no_exp*, если оно определено.

Рассмотрим пример. Предположим, что необходимо извлечь имена файлов, которые упоминаются в атрибутах `src` тегов `img` в документе HTML. Для начала попробуем просто отыскать атрибут `src`, но, в отличие от предыдущих попыток, мы учтем тот факт, что значение атрибута может указываться в трех формах: в апострофах, в кавычках и без кавычек. Вот первая попытка: `src=(["'])([^"'>]+)\1`. Часть `([^"'>]+)` сохраняет найденное совпадение максимальной длины, содержащее по меньшей мере один символ, который не является кавычкой, апострофом или `>`. Это регулярное выражение прекрасно работает, когда имена файлов указываются в кавычках, а благодаря обратной ссылке `\1` соответствие будет обнаружено, только если кавычки, обрамляющие

имя файла, – одного типа. Но это выражение не позволяет отыскивать имена файлов без кавычек. Чтобы устранить эту проблему, необходимо сделать символ открывающей кавычки необязательным, а совпадение с закрывающей кавычкой выполнять, только если было найдено совпадение с открывающей кавычкой. Вот как выглядит исправленное регулярное выражение: `src=(["']?([^"'>]+)(?(1)\1)`. Здесь мы не используем выражение *no_exp*, потому что нам не требуется выявлять совпадение еще с чем-то, если открывающая кавычка отсутствует. Теперь мы готовы поместить созданное регулярное выражение в определенный контекст. Ниже приводится полное регулярное выражение, в котором используются именованные группы и комментарии:

```
<img\s+              # начало тега
[^>]*?               # любые атрибуты, предшествующие src
src=                 # начало атрибута src
(?P<quote>["'])?     # необязательная открывающая кавычка
(?P<image>[^"'>]+)   # имя файла изображения
(?(quote)(?P=quote)) # закрывающая кавычка (если была открывающая кавычка)
[^>]*?               # любые атрибуты, следующие за src
>                    # конец тега
```

Сохраняющей группе с именем файла было присвоено имя `"image"` (этой группе соответствует порядковый номер 2).

Конечно, существует более простая, но менее очевидная альтернатива: `src=(["']?)([^"'>]+)\1`. Здесь, если имеется символ открывающей кавычки, он сохраняется в группе 1, далее идут символы, следующие за открывающей кавычкой. Если в определении атрибута открывающая кавычка отсутствовала, по-прежнему будет считаться, что для группы 1 было найдено совпадение – совпадение с пустой строкой, так как кавычка считается необязательной (ее квантификатор означает ноль или более совпадений); в этом случае обратной ссылке также будет соответствовать пустая строка.

Флаги регулярных выражений, стр. 535

Наконец, механизм регулярных выражений в языке Python предоставляет возможность устанавливать флаги, влияющие на работу регулярных выражений. Обычно флаги устанавливаются посредством их передачи функции `re.compile()` в виде дополнительных параметров, но иногда гораздо удобнее устанавливать их непосредственно в регулярных выражениях. Установка флагов выполняется простой синтаксической конструкцией (?*flags*), где *flags* – один или несколько следующих флагов: `a` (то же самое, что передача флага `re.ASCII`), `i` (`re.IGNORECASE`), `m` (`re.MULTILINE`), `s` (`re.DOTALL`) и `x` (`re.VERBOSE`).[1] При таком

[1] В языке Python используются те же символы флагов, что и в механизме регулярных выражений языка Perl, поэтому символ `s` обозначает флаг `re.DOTALL`, а символ `x` – флаг `re.VERBOSE`.

способе использования флаги должны помещаться в начало регулярного выражения. Эта конструкция не участвует в поиске совпадений – она используется только для установки флагов.

Модуль для работы с регулярными выражениями

Модуль `re` обеспечивает два способа работы с регулярными выражениями. Один из них заключается в использовании функций, перечисленных в табл. 12.5, которым в виде первого аргумента передается регулярное выражение. Каждая функция преобразует регулярное выражение во внутренний формат – этот процесс называется компиляцией, а затем выполняет свою работу. Это очень удобно для однократного применения регулярного выражения, но если одно и то же регулярное выражение требуется применить несколько раз, можно избежать излишних затрат на компиляцию при каждом использовании, скомпилировав выражение всего один раз с помощью функции `re.compile()`. После этого можно вызывать методы скомпилированного регулярного выражения столько раз, сколько потребуется. Методы скомпилированных регулярных выражений перечислены в табл. 12.6.

```
match = re.search(r"#[\dA-Fa-f]{6}\b", text)
```

Этот фрагмент программного кода демонстрирует порядок использования функции из модуля `re`. Данному регулярному выражению соответствует определение цвета в формате HTML (например, `#C0C0AB`). Если совпадение будет найдено, функция `re.search()` вернет объект совпадения; в противном случае она вернет значение `None`. Методы объектов совпадений перечислены в табл. 12.7.

Если предполагается несколько раз использовать одно и то же регулярное выражение, его можно скомпилировать один раз и использовать скомпилированное регулярное выражение везде, где только потребуется:

```
color_re = re.compile(r"#[\dA-Fa-f]{6}\b")
match = color_re.search(text)
```

Как уже отмечалось выше, мы используем «сырые» строки, чтобы избежать необходимости дублировать символы обратного слеша. Это регулярное выражение можно записать иначе, использовав в нем символьный класс `[\dA-F]` и передав флаг `re.IGNORECASE` функции `re.compile()` в виде последнего аргумента, или использовать регулярное выражение `(?i)#[\dA-F]{6}\b`, которое начинается с установки флага, делающего регулярное выражение нечувствительным к регистру символов.

Если требуется установить более одного флага, их можно объединять с помощью оператора ИЛИ (`|`) – например, `re.MULTILINE|re.DOTALL`, или `(?ms)` – в случае встраивания флагов в само регулярное выражение.

Таблица 12.5. Функции модуля регулярных выражений

Синтаксис	Описание
`re.compile(r, f)`	Возвращает скомпилированное регулярное выражение `r` с установленными флагами *f*, если они были заданы
`re.escape(s)`	Возвращает строку `s`, в которой все неалфавитно-цифровые символы экранированы символом обратного слеша, то есть возвращает строку, не содержащую специальных символов регулярных выражений
`re.findall(r, s, f)`	Возвращает все неперекрывающиеся совпадения с регулярным выражением `r` в строке `s` (с учетом флагов *f*, если они заданы). Если регулярное выражение содержит сохраняющие группы, для каждого совпадения возвращается кортеж с сохраненными фрагментами
`re.finditer(r, s, f)`	Возвращает объект совпадения для каждого неперекрывающегося совпадения с регулярным выражением `r` в строке `s` (с учетом флагов *f*, если они заданы)
`re.match(r, s, f)`	Возвращает объект совпадения, если совпадение с регулярным выражением `r` находится в начале строки `s` (с учетом флагов *f*, если они заданы); в противном случае возвращает `None`
`re.search(r, s, f)`	Возвращает объект совпадения, если совпадение с регулярным выражением `r` обнаруживается в любом месте строки `s` (с учетом флагов *f*, если они заданы); в противном случае возвращает `None`
`re.split(r, s, m)`	Возвращает список строк, который создается в результате разбиения строки `s` по каждому совпадению с регулярным выражением `r`. Количество разбиений не превышает *m* (если аргумент *m* не задан, выполняется столько разбиений, сколько возможно). Если регулярное выражение содержит сохраняющие группы, они включаются в список между частями, полученными в результате разбиения
`re.sub(r, x, s, m)`	Возвращает копию строки `s`, в которой каждое (но не более *m*, если задано) совпадение с регулярным выражением `r` замещается значением аргумента `x`, который может быть строкой или функцией – описание приводится в тексте
`re.subn(r, x, s, m)`	То же самое, что и функция `re.sub()`, за исключением того, что она возвращает кортеж из двух элементов – со строкой результата и количеством выполненных подстановок

Таблица 12.6. Методы объекта регулярного выражения

Синтаксис	Описание
`rx.findall (s, start, end)`	Возвращает все неперекрывающиеся совпадения с регулярным выражением в строке `s` (или в срезе *start:end* строки). Если регулярное выражение содержит сохраняющие группы, для каждого совпадения возвращается кортеж с сохраненными фрагментами
`rx.finditer (s, start, end)`	Возвращает объект совпадения для каждого неперекрывающегося совпадения в строке `s` (или в срезе *start:end* строки)
`rx.flags`	Возвращает флаги, которые были установлены при компиляции регулярного выражения
`rx.groupindex`	Словарь, ключами которого являются имена сохраняющих групп, а значениями – номера групп. Пустой, если имена не используются
`rx.match (s, start, end)`	Возвращает объект совпадения, если совпадение с регулярным выражением находится в начале строки `s` (или в начале среза *start:end* строки); в противном случае возвращает `None`
`rx.pattern`	Строка, из которой была получена скомпилированная версия регулярного выражения
`rx.search (s, start, end)`	Возвращает объект совпадения, если совпадение с регулярным выражением обнаруживается в любом месте строки `s` (или в любом месте в срезе *start:end* строки); в противном случае возвращает `None`
`rx.split(s, m)`	Возвращает список строк, который создается в результате разбиения строки `s` по каждому совпадению с регулярным выражением. Количество разбиений не превышает *m* (если аргумент *m* не задан, выполняется столько разбиений, сколько возможно). Если регулярное выражение содержит сохраняющие группы, они включаются в список между частями, полученными в результате разбиения
`rx.sub(x, s, m)`	Возвращает копию строки `s`, в которой каждое (но не более *m*, если задано) совпадение с регулярным выражением замещается значением аргумента `x`, который может быть строкой или функцией, – описание приводится в тексте
`rx.subn(x, s, m)`	То же самое, что и метод `rx.sub()`, за исключением того, что этот метод возвращает кортеж из двух элементов – со строкой результата и количеством выполненных подстановок

Таблица 12.7. Методы и атрибуты объекта совпадения

Синтаксис	Описание
`m.end(g)`	Возвращает конечную позицию совпадения в тексте для группы *g*, если таковая указана (или для группы 0, соответствующей совпадению со всем регулярным выражением); если группа не участвовала в совпадении, возвращается `-1`
`m.endpos`	Позиция конца поиска (конец текста или значение аргумента *end* в методах или функциях `match()` или `search()`)
`m.expand(s)`	Возвращает строку `s`, в которой номера и имена сохраняющих групп (`\1`, `\2`, `\g<name>` и подобные им) замещаются соответствующими сохраненными фрагментами
`m.group(g, ...)`	Возвращает фрагмент, сохраненный в нумерованной или именованной группе `g`; если указано более одной группы, возвращается кортеж соответствующих сохраненных фрагментов (совпадению со всем регулярным выражением соответствует группа 0)
`m.groupdict(default)`	Возвращает словарь имен всех именованных сохраняющих групп, в котором ключами являются имена групп, а значениями – сохраненные фрагменты; если указан аргумент *default*, его значение возвращается для групп, не участвовавших в совпадении
`m.groups(default)`	Возвращает кортеж всех сохраняющих групп, начиная с 1; если указан аргумент *default*, его значение возвращается для групп, не участвовавших в совпадении
`m.lastgroup`	Имя сохраняющей группы с наибольшим порядковым номером, для которой имеется совпадение, или `None`, если таких групп нет или если именованные группы не использовались
`m.lastindex`	Номер последней сохраняющей группы, для которой имеется совпадение, или `None`, если таких групп нет
`m.pos`	Начальная позиция поиска (начало текста или значение аргумента *start* в методах или функциях `match()` или `search()`)
`m.re`	Объект регулярного выражения, который воспроизвел этот объект совпадения
`m.span(g)`	Возвращает начальную и конечную позиции совпадения в тексте для группы *g*, если таковая указана (или для группы 0, соответствующей совпадению со всем регулярным выражением); если группа не участвовала в совпадении, возвращается `(-1, -1)`
`m.start(g)`	Возвращает начальную позицию совпадения в тексте для группы *g*, если таковая указана (или для группы 0, соответствующей совпадению со всем регулярным выражением); если группа не участвовала в совпадении, возвращается `-1`
`m.string`	Строка, которая была передана функциям или методам `match()` или `search()`

Мы завершим этот раздел обзором нескольких примеров и начнем с тех регулярных выражений, которые демонстрировались в предыдущих разделах, чтобы проиллюстрировать применение наиболее часто используемых функций из модуля `re`. Начнем с регулярного выражения, отыскивающего повторяющиеся слова:

```
double_word_re = re.compile(r"\b(?P<word>\w+)\s+(?P=word)(?!\w)",
                            re.IGNORECASE)
for match in double_word_re.finditer(text):
    print("{0} is duplicated".format(match.group("word")))
```

Здесь используется немного более сложная версия по сравнению с той, что мы видели выше. Регулярное выражение начинается с границы слова (чтобы гарантировать, что совпадение будет начинаться с начала слова), затем следует максимально длинное соответствие одному или более символам «слова», затем следует один или более пробельных символов, затем следует то же самое слово еще раз, но при условии, что за вторым вхождением слова не следует какой-либо символ «слова».

Если предположить, что поиск выполняется в тексте «win in vain», то *без* первой проверки было бы получено одно соответствие и два сохранения: `win in vain`. Наличие проверки на границу слова гарантирует, что первое слово в совпадении будет целым словом, в результате в этом тексте не будет найдено совпадений, и ничего не будет сохраняться, так как здесь нет повторяющихся слов. Аналогично, если предположить, что поиск выполняется в тексте «one and and two let's say», то *без* последней проверки было бы найдено два совпадения и выполнено два сохранения: `one and and two let's say`. Опережающая проверка гарантирует, что второе слово в совпадении будет целым словом, в результате в этом тексте будет найдено одно совпадение и выполнено одно сохранение: `one and and two let's say`.

Цикл `for` выполняет итерации по всем объектам совпадений, возвращаемых методом `finditer()`, и мы используем метод `group()` для получения фрагментов текста, сохраненных в группах. Точно так же мы могли бы использовать вызов `group(1)` (хотя это было бы менее понятно тому, кто будет сопровождать такой программный код) – в этом случае нам потребовалось бы отказаться от применения именованной сохраняющей группы и использовать регулярное выражение `(\w+)\s+\1(?!\w)`. Следует также отметить, что в конце регулярного выражения вместо `(?!\w)` можно было бы использовать проверку на границу слова `\b`.

Другим регулярным выражением, представленным выше, было регулярное выражение, отыскивающее имена файлов в тегах изображений в документах HTML. Ниже показано, как при компиляции регулярного выражения можно добавить флаги, чтобы оно не было чувствительно к регистру символов, и поддерживается возможность вставлять комментарии:

```
image_re = re.compile(r"""
               <img\s+              # начало тега
```

```
                [^>]*?              # другие атрибуты
                src=                # начало атрибута src
                (?P<quote>["']?)    # необязательная открывающая кавычка
                (?P<image>[^"'>]+)  # имя файла изображения
                (?(quote)(?P=quote)) # закрывающая кавычка
                [^>]*?              # другие атрибуты
                >                   # конец тега
                """, re.IGNORECASE|re.VERBOSE)
image_files = []
for match in image_re.finditer(text):
    image_files.append(match.group("image"))
```

Здесь мы снова использовали метод `finditer()` для получения всех найденных совпадений и метод `group()` объектов совпадений – для извлечения сохраненного текста. Поскольку регистр символов имеет значение только для выражений `img` и `src`, мы могли бы отбросить флаг `re.IGNORECASE` и использовать выражения `[Ii][Mm][Gg]` и `[Ss][Rr][Cc]`. Это сделало бы регулярное выражение более трудным для чтения, но, возможно, более быстрым, так как при поиске совпадений не будет требоваться преобразовывать его символы в верхний или нижний регистр. Однако, скорее всего, разницу в скорости можно будет заметить только при обработке больших объемов текста.

Одна из типичных задач заключается в том, чтобы взять документ в формате HTML и вывести только обычный текст, содержащийся в нем. Естественно, это можно сделать с применением одного из парсеров, входящих в состав Python, но с помощью регулярных выражений можно создать очень простой инструмент. Чтобы реализовать это, необходимо решить три задачи: удалить все теги, заменить сущности символами, которые они представляют, и вставить пустые строки, отделяющие параграфы. Ниже приводится функция (взята из программы *html2text.py*), которая делает эту работу:

```
def html2text(html_text):
    def char_from_entity(match):
        code = html.entities.name2codepoint.get(match.group(1), 0xFFFD)
        return chr(code)

    text = re.sub(r"<!--(?:.|\n)*?-->", "", html_text)          #1
    text = re.sub(r"<[Pp][^>]*?(?!</)>", "\n\n", text)          #2
    text = re.sub(r"<[^>]*?>", "", text)                        #3
    text = re.sub(r"&#(\d+);", lambda m: chr(int(m.group(1))), text)
    text = re.sub(r"&([A-Za-z]+);", char_from_entity, text)     #5
    text = re.sub(r"\n(?:[ \xA0\t]+\n)+", "\n", text)           #6
    return re.sub(r"\n\n+", "\n\n", text.strip())               #7
```

Первому регулярному выражению `<!--(?:.|\n)*?-->` соответствуют комментарии HTML, включая теги HTML, вложенные в них. Функция `re.sub()` замещает все найденные совпадения указанным текстом – просто удаляет совпадения, если в качестве замены используется пустая строка, как в данном случае. (Имеется возможность определить

максимальное количество совпадений, передав дополнительно целое число в виде последнего аргумента.)

Обратите внимание, что здесь выполняется поиск совпадений минимальной длины, чтобы гарантировать, что в каждом удаляемом совпадении будет только один комментарий; в противном случае мы могли бы удалить все, что находится между началом первого комментария и концом последнего.

Функция `re.sub()` не принимает никаких флагов в виде аргументов, вследствие этого символ `.` в регулярном выражении означает «любой символ за исключением символа перевода строки», поэтому мы должны выполнять поиск `.` или `\n`. Отыскивать эти совпадения следует с использованием оператора выбора, а не с помощью символьного класса, так как внутри символьных классов точка интерпретируется буквально – как символ точки. Как вариант в начало регулярного выражения можно было бы добавить флаг, например `(?s)<!--.*?-->`, или скомпилировать регулярное выражение с флагом `re.DOTALL`; в этом случае регулярное выражение можно было бы упростить до `<!--.*?-->`.

Второму регулярному выражению `<[Pp][^>]*?(?!</)>` соответствуют открывающие теги параграфов (такие как `<P>` или `<p align=center>`). Соответствие начинается с символов `<p` (или `<P`), далее могут следовать любые атрибуты (используется минимальный квантификатор) и, наконец, закрывающий символ `>` при условии, что ему не предшествует символ слеша `/` (выполняется негативная ретроспективная проверка), поскольку наличие этого символа могло бы указывать на закрывающий тег параграфа. Второй вызов функции `re.sub()` использует это регулярное выражение для замены каждого открывающего тега параграфа на два символа перевода строки (обычный способ разделения параграфов в простых текстовых файлах).

Третьему регулярному выражению `<[^>]*?>` соответствуют любые теги, и оно используется в третьем вызове функции `re.sub()` для удаления всех остальных тегов.

Сущности HTML – это способ определить символы, не входящие в набор ASCII, с помощью символов ASCII. Сущности могут записываться в двух формах: `&`*`name`*`;`, где *`name`* – это имя символа, например, сущность `&copy;` обозначает символ ©, и `&#`*`digits`*, где *`digits`* – это десятичные цифры, определяющие числовой код символа в кодировке Юникод, например, `&#165;` обозначает символ ¥. Четвертый вызов функции `re.sub()` использует регулярное выражение `&#(\d+);`, которому соответствуют числовые формы сущностей, а сами цифры сохраняются в группе 1. Вместо фактического текста замены мы передаем лямбда-функцию. Когда функция `re.sub()` получает другую функцию, она будет вызывать ее для каждого найденного совпадения, передавая ей объект совпадения в виде единственного аргумента. Внутри лямбда-функции мы извлекаем цифры (в виде строки), преобразуем их в целое число с помощью встроенной функции `int()` и затем с помощью встроенной функции

chr() получаем символ Юникода с заданным числовым кодом. Возвращаемое значение функции (или, в случае использования лямбда-выражения, результат выражения) используется в качестве текста замены.

Пятый вызов функции re.sub() использует регулярное выражение &([A-Za-z]+);, которое сохраняет именованные сущности. Модуль html.entities из стандартной библиотеки содержит словари сущностей, включая name2codepoint, ключами которого являются имена сущностей, а значениями – целочисленные коды символов. Функция re.sub() вызывает локальную функцию char_from_entity() всякий раз, когда обнаруживает совпадение. Функция char_from_entity() использует метод dict.get() со значением 0xFFFD аргумента, используемым по умолчанию (код символа Юникода, обычно используемого для замены и часто изображаемого как ❓). Это гарантирует получение кода символа и передачу его функции chr(), которая вернет соответствующий символ для замены именованной сущности, – с использованием предопределенного символа Юникода для замены неопознанной сущности.

Регулярное выражение \n(?:[\xA0\t]+\n)+ в шестом вызове функции re.sub() используется для удаления строк, содержащих только пробельные символы. Используемый здесь символьный класс содержит пробел, неразрывный пробел (представляемый сущностью , которая будет замещена предыдущим регулярным выражением) и символ табуляции. Регулярному выражению соответствует символ перевода строки (предшествующей одной или более строкам, состоящим только из пробельных символов) и по меньшей мере одна строка (или более, благодаря максимальному квантификатору), состоящая только из пробельных символов. Поскольку совпадение включает в себя символ перевода строки из строки, предшествующей строкам из пробельных символов, необходимо заменить совпадение одиночным символом перевода строки; в противном случае будут удалены не только пробельные строки, но и символ перевода строки из предшествующей им строки.

Результат седьмого и последнего вызова функции re.sub() возвращается вызывающей программе. Данное регулярное выражение \n\n+ используется для замены последовательностей из двух и более символов перевода строки точно двумя символами перевода строки, чтобы гарантировать, что параграфы будут отделяться друг от друга только одной пустой строкой.

В этом примере ни одна из замещающих строк не была взята непосредственно из совпадения (впрочем, здесь использовались именованные и числовые сущности HTML), но в некоторых ситуациях может возникнуть необходимость включить в строку замены весь текст совпадения или его часть. Например, если представить, что имеется список имен в формате *Имя ВтороеИмя1 ... ВтороеИмяN Фамилия*, в котором может указываться произвольное число вторых имен (или даже ни одного), и нам необходимо создать новый список, каждый элемент

которого имеет формат: *Фамилия Имя ВтороеИмя1 ... ВтороеИмяN*, то этого легко можно было бы добиться с помощью регулярного выражения:

```
new_names = []
for name in names:
    name = re.sub(r"(\w+(?:\s+\w+)*)\s+(\w+)", r"\2, \1", name)
    new_names.append(name)
```

В первой части `(\w+(?:\s+\w+)*)` регулярного выражения первому подвыражению `\w+` соответствует имя, а подвыражению `(?:\s+\w+)*` – ноль или более вторых имен. Подвыражению второго имени соответствуют ноль или более вхождений комбинации из пробельного символа и следующего за ним слова. Второй части `\s+(\w+)` регулярного выражения соответствует пробельный символ, за которым следует имя (и вторые имена) и фамилия.

Если такое регулярное выражение кажется вам бессмысленным нагромождением, можно попробовать использовать именованные сохраняющие группы, чтобы повысить удобочитаемость и простоту сопровождения:

```
name = re.sub(r"(?P<forenames>\w+(?:\s+\w+)*)"
              r"\s+(?P<surname>\w+)",
              r"\g<surname>, \g<forenames>", name)
```

Получить доступ к сохраненному тексту в функциях или в методах `sub()` и `subn()` можно с помощью синтаксической конструкции `\`*`i`* или `\g<`*`id`*`>`, где *`i`* – это номер сохраняющей группы, а *`id`* – это имя или номер сохраняющей группы, то есть `\1` – это то же самое, что и `\g<1>`, а в данном примере это то же самое, что и `\g<forenames>`. Этот же синтаксис можно использовать в строках, передаваемых методу `expand()` объекта совпадения.

Почему первая часть регулярного выражения не захватывает имя, вторые имена и отчество целиком? В конце концов, в ней используется максимальный квантификатор. В действительности именно это и происходит, но тогда вторая часть регулярного выражения терпит неудачу. Часть, соответствующая вторым именам, допускает ноль или более совпадений, а часть, соответствующая фамилии, должна иметь точно одно совпадение, но из-за своей жадности выражение, соответствующее вторым именам, захватило все. Когда обнаруживается неудача, механизм регулярных выражений выполняет шаг назад, освобождая последнее «второе имя» и обеспечивая тем самым совпадение для части выражения, которая соответствует фамилии.

Несмотря на то, что подвыражения с максимальными квантификаторами стремятся отыскать соответствие максимально возможной длины, тем не менее они останавливаются, достигнув позиции, когда дальнейшее увеличение длины совпадения может привести к неудаче.

Например, если представить, что текст содержит имя «James W. Loewen», регулярное выражение сначала найдет совпадение всего имени с первой частью, то есть `James W. Loewen`. Это удовлетворяет первую часть регулярного выражения, но оставляет ни с чем вторую часть, соответствующую фамилии, а поскольку совпадение с этой частью является обязательным (она имеет неявный квантификатор `{1}`), то регулярное выражение в целом потерпит неудачу. Так как часть выражения, соответствующая вторым именам, снабжена квантификатором `*`, для нее может иметься ноль или более совпадений (в текущий момент для нее обнаружено два совпадения: «W.» и «Loewen»), поэтому механизм регулярных выражений заставляет эту часть выражения отдать последнее совпадение, чтобы все регулярное выражение не потерпело неудачу. Такой шаг назад возвращает последнее совпадение с `\s+\w` (то есть текст «Loewen»); тогда совпадение приобретает вид `James W. Loewen`, которое удовлетворяет все регулярное выражение целиком, и в двух группах сохраняется корректный текст.

При использовании оператора выбора (`|`) с двумя или более сохраняющими альтернативами нет никакой возможности определить, какая из альтернатив обнаружила совпадение, поэтому нельзя определить, из какой группы следует извлекать сохраненный текст. Конечно, можно выполнить итерации по всем группам и отыскать непустую, но очень часто в подобной ситуации атрибут `lastindex` объекта совпадения может содержать номер нужной группы. Чтобы проиллюстрировать это и получить дополнительные практические навыки работы с регулярными выражениями, рассмотрим последний пример.

Предположим, что нам требуется определить, какая кодировка используется в файлах HTML, XML или Python. Мы могли бы открыть файл в двоичном режиме и прочитать, например, первые 1 000 байтов в объект типа `bytes`. После этого мы могли бы закрыть файл, определить кодировку по фрагменту в объекте `bytes` и вновь открыть файл в текстовом режиме с использованием обнаруженной кодировки или кодировки по умолчанию (UTF-8). Механизм регулярных выражений ожидает, что регулярные выражения будут поставляться ему в виде строк, а текст, к которому применяется регулярное выражение, может быть представлен объектом типа `str`, `bytes` или `bytearray`, причем, когда используются объекты типа `bytes` или `bytearray`, все функции и методы вместо строк возвращают результат типа `bytes`, при этом неявно устанавливается флаг `re.ASCII`.

В файлах HTML кодировка обычно указывается в теге `<meta>` (если он вообще присутствует), например, `<meta http-equiv='Content-Type' content='text/html; charset=ISO-8859-1'/>`. В файлах XML по умолчанию подразумевается кодировка UTF-8, но имеется возможность явно определить другую кодировку, например, `<?xml version="1.0" encoding="Shift_JIS"?>`. В Python 3 для файлов с исходным программным кодом также по умолчанию подразумевается кодировка UTF-8, но здесь так-

же имеется возможность явно определить другую кодировку, включив в файл такую строку: `latin1` или `# -*- coding: latin1 -*-` сразу вслед за строкой «shebang».

Ниже показано, как отыскать кодировку в предположении, что переменная `binary` ссылается на объект `bytes`, содержащий первые 1 000 байтов из файла HTML, XML или Python:

```
match = re.search(r"""(?<![-\w])                        #1
                      (?:(?:en)?coding|charset)         #2
                      (?:=(["']))?([-\w]+)(?(1)\1)      #3
                      |:\s*([-\w]+))""".encode("utf8"),
                  binary, re.IGNORECASE|re.VERBOSE)
encoding = match.group(match.lastindex) if match else b"utf8"
```

Чтобы выполнить поиск по объекту типа `bytes`, необходимо указать текст регулярного выражения также в виде объекта `bytes`. В данном случае мы предпочли воспользоваться удобствами, которые дают «сырые» строки, поэтому использовали в первом аргументе функции `re.search()` такую строку, попутно преобразовав ее в объект типа `bytes`.

Поиск совпадения по условию, стр. 536

Первая часть самого регулярного выражения является ретроспективной проверкой, которая говорит, что совпадению не может предшествовать дефис или символ «слова». Второй части выражения соответствуют слова «encoding», «coding» и «charset», и ее можно было бы записать как `(?:encoding|coding|charset)`. Третья часть выражения была записана в двух строках, чтобы подчеркнуть, что она состоит из двух вариантов: `=(["'])?([-\w]+)(?(1)\1)` и `:\s*([-\w]+)`, из которых только один может совпадать с текстом. Первой части соответствует знак равенства, за которым следует один или более символов «слова» или дефисов (которые, возможно, заключены в пару однотипных кавычек; применяется прием определения совпадения по условию), а второй части соответствует символ двоеточия, за которым следует необязательный пробельный символ и один или более символов «слова» или дефисов. (Не забывайте, что внутри символьных классов символ дефиса интерпретируется просто как символ, если он стоит первым; в противном случае он обозначает диапазон символов, например, `[0-9]`.)

Мы использовали флаг `re.IGNORECASE`, чтобы избежать необходимости записывать выражение `(?:(?:[Ee][Nn])?[Cc][Oo][Dd][Ii][Nn][Gg]|[Cc][Hh][Aa][Rr][Ss][Ee][Tt])`, и флаг `re.VERBOSE`, чтобы регулярное выражение можно было оформить более удобочитаемым способом и снабдить его комментариями (в данном случае это всего лишь числа, отмечающие части регулярного выражения, на которые потом делаются ссылки в тексте).

Здесь имеется три сохраняющих группы, причем все они находятся в третьей части: первая группа (["']?) сохраняет необязательную открывающую кавычку, вторая группа ([-\w]+) сохраняет название кодировки, следующее за знаком =, и третья группа ([-\w]+) (в следующей строке) сохраняет название кодировки, следующее за двоеточием. Нас интересует только название кодировки, поэтому нам требуется извлечь только фрагмент, сохраненный второй или третьей группой, причем только из той, которая участвовала в совпадении, так как они являются альтернативными. Атрибут lastindex содержит индекс последней *совпавшей* группы (либо 2, либо 3 в этом примере, если совпадение с регулярным выражением будет найдено), поэтому мы извлекаем содержимое совпавшей группы или используем кодировку по умолчанию, если совпадение не было найдено.

К настоящему моменту мы познакомились со всеми наиболее часто используемыми функциональными возможностями модуля re в действии; тем не менее в заключение этого раздела упомянем еще одну, последнюю функцию. Функция split() (или метод split() объекта регулярного выражения) может разбивать строки с применением регулярных выражений. На практике часто бывает необходимо разбить текст по пробельным символам, чтобы получить список слов (или, более точно, список строк, каждая из которых соответствует регулярному выражению \S+). Регулярные выражения обладают широчайшими возможностями и, изучив их, легко заметить, что любые задачи, связанные с обработкой текста, могут быть решены с привлечением регулярных выражений. Но иногда строковые методы дают более высокую производительность и в большей степени соответствуют решаемой задаче. Например, мы легко можем разбить текст по пробельным символам, используя метод text.split(), так как метод str.split() по умолчанию (или с первым аргументом None) выполняет разбиение по регулярному выражению \s+.

В заключение

Регулярные выражения обладают широкими возможностями поиска в тексте строк, соответствующих определенному шаблону, и замены таких строк другими строками, содержимое которых может создаваться на основе найденных строк.

В этой главе мы узнали, что большинство символов в регулярных выражениях соответствуют сами себе и неявно сопровождаются квантификатором {1}. Мы также узнали, как определять символьные классы – множества символов, как инвертировать такие множества и включать в них диапазоны символов, не указывая каждый символ в отдельности.

Мы узнали, как квантифицировать регулярные выражения, чтобы обеспечить поиск определенного числа совпадений или числа совпаде-

ний не менее и не более определенного числа раз. Как использовать максимальные и минимальные квантификаторы. Мы также узнали, как группировать подвыражения, чтобы к ним можно было применить квантификатор как к единому целому (и при желании обеспечить сохранение совпавшего фрагмента).

В этой главе также было показано, какое влияние на совпадения оказывают различные проверки, такие как позитивные и негативные опережающая и ретроспективная проверки, и различные флаги, которые, например, управляют интерпретацией символа точки и обеспечивают возможность поиска совпадений без учета регистра символов.

В заключительном разделе было показано, как использовать регулярные выражения в контексте программирования на языке Python. В этом разделе мы узнали, как использовать функции, предоставляемые модулем `re`, и методы объектов скомпилированных регулярных выражений и объектов совпадений. Мы также узнали, как замещать найденные совпадения простыми строками, строками, содержащими обратные ссылки, и результатами вызова функций или лямбда-выражений, а также – как сделать регулярные выражения более удобочитаемыми, используя именованные сохраняющие группы и комментарии.

Упражнения

1. Во многих случаях (например, при заполнении веб-форм), пользователи должны вводить номер телефона, причем в некоторых случаях допускается ввод только в определенном формате, что доставляет дополнительные неудобства пользователям. Напишите программу, которая будет читать номера телефонов США, где за первыми тремя цифрами, обозначающими код города, следуют семь цифр местного номера, принимая их как десять цифр, идущих подряд или разделенных на блоки с помощью пробелов и дефисов, причем код города может быть заключен в необязательные круглые скобки. Например, все следующие номера считаются допустимыми: 555-555-5555, (555) 5555555, (555) 555 5555 и 5555555555. Каждый номер телефона должен читаться из `sys.stdin` и выводиться в формате «(555) 555 5555», а в случае ввода недопустимого номера должно выводиться сообщение об ошибке.

   Регулярное выражение, которому соответствуют номера телефонов, занимает примерно восемь строк (при использовании флага `re.VERBOSE`) и само по себе достаточно простое. Решение приводится в файле *phone.py*, размер которого составляет двадцать пять строк.

2. Напишите небольшую программу, читающую файлы XML и HTML, имена которых задаются в командной строке. Программа должна отыскивать теги с атрибутами и выводить имена тегов с их атрибутами под ними. Например, ниже приводится фрагмент вывода про-

граммы, которой был передан один из файлов *index.html*, входящих в состав документации к языку Python:

```
html
    xmlns = http://www.w3.org/1999/xhtml
meta
    http-equiv = Content-Type
    content = text/html; charset=utf-8
li
    class = right
    style = margin-right: 10px
```

Один из вариантов решения этой задачи заключается в использовании двух регулярных выражений, одно из которых сохраняет имена тегов и их атрибутов, а второе извлекает имя и значение каждого атрибута. Значения атрибутов могут окружаться кавычками или апострофами (в этом случае они могут содержать пробелы и кавычки, по своему типу не совпадающие с кавычками, окружающими значение), но могут указываться и без кавычек (в этом случае они не могут содержать пробелы и кавычки). Вероятно, проще всего начать с создания двух регулярных выражений для обработки значений в кавычках и без кавычек, а затем объединить их в общем регулярном выражении, обрабатывающем оба случая. Чтобы сделать регулярное выражение более удобочитаемым, лучше использовать именованные группы. Эта задача не из легких, особенно если учесть, что обратные ссылки не могут использоваться в символьных классах.

Решение приводится в файле *extract_tags.py*, содержащем менее 35 строк программного кода. Регулярное выражение, извлекающее имена тегов и атрибутов, занимает одну строку. Регулярное выражение, извлекающее имена атрибутов и их значения, занимает полдюжины строк и использует альтернативы, условное совпадение (дважды, одно вложено в другое), а также максимальные и минимальные квантификаторы.

13

- Программы в виде диалога
- Программы с главным окном

Введение в программирование графического интерфейса

В языке Python отсутствует собственная поддержка возможности создания графического интерфейса, но в этом нет никакой проблемы, так как программисты, разрабатывающие программы на языке Python, могут использовать библиотеки графического интерфейса, написанные для других языков программирования. Такое возможно благодаря тому, что для языка Python существует множество *оберток*, или *привязок*, для большинства распространенных библиотек создания графического интерфейса. Эти пакеты и модули могут импортироваться и использоваться, как любые другие пакеты и модули в языке Python, при этом они обеспечивают доступ к функциональным возможностям, находящимся в библиотеках, написанных не на языке Python.

Стандартная библиотека языка Python включает в себя связку Tcl/Tk. Tcl – это язык сценариев со свободным синтаксисом, а Tk – библиотека для создания графического интерфейса, написанная на языках Tcl и C. Модуль `tkinter` языка Python обеспечивает привязку к графическому интерфейсу библиотеки Tk. В сравнении с другими библиотеками графического интерфейса библиотека Tk обладает тремя преимуществами, доступными для языка Python. Во-первых, она устанавливается вместе с интерпретатором Python, поэтому она всегда находится в распоряжении программиста. Во-вторых, она имеет небольшой размер (даже вместе с интерпретатором Tcl). И в-третьих, она поставляется вместе со средой разработки IDLE, которую удобно использовать для разработки и отладки программ на языке Python.

К сожалению, до версии 8.5 библиотека Tk воспроизводила графический интерфейс в соответствии с отошедшими в прошлое представлениями о графическом оформлении и содержала весьма ограниченный набор виджетов («элементов управления», или «контейнеров», в тер-

минологии Windows). Хотя с помощью библиотеки Tk легко можно создавать свои собственные виджеты путем компоновки других виджетов, но эта библиотека не обеспечивает прямую возможность создания собственных виджетов с нуля, позволяющую программисту рисовать необходимые ему элементы. Дополнительные виджеты, совместимые с библиотекой Tk, можно найти в библиотеке Tix, также входящей в состав стандартной библиотеки языка Python, но она не всегда предоставляется для платформ, отличных от Windows. Особняком в этом отношении стоит дистрибутив Ubuntu, который к моменту написания этих строк был единственным, который содержал эту библиотеку, хотя и в виде неподдерживаемого дополнения. Для библиотек Tk и Tix очень мало документации, ориентированной на применение этих библиотек в языке Python, – большая часть документации написана для программистов Tcl/Tk и сложна для понимания тем, кто не связан с программированием на языке Tcl.[1]

Если перед вами стоит цель создавать программы на языке Python, которые могут выполняться на любых платформах (например, Windows, Mac OS X и Linux) и используют только стандартные возможности, без привлечения дополнительных библиотек, у вас остается единственный выбор: Tk.

Если имеется возможность использовать сторонние библиотеки, число возможных вариантов возрастает существенно. Один из них состоит в том, чтобы получить комплект инструментов WCK (Widget Construction Kit – набор инструментов для создания виджетов, *www.effbot.org/zone/wck.htm*), обеспечивающий дополнительные функциональные возможности, совместимые с библиотекой Tk, включая создание собственных виджетов, содержимое которых создается программным кодом.

Все остальные варианты, в которых не используется библиотека Tk, делятся на две категории – конкретные для определенной платформы и платформенно-независимые. Платформенно-зависимые библиотеки графического интерфейса обеспечивают доступ к платформенно-зависимым особенностям, но они жестко привязывают нас к определенной платформе. Наиболее известными кросс-платформенными привязками для библиотек графического интерфейса в языке Python являются: PyGtk (*www.pygtk.org*), PyQt (*www.riverbankcomputing.com/software/pyqt*) и wxPython (*www.wxpython.org*). Все три библиотеки предлагают намного большее количество виджетов, чем библиотека Tk, позволяют воспроизводить более привлекательный графический интерфейс (хотя

[1] Автору известна единственная книга о применении Tk в языке Python – «Python and Tkinter Programming» Джона Грейсона (John Grayson), ISBN 1884777813, опубликованная в 2000 году; в некоторых вопросах она уже устарела. Хорошая книга о Tcl/Tk: «Practical Programming in Tcl and Tk» Брента Велша (Brent Welch) и Кена Джонса (Ken Jones), ISBN 0130385603. Вся документация по Tcl/Tk доступна по адресу *www.tcl.tk*.

этот недостаток в некоторой степени был устранен в версии Tk 8.5) и позволяют создавать собственные виджеты, которые воспроизводятся программным кодом. В изучении и использовании все они более простые, чем Tk, и для всех имеется гораздо больше документации, ориентированной на программистов, использующих язык Python, чем для Tk. В общем случае программы, использующие PyGtk, PyQt или wxPython, получаются более короткими и дают более качественный результат, нежели программы, использующие Tk.

И все же, несмотря на все ограничения и недостатки, библиотеку Tk можно использовать для создания полезных программ с графическим интерфейсом – среда разработки IDLE является наиболее известным в мире Python подтверждением этих слов. Кроме того, во время написания этой книги появились признаки оживления в разработке библиотеки Tk. Начиная с версии 8.5, в библиотеке была реализована поддержка тем оформления, что обеспечивает большее сходство графического интерфейса программ с графическим интерфейсом операционной системы, а также появилось множество новых виджетов.

Цель этой главы состоит в том, чтобы дать вам общее представление о программировании с применением библиотеки Tk. При необходимости разработки серьезного графического интерфейса лучше не обращаться к материалу этой главы (так как в ней демонстрируются устаревшие подходы к использованию библиотеки Tk для создания графического интерфейса) и воспользоваться одной из альтернативных библиотек. Но если Tk – единственная доступная для вас возможность, то вам определенно следует изучить язык Tcl, чтобы иметь возможность читать документацию для библиотеки Tk.

В следующих разделах мы создадим две программы с графическим интерфейсом с использованием библиотеки Tk. Первая, очень маленькая программа, в форме диалога, позволяет вычислять сложные проценты по банковским вкладам. Вторая программа – более сложная, она имеет главное окно и управляет списком закладок (имена и адреса URL). Благодаря использованию таких простых данных мы можем сконцентрировать все свое внимание на программировании графического интерфейса, не отвлекаясь на посторонние детали. В процессе обзора программы управления закладками мы увидим, как создавать свои собственные диалоги и как создавать главное окно программы, с полосой меню и панелями инструментов, а также как все это соединить в единую полноценную работающую программу. Но для начала нам необходимо рассмотреть некоторые основные принципы программирования графического интерфейса, поскольку они существенно отличаются от принципов создания консольных программ.

Консольные программы на языке Python и файлы модулей всегда имеют расширение *.py*, но для файлов программ с графическим интерфейсом используется расширение *.pyw* (однако для файлов модулей всегда используется только расширение *.py*). Для операционной системы

Linux нет никакой разницы между расширениями *.py* и *.pyw*, но в Windows расширение *.pyw* ассоциировано не с выполняемым файлом интерпретатора *python.exe*, а с *pythonw.exe*, вследствие чего при запуске программы на языке Python с графическим интерфейсом не будет появляться окно консоли. В операционной системе Mac OS X похожая ситуация – для файлов программ с графическим интерфейсом используется расширение *.pyw*.

Когда запускается программа с графическим интерфейсом, она обычно начинает с того, что создает главное окно и все виджеты главного окна, такие как полоса меню, панели инструментов, центральная область и строка состояния. После создания окна программа с графическим интерфейсом, подобно серверной программе, переходит в режим ожидания. Однако если серверная программа ожидает получения запросов от программ-клиентов, то программа с графическим интерфейсом ожидает некоторых действий пользователя, таких как щелчок мышью или нажатие клавиши. Эти различия между консольными программами и программами с графическим интерфейсом изображены на рис. 13.1. Но программа с графическим интерфейсом не просто пассивно ожидает, она выполняет *цикл обработки событий*, алгоритм работы которого на псевдокоде выглядит, как показано ниже:

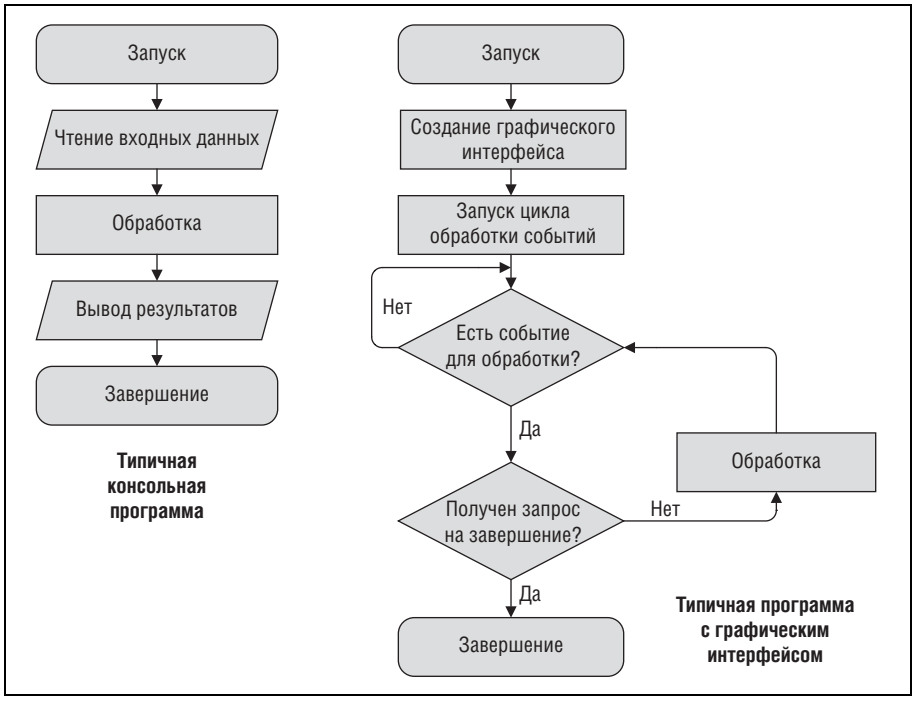

Рис. 13.1. *Консольная программа и программа с графическим интерфейсом*

```
while True:
    event = getNextEvent()
    if event:
        if event == Terminate:
            break
        processEvent(event)
```

Когда пользователь взаимодействует с программой, или когда возникают какие-либо другие события, такие как событие от таймера, или когда окно программы активируется (возможно, из-за того что было закрыто окно какой-нибудь другой программы), внутри библиотеки графического интерфейса генерируется событие, которое затем добавляется в очередь событий. Цикл обработки событий в программе постоянно проверяет эту очередь, чтобы определить, не появились ли события, требующие обработки, и если такие события появились, обрабатывает их (или передает для обработки соответствующим функциям и методам).

При создании программ с графическим интерфейсом мы можем опираться на цикл обработки событий, реализованный в библиотеке графического интерфейса. На нашу долю остается создать классы, представляющие окна и виджеты, необходимые программе, и снабдить их методами, которые будут соответствующим образом обрабатывать действия пользователя.

Программы в виде диалога

Первая программа, которую мы рассмотрим, – это программа Interest. Эта программа, имеющая вид диалога (то есть она не имеет полосы меню), может использоваться для вычисления сложных процентов по банковским вкладам. Окно программы представлено на рис. 13.2.

В большинстве объектно-ориентированных программ с графическим интерфейсом для представления единственного главного окна или диа-

Рис. 13.2. *Программа Interest*

логов используются классы, а большинство виджетов, содержащихся в окне, являются экземплярами стандартных виджетов, таких как кнопки или флажки, предоставляемые библиотекой. Как и большинство кросс-платформенных библиотек графического интерфейса, библиотека Tk в действительности не различает главное окно и виджеты, окно – это обычный виджет, который не имеет родительского виджета (то есть он не содержится внутри другого виджета). Виджеты, не имеющие родительского виджета (окна), автоматически снабжаются рамкой и прочими визуальными атрибутами окна (такими как строка заголовка и кнопка закрытия окна) и, как правило, содержат другие виджеты.

В большинстве своем виджеты создаются как дочерние по отношению к другим виджетам (и содержатся внутри своих родительских виджетов), а окна создаются как дочерние виджеты объекта `tkinter.Tk`. Этот объект обычно концептуально представляет приложение. Помимо разделения на виджеты и окна (которые также называются виджетами верхнего уровня), отношения родитель-потомок обеспечивают корректный порядок удаления виджетов и автоматическое удаление всех дочерних виджетов при удалении родительского виджета.

Метод инициализации служит местом, где создается пользовательский интерфейс (создаются и размещаются виджеты, выполняется привязка клавиатуры и мыши), а другие методы используются для обработки действий пользователя. Библиотека Tk позволяет создавать собственные виджеты либо посредством создания подклассов предопределенных виджетов, таких как `tkinter.Frame`, либо посредством создания обычных классов и добавления в них виджетов в виде атрибутов. В первом примере мы будем создавать свои подклассы, а в следующем будут применены оба подхода.

Так как программа Interest имеет единственное главное окно, она реализована в виде единственного класса. Начнем разбор с метода инициализации, разбитого на пять частей, так как он достаточно длинный.

```
class MainWindow(tkinter.Frame):

    def __init__(self, parent):
        super(MainWindow, self).__init__(parent)
        self.parent = parent
        self.grid(row=0, column=0)
```

Сначала вызывается метод инициализации базового класса. (В идеале можно было бы вызвать метод как `super().__init__(parent)`, но к моменту написания этих строк такой способ уже не работал, поэтому вызывается функция `super()` с базовым классом и объектом `self` в виде явно передаваемых аргументов.) Затем сохраняется копия родительского виджета для последующего использования. Виджеты располагаются внутри других виджетов с помощью менеджеров компоновки, а не с помощью абсолютных координат и размеров. Вызов метода `grid()` помещает окно внутрь менеджера компоновки типа «сетка». Каждый

отображаемый виджет должен быть помещен в менеджер компоновки, даже виджет верхнего уровня. В библиотеке Tk имеется несколько менеджеров компоновки, но менеджер типа «сетка» является наиболее простым в использовании, хотя для виджетов верхнего уровня, чтобы добиться того же эффекта, можно было бы использовать менеджер компоновки типа «мозаика» (*packer*), вызвав метод `pack()` вместо `grid(row=0, column=0)`.

```
self.principal = tkinter.DoubleVar()
self.principal.set(1000.0)
self.rate = tkinter.DoubleVar()
self.rate.set(5.0)
self.years = tkinter.IntVar()
self.amount = tkinter.StringVar()
```

Библиотека Tk позволяет создавать переменные, ассоциированные с виджетами. Если значение переменной изменяется программным способом, изменения тут же отражаются в соответствующем ей виджете, и точно так же, если пользователь изменяет значение в виджете, изменяется значение в ассоциированной переменной. Здесь были созданы две переменные типа «double» (они хранят значения типа `float`), целочисленная переменная, строковая переменная и заданы начальные значения в двух из них.

```
principalLabel = tkinter.Label(self, text="Principal $:",
                               anchor=tkinter.W, underline=0)
principalScale = tkinter.Scale(self, variable=self.principal,
        command=self.updateUi, from_=100, to=10000000,
        resolution=100, orient=tkinter.HORIZONTAL)
rateLabel = tkinter.Label(self, text="Rate %:", underline=0,
                          anchor=tkinter.W)
rateScale = tkinter.Scale(self, variable=self.rate,
        command=self.updateUi, from_=1, to=100,
        resolution=0.25, digits=5, orient=tkinter.HORIZONTAL)
yearsLabel = tkinter.Label(self, text="Years:", underline=0,
                           anchor=tkinter.W)
yearsScale = tkinter.Scale(self, variable=self.years,
        command=self.updateUi, from_=1, to=50,
        orient=tkinter.HORIZONTAL)
amountLabel = tkinter.Label(self, text="Amount $",
                            anchor=tkinter.W)
actualAmountLabel = tkinter.Label(self,
        textvariable=self.amount, relief=tkinter.SUNKEN,
        anchor=tkinter.E)
```

В этой части метода инициализации создаются виджеты. Виджет `tkinter.Label` используется для отображения текста, доступного только для чтения. Как и в случае любых других виджетов, при его создании определяется родительский виджет (в данном случае, как и в любом другом, родителем является содержащий его виджет), а затем через именованные аргументы задаются настройки различных аспектов

поведения виджета и его внешнего вида. Для виджета principalLabel мы определили отображаемый им текст, а в аргументе anchor передали значение tkinter.W, которое означает, что текст метки будет выравниваться по западному (west), то есть по левому краю. Аргумент underline определяет, какой символ в тексте будет использоваться для обозначения *горячей клавиши* (например, Alt+P); далее будет показано, как работать с горячими клавишами. (Горячая клавиша – это комбинация клавиш вида Alt+символ, где символ – это подчеркнутый символ метки; при нажатии горячей клавиши фокус ввода передается виджету, ассоциированному с горячей клавишей, обычно расположенному правее или ниже метки, в которой определен символ горячей клавиши.)

Виджетам tkinter.Scale передается в качестве родителя, как обычно, объект self и ассоциированная с каждым из них переменная. Кроме того, в аргументе command им передаются ссылки на объекты функций (в данном случае методы). Эти методы будут вызываться автоматически при каждом изменении положения движка, устанавливаются минимальное (имя аргумента from_ записывается с завершающим символом подчеркивания, потому что from – это зарезервированное ключевое слово) и максимальное (to) значения и определяется горизонтальная ориентация. Для некоторых движков указывается разрешение (resolution) – шаг изменения значений, а для rateScale – еще и число отображаемых цифр.

Виджет actualAmountLabel также имеет ассоциированную переменную, благодаря этому позднее мы легко можем изменять отображаемый им текст. Кроме того, мы придали этой метке вид утопленного поля, чтобы визуально он гармонировал с движками.

```
principalLabel.grid(row=0, column=0, padx=2, pady=2,
                    sticky=tkinter.W)
principalScale.grid(row=0, column=1, padx=2, pady=2,
                    sticky=tkinter.EW)
rateLabel.grid(row=1, column=0, padx=2, pady=2,
               sticky=tkinter.W)
rateScale.grid(row=1, column=1, padx=2, pady=2,
               sticky=tkinter.EW)
yearsLabel.grid(row=2, column=0, padx=2, pady=2,
                sticky=tkinter.W)
yearsScale.grid(row=2, column=1, padx=2, pady=2,
                sticky=tkinter.EW)
amountLabel.grid(row=3, column=0, padx=2, pady=2,
                 sticky=tkinter.W)
actualAmountLabel.grid(row=3, column=1, padx=2, pady=2,
                       sticky=tkinter.EW)
```

После создания виджетов их нужно разместить. Мы будем использовать схему размещения типа «сетка», как показано на рис. 13.3.

Все виджеты поддерживают метод grid() (и некоторые другие методы схемы размещения, такие как pack()). Вызовом метода grid() виджет

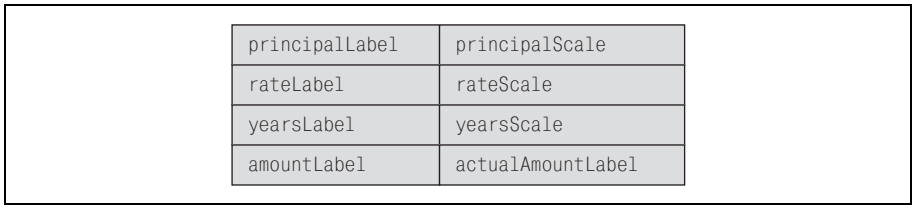

Рис. 13.3. *Схема размещения виджетов в окне программы Interest*

помещается внутрь родительского виджета в ячейку сетки с указанным номером строки и столбца. Имеется возможность заставить виджет занять несколько столбцов и несколько строк, используя дополнительные именованные аргументы (`rowspan` и `columnspan`), а также имеется возможность добавить отступы вокруг виджета – с помощью именованных аргументов `padx` (отступы слева и справа) и `pady` (отступы сверху и снизу), определяя в них величину отступов в пикселях. Если виджет занимает больше места, чем ему требуется, с помощью аргумента `sticky` можно указать, что он должен делать со свободным пространством; если аргумент не задан, виджет будет занимать середину выделенного пространства. Для всех меток в первом столбце в аргументе `sticky` передается значение `tkinter.W` (west – выравнивание по западному, то есть по левому краю), а для всех виджетов в правом столбце – значение `tkinter.EW` (east-west – с востока на запад, то есть по ширине), что заставляет их растягиваться по всей ширине доступного пространства.

Все виджеты сохраняются в локальных переменных, но они не будут утилизированы сборщиком мусора после выхода из метода инициализации благодаря отношениям родитель-потомок, потому что для всех виджетов определен родительский виджет – главное окно. Иногда виджеты создаются в виде переменных экземпляра – например, когда необходимо обеспечить доступ к ним за пределами метода инициализации, но в данном случае мы использовали переменные экземпляра в роли переменных, ассоциированных со значениями виджетов (`self.principal`, `self.rate` и `self.years`), благодаря чему они будут доступны за пределами метода инициализации.

```
principalScale.focus_set()
self.updateUi()
parent.bind("<Alt-p>", lambda *ignore: principalScale.focus_set())
parent.bind("<Alt-r>", lambda *ignore: rateScale.focus_set())
parent.bind("<Alt-y>", lambda *ignore: yearsScale.focus_set())
parent.bind("<Control-q>", self.quit)
parent.bind("<Escape>", self.quit)
```

В конце метода инициализации фокус ввода передается виджету `principalScale`, чтобы сразу после запуска программы пользователь мог

установить начальную сумму вклада. Затем вызывается метод `self.updateUi()`, вычисляющий конечную общую сумму с учетом процентов.

Затем выполняются привязки горячих клавиш. (К сожалению, термин *привязка* имеет три совершенно разных значения – привязка переменной, когда имя, то есть ссылка на объект, связывается с объектом; привязка горячей клавиши, когда событие нажатия или отпускания клавиши связывается с функцией или методом, вызываемым при появлении этого события; и привязка для библиотеки – это промежуточный программный код, обеспечивающий доступность библиотеки, написанной на другом языке программирования, в программах на языке Python, посредством использования модулей Python.) Привязка горячих клавиш особенно важна для пользователей с ограниченными возможностями, испытывающих сложности или вообще не способных использовать мышь, а также для тех, кто в совершенстве владеет клавиатурой и старается не пользоваться мышью, так как в их случае использование мыши замедляет темп работы.

Первые три привязки горячих клавиш используются для передачи фокуса ввода виджетам движков. Например, для виджета `principalLabel` устанавливается отображаемый текст `Principal $:`, в котором нулевой символ выводится с подчеркиванием, поэтому текст метки будет выглядеть, как Principal $:, и первая привязка горячей клавиши переместит фокус ввода в виджет `principalScale`, когда пользователь нажмет комбинацию клавиш Alt+P. То же относится и к другим двум привязкам. Обратите внимание, что в привязках мы не указываем метод `focus_set()` непосредственно, потому что при вызове функции или метода по событию в первом аргументе им передается само событие, которое нам не нужно. По этой причине мы используем лямбда-функции, которые принимают, но игнорируют объекты событий и вызывают методы без нежелательного аргумента.

Мы также создали две дополнительные *горячие комбинации*, то есть комбинации клавиш, которые вызывают определенное действие. Мы настроили комбинации клавиш Ctrl+Q и Esc, привязав их к методу `self.quit()`, который вызывает завершение программы.

Имеется возможность привязывать горячие клавиши непосредственно к самим виджетам, но мы предпочли выполнить все привязки в родительском виджете (в приложении), поэтому они будут действовать независимо от того, где находится фокус ввода.

Метод `bind()` из библиотеки Tk может использоваться для привязки нажатий клавиш и щелчков мышью, а также для привязки программных событий. Специальные клавиши, такие как Ctrl и Esc, в библиотеке Tk имеют собственные имена (`Control` и `Escape`), а символьным клавишам соответствуют имена, состоящие из самих символов. Комбинации клавиш создаются посредством помещения имен, составляющих комбинацию клавиш, разделенных символом дефиса, в угловые скобки.

Создав и разместив виджеты и настроив горячие комбинации клавиш, мы определили основное поведение приложения. Теперь рассмотрим методы, отвечающие за реакцию на действия пользователя и дополняющие поведение приложения.

```
def updateUi(self, *ignore):
    amount = self.principal.get() * (
             (1 + (self.rate.get() / 100.0)) ** self.years.get())
    self.amount.set("{0:.2f}".format(amount))
```

Этот метод вызывается всякий раз, когда пользователь изменяет сумму вклада, процент или интервал времени, поскольку этот метод связан с каждым из движков. Все, что он делает, – извлекает значения движков из ассоциированных переменных, выполняет вычисление процентов и записывает результат (в виде строки) в переменной, ассоциированной с меткой, представляющей окончательную сумму вклада. Благодаря этому метка всегда отображает актуальное значение.

```
def quit(self, event=None):
    self.parent.destroy()
```

Этот метод вызывается, когда пользователь завершает работу с программой (нажатием комбинации клавиш Ctrl+Q или Esc, или щелчком мыши на кнопке закрытия окна). Поскольку в программе нет данных, которые требовалось бы сохранить, мы просто сообщаем родительскому виджету (который представляет само приложение), что он должен уничтожить себя. Родительский виджет уничтожит все дочерние виджеты – все окна, которые в свою очередь уничтожат все свои дочерние виджеты, благодаря чему приложение корректно завершит свою работу.

```
application = tkinter.Tk()
path = os.path.join(os.path.dirname(__file__), "images/")
if sys.platform.startswith("win"):
    icon = path + "interest.ico"
else:
    icon = "@" + path + "interest.xbm"
application.iconbitmap(icon)
application.title("Interest")
window = MainWindow(application)
application.protocol("WM_DELETE_WINDOW", window.quit)
application.mainloop()
```

За определением класса главного (и в данном случае – единственного) окна следует программный код, который запускает программу. Сначала создается объект, представляющий само приложение. Чтобы снабдить программу ярлыком, в операционной системе Windows используется файл *.ico*, имя которого (и полный путь) передается методу `iconbitmap()`. Но в системах UNIX следует передавать файл с растровым изображением (то есть монохромное изображение). В библиотеке Tk имеется несколько встроенных растровых изображений, поэтому, что-

бы обозначить, что изображение берется из файла в файловой системе, имя файла с изображением необходимо предварять символом @. Затем мы присваиваем заголовок приложению (он будет отображаться в полосе заголовка окна) и создаем экземпляр класса MainWindow, которому в качестве родительского виджета передается ссылка на объект приложения. В заключение вызывается метод protocol(), чтобы определить, какое действие выполнять, когда пользователь щелкнет на кнопке закрытия окна. В данном случае мы определили, что должен вызываться метод MainWindow.quit(). И в самом конце запускается цикл обработки событий – только когда мы достигнем этой точки, окно появится на экране и будет способно откликаться на действия пользователя.

Программы с главным окном

Программ в виде диалогов нередко бывает вполне достаточно для решения простых задач, но с ростом функциональных возможностей программ часто имеет смысл создавать полноценные приложения с главным окном, содержащим полосу меню и панели инструментов. Такие приложения обычно легче поддаются расширению по сравнению с программами в виде диалогов, потому что мы легко можем добавлять новые пункты меню и инструментальные кнопки, не вызывая изменений в размещении виджетов в главном окне.

В этом разделе мы рассмотрим программу *bookmarks-tk.py*, главное окно которой изображено на рис. 13.4. Программа предназначена для ведения списка закладок, каждая из которых представлена парой строк (имя, URL), и дает пользователям возможность добавлять, редактировать и удалять закладки, а также открывать их в веб-броузере.

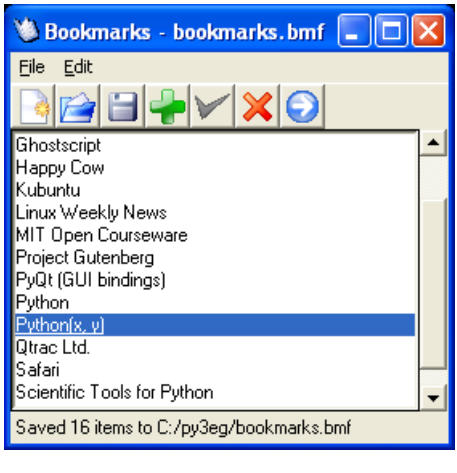

Рис. 13.4. *Программа Bookmarks*

Программа имеет два окна: главное окно – с полосой меню, панелью инструментов, списком закладок и полосой состояния, и окно диалога – для добавления и редактирования закладок.

Создание главного окна

Главное окно напоминает окно диалога в том смысле, что оно также содержит виджеты, которые необходимо создать и разместить. Но, кроме того, нам потребуется создать полосу меню, пункты меню, панель инструментов и полосу состояния, а также методы, выполняющие действия по запросам пользователя. Пользовательский интерфейс целиком создается в методе инициализации главного окна, который мы рассмотрим, разбив его на пять частей, так как он достаточно длинный.

```
class MainWindow:

    def __init__(self, parent):
        self.parent = parent

        self.filename = None
        self.dirty = False
        self.data = {}

        menubar = tkinter.Menu(self.parent)
        self.parent["menu"] = menubar
```

В этом примере, вместо того чтобы наследовать стандартный виджет, как это было сделано в предыдущем примере, мы создадим обычный класс. В случае наследования мы можем переопределять методы наследуемого класса, но когда в этом нет необходимости, мы можем просто использовать прием композиции, как сделано в данном примере. Визуальное представление приложения создается за счет создания переменных экземпляра, содержащих виджеты, которые помещаются внутрь виджета `tkinter.Frame`, как будет показано чуть ниже.

Нам необходимо сохранить следующие данные: ссылку на родительский объект (приложение), имя текущего файла с закладками, флаг наличия изменений (если имеет значение `True`, это свидетельствует о том, что в данных имеются изменения, которые не были сохранены на диск) и сами закладки – словарь, ключами которого служат имена закладок, а значениями – адреса URL.

Чтобы создать полосу меню, необходимо создать объект `tkinter.Menu`, родителем которого является объект-родитель окна, и сообщить родителю, что у него имеется меню. (Может показаться странным, что полоса меню является пунктом меню, но библиотека Tk прошла очень длинный путь развития, и в ней еще остались некоторые пережитки прошлого.) Полосы меню, создаваемые таким способом, не требуют специального размещения – библиотека Tk сделает это самостоятельно.

```
        fileMenu = tkinter.Menu(menubar)
        for label, command, shortcut_text, shortcut in (
```

```
                ("New...", self.fileNew, "Ctrl+N", "<Control-n>"),
                ("Open...", self.fileOpen, "Ctrl+O", "<Control-o>"),
                ("Save", self.fileSave, "Ctrl+S", "<Control-s>"),
                (None, None, None, None),
                ("Quit", self.fileQuit, "Ctrl+Q", "<Control-q>")):
            if label is None:
                fileMenu.add_separator()
            else:
                fileMenu.add_command(label=label, underline=0,
                        command=command, accelerator=shortcut_text)
                self.parent.bind(shortcut, command)
        menubar.add_cascade(label="File", menu=fileMenu, underline=0)
```

Таким способом создаются все полосы меню. Сначала создается объект `tkinter.Menu`, который является дочерним по отношению к полосе меню, а затем в меню добавляются команды и разделители. (Обратите внимание, что акселератором в терминологии библиотеки Tk называется горячая комбинация клавиш и что в параметре `accelerator` задается текст, обозначающий комбинацию клавиш, а вовсе не привязку клавиш.) Параметр `underline` определяет, какой символ должен выводиться с подчеркиванием; в данном случае с подчеркиванием будут выводиться первые символы всех пунктов меню, и они же будут играть роль акселераторов. При добавлении пункта меню (названного командой) мы также определяем горячую комбинацию клавиш, привязывая ее к той же команде, которая будет выполняться при выборе соответствующего пункта меню. В самом конце меню добавляется в полосу меню вызовом метода `add_cascade()`.

Мы опустили определение меню Edit, так как программный код, создающий его, по своей структуре идентичен программному коду, создающему меню File.

```
        frame = tkinter.Frame(self.parent)
        self.toolbar_images = []
        toolbar = tkinter.Frame(frame)
        for image, command in (
                ("images/filenew.gif", self.fileNew),
                ("images/fileopen.gif", self.fileOpen),
                ("images/filesave.gif", self.fileSave),
                ("images/editadd.gif", self.editAdd),
                ("images/editedit.gif", self.editEdit),
                ("images/editdelete.gif", self.editDelete),
                ("images/editshowwebpage.gif", self.editShowWebPage)):
            image = os.path.join(os.path.dirname(__file__), image)
            try:
                image = tkinter.PhotoImage(file=image)
                self.toolbar_images.append(image)
                button = tkinter.Button(toolbar, image=image,
                                        command=command)
                button.grid(row=0, column=len(self.toolbar_images) -1)
            except tkinter.TclError as err:
```

```
            print(err)
        toolbar.grid(row=0, column=0, columnspan=2, sticky=tkinter.NW)
```

Затем создается рабочее пространство, в котором будут размещаться все виджеты окна. Затем создается еще одна область, `toolbar`, содержащая горизонтальную полосу с кнопками с изображениями вместо текстовых надписей, которые будут играть роль инструментальных кнопок. Все кнопки располагаются друг за другом в ячейках сетки, содержащей одну строку и столько столбцов, сколько требуется кнопок. В конце пространство с панелью инструментов помещается в первую строку сетки рабочего пространства окна, с выравниванием по северо-западу, благодаря чему панель всегда будет находиться в верхнем левом углу окна. (Библиотека Tk автоматически помещает полосу меню выше всех виджетов окна.) Схема расположения виджетов в главном окне показана на рис. 13.5, где полоса меню, размещаемая самой библиотекой Tk, выделена белым цветом, а виджеты, размещаемые нами, – серым.

Когда изображение добавляется в кнопку, оно добавляется как слабая ссылка, поэтому сразу после выхода из области видимости эти изображения будут запланированы к утилизации. Нам следует избежать этого эффекта, потому что нам необходимо, чтобы изображения оставались на кнопках после выхода из метода инициализации, поэтому мы создаем переменную экземпляра `self.toolbar_images`, которая будет хранить ссылки на изображения, предохраняя их от исчезновения на протяжении всего времени жизни программы.

По умолчанию библиотека Tk имеет возможность читать файлы изображений только определенных форматов, поэтому мы вынуждены использовать изображения в формате *.gif*.[1] Если какой-то файл с изображением не будет найден, будет возбуждено исключение `tkinter.TclError`, которое необходимо перехватить и обработать, чтобы избежать завершения программы из-за нехватки какого-нибудь изображения.

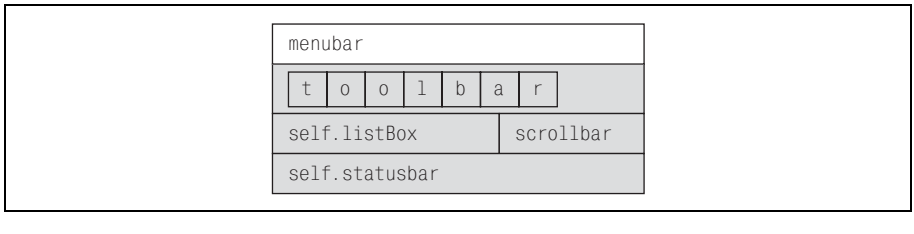

Рис. 13.5. *Схема расположения виджетов в главном окне программы Bookmarks*

[1] Если для библиотеки Tk установлено расширение Python Imaging Library, появляется возможность использовать все современные форматы графических изображений. Подробности можно найти на сайте *www.pythonware.com/products/pil/*.

Обратите внимание, что не все действия, предусматриваемые системой меню, мы сделали доступными в виде кнопок панели инструментов – это распространенная практика.

```
scrollbar = tkinter.Scrollbar(frame, orient=tkinter.VERTICAL)
self.listBox = tkinter.Listbox(frame,
                               yscrollcommand=scrollbar.set)
self.listBox.grid(row=1, column=0, sticky=tkinter.NSEW)
self.listBox.focus_set()
scrollbar["command"] = self.listBox.yview
scrollbar.grid(row=1, column=1, sticky=tkinter.NS)

self.statusbar = tkinter.Label(frame, text="Ready...",
                               anchor=tkinter.W)
self.statusbar.after(5000, self.clearStatusBar)
self.statusbar.grid(row=2, column=0, columnspan=2,
                    sticky=tkinter.EW)

frame.grid(row=0, column=0, sticky=tkinter.NSEW)
```

Центральную область окна (область между панелью инструментов и строкой состояния) занимает виджет списка и ассоциированная с ним полоса прокрутки. Виджет списка размещается с выравниванием по всем направлениям, а полоса прокрутки – только с выравниванием по северу и югу (north-south, то есть вертикально). Оба виджета добавляются в сетку центральной области окна бок о бок.

Нам необходимо гарантировать, что при перемещении по списку с помощью клавиш управления курсором или при изменении позиции движка в полосе прокрутки, оба виджета будут синхронизироваться между собой. Для этого при создании виджета списка в аргументе `yscrollcommand` ему передается метод `set()` полосы прокрутки (чтобы при перемещении по списку полоса прокрутки перемещала бы движок соответственно текущей позиции в списке), а в атрибут `command` полосы прокрутки записывается метод `yview()` виджета списка (чтобы при перемещении движка полосы прокрутки соответственным образом происходила и прокрутка списка).

Полоса состояния – это просто метка. Метод `after()` запускает таймер однократного срабатывания (таймер, который срабатывает всего один раз через указанный интервал времени). В первом аргументе методу передается интервал времени в миллисекундах, а во втором аргументе – функция или метод, который должен быть вызван по истечении указанного времени. Это означает, что сразу после запуска программы, в течение пяти секунд, в строке состояния будет отображаться текст «Ready...», после чего строка состояния будет очищена. Строка состояния размещается в самой последней строке, с выравниванием с запада на восток (west-east, горизонтально).

В самом конце в окно добавляется сама рабочая область. На этом мы завершили создание главного окна и размещение виджетов в нем, но при такой реализации все виджеты по умолчанию будут иметь фикси-

рованный размер и изменение размеров окна не будет приводить к изменениям размеров виджетов. Следующий фрагмент программного кода решает эту проблему и завершает метод инициализации.

```
frame.columnconfigure(0, weight=999)
frame.columnconfigure(1, weight=1)
frame.rowconfigure(0, weight=1)
frame.rowconfigure(1, weight=999)
frame.rowconfigure(2, weight=1)

window = self.parent.winfo_toplevel()
window.columnconfigure(0, weight=1)
window.rowconfigure(0, weight=1)

self.parent.geometry("{0}x{1}+{2}+{3}".format(400, 500,
                                              0, 50))
self.parent.title("Bookmarks - Unnamed")
```

Методы `columnconfigure()` и `rowconfigure()` позволяют определять вес столбцов и строк сетки. Сначала определяется вес для сетки рабочей области окна, первому столбцу и второй строке (где находится виджет списка) придается максимальный вес, поэтому при изменении размеров рабочей области весь избыток пространства будет отдан виджету списка. Только этого еще недостаточно – нам необходимо разрешить изменение размеров окна верхнего уровня, содержащего рабочую область, что мы и сделаем, получив ссылку на объект окна вызовом метода `wininfo_toplevel()` и установив вес строки и столбца равным 1.

В конце метода устанавливаются начальные размеры окна и его положение с помощью строки в формате *ширинахвысота+x+y*. (Если бы нам потребовалось определить только размеры окна, мы могли бы сделать это с помощью строки в формате *ширинахвысота*.) Наконец, установкой заголовка окна мы завершаем создание пользовательского интерфейса.

Если пользователь щелкнет на кнопке в панели инструментов или выберет пункт меню, будет вызван метод, ответственный за выполнение данной операции. Некоторые методы опираются на использование других, вспомогательных методов. Мы поочередно рассмотрим все методы, начав с того, который вызывается через пять секунд после запуска программы.

```
def clearStatusBar(self):
    self.statusbar["text"] = ""
```

Строка состояния – это простой виджет `tkinter.Label`. Для ее очистки мы могли бы в методе `after()` использовать лямбда-выражение, но, так как нам потребуется очищать строку состояния из разных точек программы, мы создали для этого специальный метод.

```
def fileNew(self, *ignore):
    if not self.okayToContinue():
        return
    self.listBox.delete(0, tkinter.END)
```

```
        self.dirty = False
        self.filename = None
        self.data = {}
        self.parent.title("Bookmarks - Unnamed")
```

Если пользователю потребуется создать новый файл с закладками, мы сначала должны предоставить ему возможность сохранить имеющиеся изменения в текущем файле. Эта возможность реализована в виде отдельного метода `MainWindow.okayToContinue()`, потому что она будет использоваться в нескольких местах в программе. Метод возвращает `True`, если можно продолжать создание файла, и `False` – в противном случае. Если можно продолжать, мы очищаем виджет списка, удаляя все записи – от первой до последней, где `tkinter.END` – это константа, которая используется для определения последнего элемента в случаях, когда виджеты могут содержать несколько элементов. Затем сбрасывается флаг наличия изменений, имя файла и словарь с данными, поскольку новый файл еще пустой и никаких изменений еще не было сделано, а затем мы устанавливаем текст заголовка, отражающий тот факт, что был создан новый и еще не сохранявшийся файл.

Переменная `ignore` хранит последовательность из нуля или более позиционных аргументов, которые нам не нужны. В случае когда метод вызывается в результате выбора пункта меню или щелчка на инструментальной кнопке, ему не передается никаких дополнительных аргументов, но в случае вызова по нажатию горячей комбинации клавиш (например, Ctrl+N) ему будет передаваться объект события, а поскольку мы не различаем, как именно пользователь вызывает данное действие, мы просто игнорируем объект события.

```
    def okayToContinue(self):
        if not self.dirty:
            return True
        reply = tkinter.messagebox.askyesnocancel(
                        "Bookmarks - Unsaved Changes",
                        "Save unsaved changes?", parent=self.parent)
        if reply is None:
            return False
        if reply:
            return self.fileSave()
        return True
```

Если пользователь желает выполнить действие, которое приведет к очистке виджета списка (например, когда создается новый или открывается существующий файл), нам необходимо предоставить ему возможность сохранить любые несохраненные изменения. Если содержимое файла не изменялось, следовательно, отпадает необходимость выполнять сохранение и можно сразу же вернуть значение `True`. В противном случае выводится стандартный диалог с сообщением, содержащий кнопки Yes (да), No (нет) и Cancel (отмена). Если пользователь отменяет операцию, в переменную `reply` записывается значение `None`, мы

трактуем это как то, что пользователь не хочет продолжать начатую операцию и не хочет сохранять изменения, и просто возвращаем значение `False`. Если пользователь отвечает согласием, в переменную `replay` записывается значение `True`, поэтому мы даем пользователю возможность сохранить изменения и возвращаем `True`, если изменения были сохранены и `False` – в противном случае. Если пользователь ответил отказом, в переменную `replay` записывается значение `False`, что для нас означает отказ от сохранения изменений, но мы все равно возвращаем `True`, потому что пользователь выразил желание продолжить операцию без сохранения изменений.

Стандартные диалоги библиотеки Tk не импортируются инструкцией `import tkinter`, поэтому для данного метода необходимо добавить инструкцию `import tkinter.messagebox` и инструкцию `import tkinter.filedialog` – для следующего метода. В Windows и Mac OS X используются стандартные диалоги этих операционных систем, а для других платформ используются диалоги, реализованные в самой библиотеке Tk. Мы всегда передаем диалогам ссылку на родительское окно, чтобы при вызове они автоматически располагались в центре родительского окна.

Все стандартные диалоги являются *модальными*, то есть при появлении на экране они становятся единственным окном приложения, с которым пользователь может взаимодействовать, поэтому, чтобы получить возможность продолжить работу с приложением, пользователь должен закрыть их (щелчком на кнопке OK (Готово), Open (Открыть), Cancel (Отменить) или подобной им). Модальные диалоги являются для программиста самой удобной разновидностью диалогов, так как пользователь лишен возможности изменить состояние программы, пока открыто окно диалога, поскольку модальный диалог блокирует приложение, пока не будет закрыт. Слово «блокирует» здесь означает, что инструкция, следующая за вызовом модального диалога, выполнится, только когда диалог будет закрыт.

```
    def fileSave(self, *ignore):
        if self.filename is None:
            filename = tkinter.filedialog.asksaveasfilename(
                            title="Bookmarks - Save File",
                            initialdir=".",
                            filetypes=[("Bookmarks files", "*.bmf")],
                            defaultextension=".bmf",
                            parent=self.parent)
            if not filename:
                return False
            self.filename = filename
            if not self.filename.endswith(".bmf"):
                self.filename += ".bmf"
        try:
            with open(self.filename, "wb") as fh:
                pickle.dump(self.data, fh, pickle.HIGHEST_PROTOCOL)
            self.dirty = False
```

```
            self.setStatusBar("Saved {0} items to {1}".format(
                              len(self.data), self.filename))
            self.parent.title("Bookmarks - {0}".format(
                              os.path.basename(self.filename)))
        except (EnvironmentError, pickle.PickleError) as err:
            tkinter.messagebox.showwarning("Bookmarks - Error",
                    "Failed to save {0}:\n{1}".format(
                    self.filename, err), parent=self.parent)
        return True
```

Если имя текущего файла не задано, мы должны потребовать от пользователя выбрать имя файла. Если пользователь отменяет операцию, мы возвращаем `False`, чтобы показать, что операцию следует отменить. В противном случае мы, если это необходимо, устанавливаем корректное расширение имени файла. Используя существующий или создавая новый файл, мы сохраняем в файле словарь `self.data` в законсервированном виде. После сохранения закладок мы сбрасываем признак наличия изменений, выводим сообщение в строку состояния (которое будет очищено через определенное время, в чем мы вскоре убедимся) и помещаем имя файла в заголовок окна (без пути к нему). Если попытка сохранить файл потерпела неудачу, мы выводим диалог с текстом предупреждения (в котором уже присутствует кнопка `OK`), чтобы проинформировать пользователя о возникшей проблеме.

```
    def setStatusBar(self, text, timeout=5000):
        self.statusbar["text"] = text
        if timeout:
            self.statusbar.after(timeout, self.clearStatusBar)
```

Этот метод выводит указанный текст в строке состояния и, если задано предельное время отображения (по умолчанию – пять секунд), метод запускает таймер однократного срабатывания, который очистит строку состояния по прошествии указанного времени.

```
    def fileOpen(self, *ignore):
        if not self.okayToContinue():
            return
        dir = (os.path.dirname(self.filename)
               if self.filename is not None else ".")
        filename = tkinter.filedialog.askopenfilename(
                        title="Bookmarks - Open File",
                        initialdir=dir,
                        filetypes=[("Bookmarks files", "*.bmf")],
                        defaultextension=".bmf", parent=self.parent)
        if filename:
            self.loadFile(filename)
```

Этот метод начинается точно так же, как и метод `MainWindow.fileNew()`, предоставляя пользователю возможность сохранить имеющиеся изменения или отменить операцию открытия файла. Если пользователь подтверждает свое желание продолжить, мы стараемся предложить

пользователю рационально выбранный каталог, поэтому мы используем каталог, где находится текущий файл, если таковой имеется; в противном случае – текущий рабочий каталог. Аргумент `filetypes` – это список (описание и маска) кортежей из двух элементов, которые отображаются диалогом выбора файла. Если пользователь выберет имя файла, мы запоминаем его выбор и вызываем метод `loadFile()`, который прочитает содержимое файла.

Создание отдельного метода `loadFile()`, позволяющего загружать файлы без привлечения внимания пользователя, – это обычная практика. Например, некоторые программы на запуске автоматически загружают последний использовавшийся файл, а некоторые программы даже сохраняют список последних использовавшихся файлов в виде пунктов меню, чтобы при выборе любого из таких пунктов напрямую вызывался бы метод `loadFile()` с именем файла, ассоциированным с выбранным пунктом меню.

```
def loadFile(self, filename):
    self.filename = filename
    self.listBox.delete(0, tkinter.END)
    self.dirty = False
    try:
        with open(self.filename, "rb") as fh:
            self.data = pickle.load(fh)
        for name in sorted(self.data, key=str.lower):
            self.listBox.insert(tkinter.END, name)
        self.setStatusBar("Loaded {0} bookmarks from {1}".format(
                          self.listBox.size(), self.filename))
        self.parent.title("Bookmarks - {0}".format(
                          os.path.basename(self.filename)))
    except (EnvironmentError, pickle.PickleError) as err:
        tkinter.messagebox.showwarning("Bookmarks - Error",
                "Failed to load {0}:\n{1}".format(
                self.filename, err), parent=self.parent)
```

Когда этот метод вызывается, то точно известно, что любые изменения уже были сохранены или отвергнуты, поэтому можно очистить виджет списка. Мы сохраняем полученное имя файла, очищаем виджет списка и признак наличия изменений и затем производим попытку открыть файл и распаковать его содержимое в словарь `self.data`. После того как данные будут прочитаны, выполняется обход всех имен закладок и добавление их по очереди в виджет списка. В заключение в строку состояния выводится информационное сообщение и обновляется текст заголовка окна.

```
def fileQuit(self, event=None):
    if self.okayToContinue():
        self.parent.destroy()
```

Это метод последнего пункта в меню File. Мы даем пользователю возможность сохранить имеющиеся изменения. Если при этом пользова-

тель отменяет операцию, то ничего не происходит и программа продолжит свою работу. В противном случае мы предписываем родительскому виджету уничтожить себя, что приводит к благополучному завершению программы. Если бы нам потребовалось сохранить пользовательские настройки, это можно было бы сделать здесь, непосредственно перед вызовом метода `destroy()`.

```
def editAdd(self, *ignore):
    form = AddEditForm(self.parent)
    if form.accepted and form.name:
        self.data[form.name] = form.url
        self.listBox.delete(0, tkinter.END)
        for name in sorted(self.data, key=str.lower):
            self.listBox.insert(tkinter.END, name)
        self.dirty = True
```

Этот метод вызывается, если пользователь запрашивает операцию добавления новой закладки (выбором пункта меню Edit→Add (Правка→Добавить), или щелчком на кнопке с изображением  в панели инструментов, или нажатием комбинации клавиш Ctrl+A на клавиатуре). Класс `AddEditForm` – это наш собственный диалог, который будет описываться в следующем подразделе, а пока нам достаточно знать, что он имеет флаг `accepted`, который устанавливается в значение `True`, если пользователь щелкнет на кнопке OK, и `False`, если пользователь щелкнет на кнопке Cancel, а также два атрибута, `name` и `url`, в которых хранятся имя и адрес URL закладки, добавляемой или редактируемой пользователем.

Мы создаем новый экземпляр диалога `AddEditForm`, который тут же появляется на экране как модальный диалог – вследствие чего дальнейшее выполнение приложения блокируется и инструкция `if form.accepted ...` не будет выполнена, пока диалог не будет закрыт.

Если в диалоге `AddEditForm` пользователь щелкнет на кнопке OK и при этом укажет имя закладки, мы добавляем новую закладку с указанным именем и адресом URL в словарь `self.data`. Затем мы очищаем виджет списка и снова вставляем в него все закладки в отсортированном порядке. Возможно, более эффективно было бы просто вставлять новую закладку сразу в нужное место, но, даже когда имеется несколько сотен закладок, на современной машине различия будут едва заметны. В конце мы устанавливаем флаг наличия несохраненных изменений, поскольку теперь у нас появилось изменение, которое еще не было сохранено.

```
def editEdit(self, *ignore):
    indexes = self.listBox.curselection()
    if not indexes or len(indexes) > 1:
        return
    index = indexes[0]
    name = self.listBox.get(index)
    form = AddEditForm(self.parent, name, self.data[name])
    if form.accepted and form.name:
```

```
            self.data[form.name] = form.url
            if form.name != name:
                del self.data[name]
                self.listBox.delete(0, tkinter.END)
                for name in sorted(self.data, key=str.lower):
                    self.listBox.insert(tkinter.END, name)
            self.dirty = True
```

Редактирование представляет собой немного более сложную операцию, чем добавление новой закладки, потому что сначала нам необходимо отыскать закладку, которую пользователь желает отредактировать. Метод `curselection()` возвращает список (возможно пустой) позиций всех выделенных элементов в виджете списка. Если выделен только один элемент, мы запоминаем его текст, то есть имя закладки (а также ключ в словаре `self.data`), которую пользователь собрался отредактировать. Затем создается новый экземпляр диалога `AddEditForm`, которому передается имя и адрес URL закладки.

После того как диалог будет закрыт, если пользователь указал непустое имя закладки и щелкнул на кнопке `OK`, выполняется обновление словаря `self.data`. Если имя закладки осталось прежним, можно просто установить флаг наличия несохраненных изменений и выйти (в данном случае предполагается, что пользователь изменил только адрес URL), но если имя закладки изменилось, мы удаляем элемент словаря, ключом которого является прежнее имя, очищаем виджет списка и вновь заполняем его информацией о закладках, как мы делали это в методе добавления новой закладки.

```
    def editDelete(self, *ignore):
        indexes = self.listBox.curselection()
        if not indexes or len(indexes) > 1:
            return
        index = indexes[0]
        name = self.listBox.get(index)
        if tkinter.messagebox.askyesno("Bookmarks - Delete",
                                       "Delete '{0}'?".format(name)):
            self.listBox.delete(index)
            self.listBox.focus_set()
            del self.data[name]
            self.dirty = True
```

Чтобы удалить закладку, мы сначала должны отыскать закладку, выбранную пользователем, поэтому данный метод начинается точно так же, как и метод `MainWindow.editEdit()`. Если была выбрана всего одна закладка, мы выводим диалог, спрашивая пользователя, действительно ли он желает удалить закладку. Если пользователь отвечает утвердительно, функция вызова диалога возвращает значение `True`, мы удаляем закладку из виджета списка и из словаря `self.data` и устанавливаем флаг наличия несохраненных изменений. Кроме того, мы возвращаем фокус ввода в виджет списка.

```
    def editShowWebPage(self, *ignore):
        indexes = self.listBox.curselection()
        if not indexes or len(indexes) > 1:
            return
        index = indexes[0]
        url = self.data[self.listBox.get(index)]
        webbrowser.open_new_tab(url)
```

Когда пользователь вызывает этот метод, мы отыскиваем выбранную закладку и извлекаем соответствующий адрес URL из словаря `self.data`. Затем, с помощью функции `webbrowser.open_new_tab()` из модуля `webbrowser`, мы открываем веб-броузер, передавая ему указанный адрес URL. Если перед этим веб-броузер еще не был открыт, он будет автоматически запущен.

```
application = tkinter.Tk()
path = os.path.join(os.path.dirname(__file__), "images/")
if sys.platform.startswith("win"):
    icon = path + "bookmark.ico"
    application.iconbitmap(icon, default=icon)
else:
    application.iconbitmap("@" + path + "bookmark.xbm")
window = MainWindow(application)
application.protocol("WM_DELETE_WINDOW", window.fileQuit)
application.mainloop()
```

Последние строки программы похожи на последние строки в программе *interest-tk.pyw*, которая рассматривалась ранее, за исключением трех различий. Первое различие заключается в том, что когда пользователь щелкнет на кнопке закрытия окна, в программе Bookmarks, в отличие от программы Interest, будет вызван другой метод. Другое отличие состоит в том, что в системе Windows метод `iconbitmap()` имеет дополнительный аргумент, позволяющий указывать пиктограмму по умолчанию, которая будет отображаться в заголовках всех окон приложения – в системах UNIX это делать необязательно, так как это происходит автоматически. И последнее различие заключается в том, что мы определяем текст, отображаемый в заголовке окна приложения, внутри методов класса `MainWindow`, а не здесь. В программе Interest заголовок окна программы никогда не изменялся, поэтому было достаточно установить его только один раз, но в программе Bookmark мы изменяем текст заголовка, включая в него имя текущего файла с закладками.

Теперь, когда мы рассмотрели реализацию класса главного окна и программный код, выполняющий инициализацию программы и запускающий цикл обработки событий, можно обратить свое внимание на диалог `AddEditForm`.

Создание собственного диалога

Диалог `AddEditForm` представляет собой средство, с помощью которого пользователь может добавлять и редактировать имена закладок и адреса URL. Окно диалога показано на рис. 13.6, где диалог используется для редактирования существующей закладки (отсюда слово «Edit» (правка) в заголовке окна). Тот же диалог может использоваться для добавления закладок. Сначала рассмотрим метод инициализации, разбив его на четыре части.

```
class AddEditForm(tkinter.Toplevel):

    def __init__(self, parent, name=None, url=None):
        super(AddEditForm, self).__init__(parent)
        self.parent = parent
        self.accepted = False
        self.transient(self.parent)
        self.title("Bookmarks - " + (
                   "Edit" if name is not None else "Add"))
        self.nameVar = tkinter.StringVar()
        if name is not None:
            self.nameVar.set(name)
        self.urlVar = tkinter.StringVar()
        self.urlVar.set(url if url is not None else "http://")
```

Мы решили унаследовать класс `tkinter.TopLevel` – виджет, который создан, чтобы служить базовым классом для виджетов, которые будут играть роль окон верхнего уровня. (Как отмечалось ранее, мы могли бы использовать вызов `super().__init__(parent)`, но так как этот прием уже не работает, мы вызываем функцию `super()`, явно передавая ей родительский класс и объект `self` в виде аргументов.) Мы сохраняем ссылку на родительский виджет, создаем атрибут `self.accepted` и присваиваем ему значение `False`. Вызов метода `transient()` выполняется, чтобы проинформировать родительское окно, что данное окно всегда должно появляться поверх родительского окна. В зависимости от того, было ли передано имя закладки и адрес URL, в заголовок окна записывается текст, отражающий выполняемую операцию – добавление или редактирование. Затем создаются два объекта `tkinter.StringVar` – для хранения имени закладки и адреса URL, которые инициализируются полученными значениями, если диалог используется для редактирования.

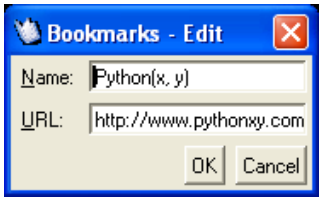

Рис. 13.6. *Диалог добавления/редактирования в программе Bookmarks*

```
frame = tkinter.Frame(self)
nameLabel = tkinter.Label(frame, text="Name:", underline=0)
nameEntry = tkinter.Entry(frame, textvariable=self.nameVar)
nameEntry.focus_set()
urlLabel = tkinter.Label(frame, text="URL:", underline=0)
urlEntry = tkinter.Entry(frame, textvariable=self.urlVar)
okButton = tkinter.Button(frame, text="OK", command=self.ok)
cancelButton = tkinter.Button(frame, text="Cancel",
                              command=self.close)

nameLabel.grid(row=0, column=0, sticky=tkinter.W, pady=3,
               padx=3)
nameEntry.grid(row=0, column=1, columnspan=3,
               sticky=tkinter.EW, pady=3, padx=3)
urlLabel.grid(row=1, column=0, sticky=tkinter.W, pady=3,
              padx=3)
urlEntry.grid(row=1, column=1, columnspan=3,
              sticky=tkinter.EW, pady=3, padx=3)
okButton.grid(row=2, column=2, sticky=tkinter.EW, pady=3,
              padx=3)
cancelButton.grid(row=2, column=3, sticky=tkinter.EW, pady=3,
                  padx=3)
```

Виджеты создаются и располагаются в сетке, как показано на рис. 13.7. Виджеты ввода текста имени и адреса URL ассоциированы с соответствующими переменными типа `tkinter.StringVar`, а две кнопки связаны с методами `self.ok()` и `self.close()`, которые будут показаны немного ниже.

```
frame.grid(row=0, column=0, sticky=tkinter.NSEW)
frame.columnconfigure(1, weight=1)
window = self.winfo_toplevel()
window.columnconfigure(0, weight=1)
```

Изменять размеры диалога имеет смысл только по горизонтали, поэтому для второй колонки рабочей области окна мы указали, что она допускает изменение размера по горизонтали, установив вес колонки равным 1. Это означает, что при изменении горизонтального размера рабочей области окна все дополнительное пространство будет отдаваться виджетам в колонке с порядковым номером 1 (виджетам ввода имени и адреса URL). Точно так же для колонки самого окна мы указали,

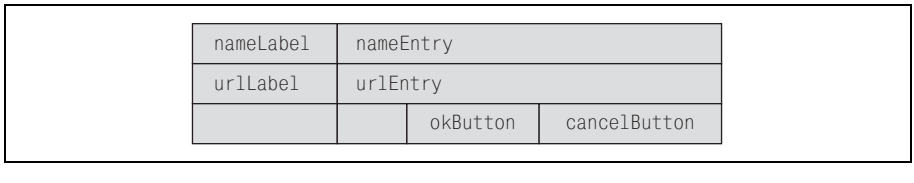

Рис. 13.7. *Схема расположения виджетов в окне диалога добавления/редактирования программы Booktmarks*

что она допускает изменение размера по горизонтали, установив ее вес равным 1. Если пользователь попытается изменить вертикальный размер диалога, все виджеты сохранят свое положение относительно друг друга и сместятся в центр окна по вертикали. Но если пользователь попытается изменить горизонтальный размер диалога, виджеты ввода имени и адреса URL изменят свой горизонтальный размер, уменьшившись или растянувшись по горизонтали так, чтобы привести свои размеры в соответствие с доступным пространством.

```
self.bind("<Alt-n>", lambda *ignore: nameEntry.focus_set())
self.bind("<Alt-u>", lambda *ignore: urlEntry.focus_set())
self.bind("<Return>", self.ok)
self.bind("<Escape>", self.close)

self.protocol("WM_DELETE_WINDOW", self.close)
self.grab_set()
self.wait_window(self)
```

Ранее были созданы две метки, Name: и URL:, показывающие, что имеются две горячие комбинации клавиш Alt+N и Alt+U, нажатие которых вызывает перемещение фокуса ввода в соответствующий виджет ввода текста. Чтобы подкрепить это конкретной реализацией, мы добавили необходимые привязки клавиш. Вместо прямого вызова метода `focus_set()` мы использовали лямбда-функции, чтобы иметь возможность игнорировать объект события, который автоматически передается в виде аргумента. Мы также предусмотрели стандартные привязки клавиш (Enter и Esc) для кнопок OK и Cancel.

Метод `protocol()` использован, чтобы определить, какой метод должен вызываться, когда пользователь попытается покинуть диалог щелчком на кнопке закрытия окна. Оба вызова `grab_set()` и `wait_window()` необходимы, чтобы сделать диалог модальным.

```
def ok(self, event=None):
    self.name = self.nameVar.get()
    self.url = self.urlVar.get()
    self.accepted = True
    self.close()
```

Этот метод будет вызван, если пользователь щелкнет на кнопке OK (или нажмет клавишу Enter). Он скопирует текст из ассоциированных переменных типа `tkinter.StringVar` в соответствующие переменные экземпляра (которые создаются только сейчас), в переменную `self.accepted` будет записано значение `True` и будет вызван метод `self.close()`, который закроет диалог.

```
def close(self, event=None):
    self.parent.focus_set()
    self.destroy()
```

Этот метод вызывается из метода `self.ok()`, а также когда пользователь щелкнет на кнопке закрытия окна или на кнопке Cancel (или нажмет

клавишу Esc). Он передает фокус ввода родительскому окну и уничтожает диалог. В данном случае слово «уничтожит» означает, что будут уничтожены только само окно и все виджеты, содержащиеся в нем, сам же экземпляр класса `AddEditForm` продолжит свое существование, так как вызывающая программа по-прежнему имеет ссылку на него.

После закрытия диалога вызывающая программа проверит значение переменной `accepted` и, если оно равно `True`, извлечет имя и адрес URL, которые были добавлены или отредактированы. Затем, как только поток выполнения покинет метод `MainWindow.editAdd()` или `MainWindow.editEdit()`, объект `AddEditForm` выйдет из области видимости и будет запланирован для утилизации механизмом сборки мусора.

В заключение

В этой главе вы получили некоторое представление о программировании графического интерфейса с использованием библиотеки Tk. Основное преимущество библиотеки Tk заключается в том, что она входит в комплект поставки Python как стандартный компонент. Но она имеет множество недостатков, из которых не самый последний заключается в том, что эта старинная библиотека работает несколько иначе, чем большинство более современных альтернатив.

Если вы плохо знакомы с принципами программирования графического интерфейса, запомните, что основными кросс-платформенными альтернативами библиотеке Tk являются библиотеки PyGTK, PyQt и wxPython, намного более простые в освоении и использовании, причем все они позволяют добиться лучших результатов при меньшем объеме программного кода. Более того, для всех этих альтернатив библиотеке Tk имеется и более качественная документация, учитывающая специфику языка Python, они содержат намного больше виджетов с более привлекательным внешним видом и позволяют создавать собственные виджеты с нуля, обеспечивая возможность целиком и полностью контролировать их внешний вид и поведение.

Несмотря на то, что библиотеку Tk удобно использовать при создании очень маленьких программ или когда в распоряжении программиста имеется только стандартная библиотека Python, тем не менее во всех остальных случаях любая из кросс-платформенных библиотек будет лучшим выбором.

Упражнения

Для выполнения первого упражнения необходимо будет скопировать и модифицировать программу Bookmarks, которая была продемонстрирована в этой главе. Во втором упражнении будет предложено создать программу с графическим интерфейсом с самого начала.

1. Скопируйте программу *bookmarks-tk.pyw* и модифицируйте ее так, чтобы она могла импортировать и экспортировать файлы DBM, которые используются программой *bookmarks.py* (созданной в главе 11). Добавьте в меню File два новых пункта – Import и Export. Не забудьте выполнить привязку горячих комбинаций клавиш для них (имейте в виду, что комбинация Ctrl+E уже используется для пункта меню Edit→Edit). Кроме того, добавьте на панель инструментов соответствующие кнопки. Для этого в метод инициализации придется добавить порядка пяти строк программного кода.

   Дополнительно потребуется создать два метода – `fileImport()` и `fileExport()`, общий размер которых составляет немногим меньше 60 строк, включая обработку ошибок. При реализации операции импортирования вы сами можете решить, как выполнять импортирование – объединять импортируемые закладки с существующими или замещать их. Реализация сама по себе несложная, но требует некоторого внимания. Решение (в котором импортируемые закладки объединяются с существующими) приводится в файле *bookmarks-tk_ans.py*.

2. В главе 12 мы видели, как создавать и использовать регулярные выражения для поиска соответствий в тексте. Создайте программу с графическим интерфейсом в виде диалога, которая могла бы использоваться для ввода и тестирования регулярных выражений, как показано на рис. 13.8.

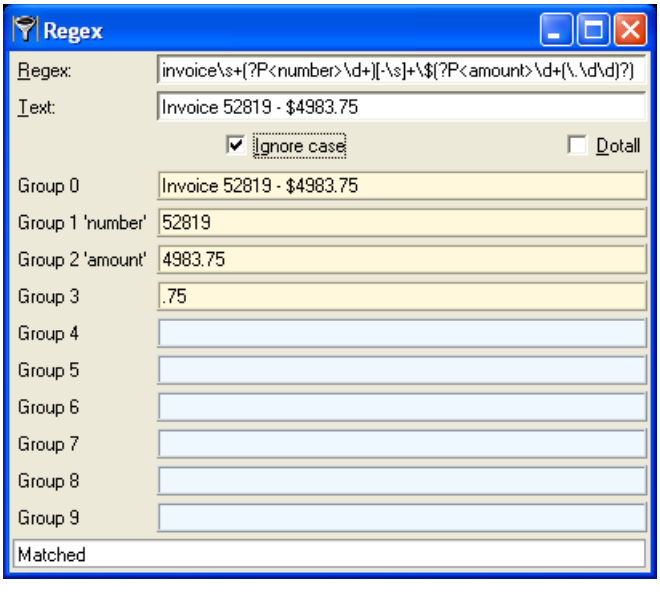

***Рис. 13.8.** Окно программы Regex*

Вам потребуется ознакомиться с документацией к модулю `re`, так как программа должна вести себя корректно как при вводе регулярных выражений, содержащих ошибки, так и при выполнении итераций по группам, так как в большинстве случаев в регулярных выражениях количество сохраняющих групп меньше, чем меток, предусмотренных для отображения их содержимого. Обеспечьте в программе полную поддержку клавиатуры, включая переход в виджеты ввода текста с использованием комбинаций Alt+R и Alt+T, управление флажками с использованием комбинаций Alt+I и Alt+D, завершение программы с использованием комбинаций Ctrl+Q и Esc и пересчет результатов при нажатии и отпускании клавиш в любом из виджетов ввода текста, а также при изменении состояния любого из флажков.

Программа не очень сложная, хотя над программным кодом, отображающим совпадения и номера групп (и имена, там где они указаны), придется немного поломать голову. Решение приводится в файле *regex-tk.py*, содержащем примерно сто сорок строк программного кода.

Эпилог

Если вы прочитали хотя бы первые шесть глав и либо выполнили упражнения, либо написали несколько своих собственных программ на языке Python 3, вы уже обладаете неплохим фундаментом, на основе которого сможете расширять свой дальнейший опыт и навыки в соответствии со своими потребностями – Python не будет сковывать ваше продвижение!

Для углубления знаний и улучшения навыков программирования на языке Python, если вы прочитали только первые шесть глав, вам обязательно нужно ознакомиться с материалом главы 7 и прочитать хотя бы часть главы 8 – в частности, раздел, где описываются менеджеры контекста и инструкция `with`.

Стоит иметь в виду, что процесс разработки с чистого листа не дает ничего, кроме чувства гордости и расширения знаний, так как необходимость в этом при использовании языка Python возникает крайне редко. Мы уже упоминали стандартную библиотеку и каталог пакетов Python Package Index, *pypi.python.org/pypi*, содержащих огромный объем функциональных возможностей. Кроме того, значительное число советов, подсказок и идей предлагается в справочнике рецептов «Python Cookbook» по адресу *code.activestate.com/recipes/langs/python/*, хотя на момент написания этой книги он был ориентирован на использование Python 2.

Помимо всего прочего, существует возможность создавать модули для языка Python на других языках программирования (на любых, которые могут экспортировать функции C, что доступно во многих языках). Эти модули могут разрабатываться для совместной работы с языком Python посредством C API языка Python. С помощью модуля `ctypes` из программного кода на языке Python можно получить доступ к функциональным возможностям, заключенным в библиотеках совместного использования (библиотеки DLL в Windows), наших собственных или полученных от сторонних разработчиков, что дает нам практически неограниченный доступ к огромному количеству функциональных возможностей, которые можно получить в Интернете благодаря умениям и великодушию программистов, создающих свободные программные продукты.

Если у вас появится желание поучаствовать в жизни сообщества Python, можно начать с сайта *www.python.org/community*, где вы найдете массу справочных материалов и списки рассылки по интересам.

Алфавитный указатель

Специальные символы

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

А

Б

В

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Ю